IST-2002-507932

ECRYPT

European Network of Excellence in Cryptology

Network of Excellence

Information Society Technologies

D.VAM.1
Performance Benchmarks

Due date of deliverable: 31. July 2005
Actual submission date: 3. August 2005

Start date of project: 1. February 2004 Duration: 4 years

Lead contractor: Gemplus S.A. (G+)

Revision 1.1

Project co-funded by the European Commission within the 6th Framework Programme		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission services)	
RE	Restricted to a group specified by the consortium (including the Commission services)	
CO	Confidential, only for members of the consortium (including the Commission services)	

Performance Benchmarks

Editor
Marc Joye (G+)

Contributors
Roberto Avanzi (RUB), Lejla Batina (KUL), Benoît Chevallier-Mames (G+), Nicolas Courtois (SLB), Claus Diem (IEM), Rob Granger (BRIS), Johann Großschädl (IAIK), Marc Joye (G+), Tanja Lange (DTU), Nele Mentens (KUL), Roger Oyono (IEM), Dan Page (BRIS), Jan Pelzl (RUB), Andy Rupp (RUB), Stefan Tillich (IAIK)

3. August 2005
Revision 1.1

The work described in this report has in part been supported by the Commission of the European Communities through the IST program under contract IST-2002-507932.

Contents

1 Primitives Based on Integer Factorization **3**
- 1.1 RSA primitive . . . 3
- 1.2 The need for paddings . . . 7
- 1.3 RSA-OAEP encryption scheme . . . 8
- 1.4 RSA-PSS signature scheme . . . 8
- 1.5 GQ signature scheme . . . 9

2 Primitives Based on Discrete Logarithm in a Finite Field **11**
- 2.1 Reduction modulo pseudo-Mersenne numbers . . . 11
- 2.2 Modular exponentiation . . . 13
- 2.3 Primitives over prime fields . . . 14

3 Primitives Based on Cyclotomic Subgroups **17**
- 3.1 XTR primitive . . . 17
- 3.2 CEILIDH primitive . . . 18

4 Primitives Based on Pairings **21**
- 4.1 Pairing evaluation . . . 21
- 4.2 Pairing based schemes . . . 28

5 Primitives Based on (Hyper-)Elliptic Curves **33**
- 5.1 Elliptic curve cryptography . . . 33
- 5.2 Elliptic curve based schemes . . . 36
- 5.3 Hyperelliptic curves . . . 38

6 Primitives Based on Non-Hyperelliptic Curves **41**
- 6.1 Fast arithmetic . . . 41
- 6.2 Construction of secure non-hyperelliptic curves of genus 3 . . . 52
- 6.3 Solving the DLP in Jacobians of non-hyperelliptic genus 3 curves . . . 54
- 6.4 Conclusion . . . 57

7 Primitives Based on Multivariate Polynomials **59**
- 7.1 Introduction . . . 59
- 7.2 Multivariate-polynomial based schemes . . . 60

Bibliography **63**

A Basic Algorithms **69**
A.1 Multiple-precision multiplication 69
A.2 Modular multiplication 74
A.3 Modular exponentiation 78

B Index to Notations **79**
B.1 General symbols 79
B.2 Costs 79
B.3 Acronyms 80

Executive summary

This is the final VAM-1 report on performance benchmarks for cryptosystems. In order not to overlap with European project NESSIE, a special focus has been made on public-key cryptography. The report presents detailed operation counting for basic operations in Appendix A. These figures are then used to derive the requirements for various cryptosystems in Chapters 1–7. Our approach presents the tremendous advantage of being independent of the underlying hardware or compiler.

Chapter 1

Primitives Based on Integer Factorization

1.1 RSA primitive

The RSA cryptosystem is the first realization of a public-key cryptosystem [68]. Let κ be a security parameter and let $N = pq$ be the product of two (large) $\frac{\kappa}{2}$-bit primes p and q. Let also a *public* exponent e such that $\gcd(e, (p-1)(q-1)) = 1$ and the corresponding *private* exponent $d \equiv e^{-1} \pmod{\lambda(N)}$ where λ denotes Carmichael's function (i.e., $\lambda(N) = \mathrm{lcm}(p-1, q-1)$).

In the following analysis, we assume that we are working on a Ω-bit CPU. To ease the presentation, we suppose w.l.o.g that κ can be written as $\kappa = \Omega k$ for an even integer k.

1.1.1 Plain RSA public exponentiation

The public operation consists in computing $y = x^e \bmod N$ where e is an ℓ-bit integer. Commonly used values for e are $e = 3$, 17 or $2^{16} + 1$ (e.g., see [1]).

With the square-and-multiply algorithm, an RSA exponentiation costs roughly, for a form-free ℓ-bit exponent e

$$\boxed{{}^3\!/_2\, \ell\, \mathsf{M}_{\mathbb{Z}_N}}$$

on average, where $\mathsf{M}_{\mathbb{Z}_N}$ denotes a multiplication modulo N, and only $(\ell+1)\mathsf{M}_{\mathbb{Z}_N}$ when $e = 2^\ell + 1$.

Using Montgomery representation (see Appendix A), for a (form-free) ℓ-bit exponent e, the square-and-multiply algorithm (Algorithm A.8) needs, on average, $3\ell/2$ Montgomery multiplications (modulo N) for evaluating $y = x^e \bmod N$ plus two more Montgomery multiplications for the normalization and denormalization steps.

So, assuming that

C1. interleaved Montgomery multiplication is used and parameters n'_0 and $\mathfrak{R}$ ($= R^2 \bmod N$) are precomputed (see Algorithm A.4);

C2. Montgomery squaring is carried out with Montgomery multiplication algorithm: no additional trick is used and so the Montgomery squaring and multiplication have the same complexity;

C3. exponentiation is done with the square-and-multiply algorithm (Algorithm A.8);

C4. cost of data reading/writting in various memory types is neglected;

a plain RSA public exponentiation $x^e \bmod N$ for an ℓ-bit integer requires

$$\boxed{2 + 3/2\,\ell \text{ Montgomery multiplications (modulo } N)}$$

on average, that is,

$$\boxed{k(2k+1)(2 + 3/2\,\ell) \text{ CPU multiplications}\,.}$$

Concerning the working-memory requirements, in addition to a few CPU registers, *under Assumptions C1–C4 and C5*

C5. writing in rewritable memory (i.e., EEPROM-type memory) is not permitted;

a plain RSA public exponentiation requires

$$\boxed{\left(4k + 2 + \tfrac{\ell}{\Omega}\right) \text{ words of working memory}\,.}$$

Let `MEM0`, `MEM1` and `MEM2` be three k-word memory buffers and let `TMP0` be a $(k+2)$-word temporary buffer (used for Montgomery multiplication). Let also `mem0` an ℓ/Ω-word memory buffer for storing ℓ-bit exponent e. For example, a plain RSA public exponentiation can be obtained within the previous memory requirements as:

1. load x in `MEM0`, $\mathfrak{R} = R^2 \bmod N$ in `MEM1`, and N in `MEM2`;
   $[\texttt{MEM0} \leftarrow x; \texttt{MEM1} \leftarrow \mathfrak{R}; \texttt{MEM2} \leftarrow N]$
2. load e in `mem0`;
   $[\texttt{mem0} \leftarrow e]$
3. normalize x with `TMP0` as temporary buffer and recopy the result, MontMult(`MEM0`, `MEM1`, `MEM2`), in `MEM0`;
   $[\texttt{MEM0} \leftarrow \widetilde{x}]$
4. recopy the value of `MEM0` in `MEM1`;
   $[\texttt{MEM1} \leftarrow \widetilde{x}]$
5. apply the (Montgomery) square-and-multiply loop (Algorithm A.8, Lines 2–7) with `TMP0` as temporary buffer for the Montgomery multiplication;
   $[\texttt{MEM0} \leftarrow \widetilde{x^e}]$
6. denormalize $\widetilde{x^e}$ with `TMP0` as temporary buffer as MontMult(`MEM0`, 1, `MEM2`) and recopy the result in `MEM0`;
   $[\texttt{MEM0} \leftarrow y]$

1.1.2 Plain RSA private exponentiation

Standard mode The private operation is $y^d \bmod N$ where, by construction, the size of d is supposed to be the same as that of N. Therefore the previous analysis applies for $\ell = \kappa = \Omega k$ and so the cost is roughly

$$\boxed{3/2\,\Omega k\,\mathsf{M}_{\mathbb{Z}_N}\,.}$$

More precisely, using Montgomery representation, *under Assumptions C1–C5*,

$$\boxed{2 + 3/2\,\Omega k \text{ Montgomery multiplications (modulo } N)}$$

are required, on average, that is,

$$\boxed{k(2k+1)(2 + 3/2\,\Omega k) \text{ CPU multiplications}}$$

and the memory requirements become

$$\boxed{(5k+2) \text{ words of working memory}\,.}$$

CRT mode A more efficient way for evaluating the private exponentiation $y^d \bmod N$ is via the Chinese Remainder Theorem (CRT) [66]. Knowing the factorisation of $N = pq$, $y^d \bmod N$ can be obtained as

$$y^d \bmod N = s_q + q\big(i_q(s_p - s_q) \bmod p\big) \quad \text{with} \begin{cases} s_p = y^{d_p} \bmod p \\ s_q = y^{d_q} \bmod q \end{cases}$$

and $d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$ and $i_q = q^{-1} \bmod p$.

Since by assumption $|p|_2 = |q|_2 = |d_p|_2 = |d_q|_2 = \kappa/2 = \Omega k/2$, it follows that the computation of x_p (resp. x_q) with the square-and-multiply algorithm roughly requires on average $(2 + 3/2\,\Omega k/2)$ Montgomery multiplications modulo p (resp. modulo q). Hence, as these Montgomery multiplications deal with numbers of $k/2$ words, the computation of s_p and of s_q costs roughly

$$\boxed{1/4\,k(k+1)(8+3\Omega k) \text{ CPU multiplications}\,.}$$

In other words, neglecting the cost of the recombination, we see that the cost of the CRT mode is approximatively a quarter of the cost of the standard mode. This is why CRT mode is most of the time preferred to standard mode.

A detailed procedure for evaluating $y^d \bmod N$ using CRT is given below.

Let `MEM0`, `MEM1`, `MEM2`, `MEM3` and `mem0` be five $k/2$-word memory buffers and let `TMP0` be a $(k/2+2)$-word temporary buffer (used for Montgomery multiplication). Then,

1. consider $y = y_1 R + y_0$ with $y_i < R$ $(i \in \{1,2\})$ and $R = 2^{\Omega\, k/2}$;
2. load y_1 in `MEM1`, $\mathfrak{R}_q = R^2 \bmod q$ in `MEM2`, q in `MEM3`, and $d_q \in$ `mem0`;
   $[\texttt{MEM1} \leftarrow y_1; \texttt{MEM2} \leftarrow \mathfrak{R}_q; \texttt{MEM3} \leftarrow q; \texttt{mem0} \leftarrow d_q]$
3. normalize y_1 with `TMP0` as temporary buffer, MontMult(`MEM1`, `MEM2`, `MEM3`);
   $[\texttt{TMP0} \leftarrow \widetilde{y_1} = y_1 R \bmod q]$

4. load y_0 in MEM1 and compute $y_1R + y_0 \pmod q$ in MEM1;
$$[\texttt{MEM1} \leftarrow y_1R + y_0 \pmod q]$$

5. normalize it with TMP0 as temporary buffer and recopy the result, MontMult(MEM1, MEM2, MEM3), in MEM1 and in MEM2;
$$[\texttt{MEM1} \leftarrow \widetilde{y} = yR \bmod q; \texttt{MEM2} \leftarrow \widetilde{y}]$$

6. apply the (Montgomery) square-and-multiply loop (Algorithm A.8, Lines 2–7) with TMP0 as temporary buffer for the Montgomery multiplication, and get the result, $\widetilde{s_q} = s_qR \bmod q$, in MEM1;
$$[\texttt{MEM1} \leftarrow \widetilde{s_q}]$$

7. denormalize $\widetilde{s_q}$ with TMP0 as temporary buffer as MontMult(MEM1, 1, MEM3) and recopy the result in MEM0;
$$[\texttt{MEM0} \leftarrow s_q]$$

8. repeat Steps 2–6 for the modulo p computation and get the value of $\widetilde{s_p} = s_pR \bmod p$ in MEM1;
$$[\texttt{MEM1} \leftarrow \widetilde{s_p}; \texttt{MEM3} \leftarrow p]$$

9. load $\mathfrak{R}_p = R^2 \bmod p$ in MEM2 and normalize s_q modulo p (that is, compute $\widehat{s_q} := s_qR \bmod p$) as MontMult(MEM0, MEM2, MEM3) with TMP0 as temporary buffer, and recopy the result in MEM2;
$$[\texttt{MEM2} \leftarrow \widehat{s_q} = s_qR \bmod p]$$

10. compute $\widetilde{s_p} - \widehat{s_q} \pmod p$ in MEM1;
$$[\texttt{MEM1} \leftarrow \widetilde{s_p} - \widehat{s_q} \pmod p]$$

11. load i_q in MEM2, multiply it by $\widetilde{s_p} - \widehat{s_q}$ as MontMult(MEM1, MEM2, MEM3) with TMP0 as temporary buffer, and recopy the result in MEM1;
$$[\texttt{MEM1} \leftarrow i_q(s_p - s_q) \bmod p]$$

12. load q in MEM3 and compute integer multiplication (e.g., using Algorithm A.2) of q and $(i_q(s_p - s_q) \bmod p)$ as IntMult(MEM3, MEM1) with MEM2 as working register;
$$[(\texttt{MEM2}, \texttt{MEM3}) \leftarrow q \cdot (i_q(s_p - s_q) \bmod p)]$$

13. add s_q and get the result in MEM0.
$$[\texttt{MEM0} \leftarrow \mathrm{CRT}(s_p, s_q) = y^d \bmod N]$$

To sum up, again *under Assumption C1'*

C1'. interleaved Montgomery multiplication is used and parameters $n'_{0,p} = -p^{-1} \bmod 2^\Omega$, $n'_{0,q} = -q^{-1} \bmod 2^\Omega$, $\mathfrak{R}_p$ and $\mathfrak{R}_q$ are precomputed

and Assumptions C2–C5, the CRT mode of operation requires on average

$7 + {}^3\!/_2\,\Omega k$ Montgomery multiplications (modulo p or q) and 1 integer multiplication

that is,

${}^1\!/_4\,k\big[(k+1)(14+3\Omega k)+k\big]$ CPU multiplications

and

$$\boxed{(3k+2) \text{ words of working memory}}$$

where k denotes the number of words of modulus N.

There are of course many possible variants of the described implementation. One of these is to make use of Barrett algorithm [15] for the normalisation steps and the CRT recombination. The resulting implementation should be slightly faster but requires more code memory.

1.2 The need for paddings

The RSA primitive is highly *malleable*, which is a undesirable property for security reasons.

Needs for redundancy A simple example against the RSA primitive goes as follows. Anybody can take the Alice's public key (e, N), and sign the message $m = r^e \bmod N$ for on her behalf: namely, $s = r$ is the valid signature of this message m. This is called an *existential forgery*.

Another example of attack on signature is the following: if Alice has made a valid signature s of a message m, anybody can create a new couple: for a new random r, $s' = s \cdot r \bmod N$ is the valid signature on message $m' = m \cdot r^e \bmod N$. This is called a *chosen-message attack*.

For encryption, there is a similar problem. Imagine Bob gets a ciphertext $c = m^e \bmod N$, encrypted under Alice's pubic key. Bob wants to recover the message m, without asking Alice to decrypt c directly. So, Bob creates a new ciphertext $c' = c \cdot r^e \bmod N$, for a non-zero random r, and asks to Alice to decrypt c'. Hence Bob recovers the corresponding plaintext message $m' = m \cdot r \bmod N$. As message m' is likely meaningless, Alice will put message m' in the garbage. If Bob has access to the garbage of Alice, he recovers plaintext message m as $m = m'/r \bmod N$. This is called a *chosen-ciphertext attack*.

These examples show that one needs to ensures non-malleability of the primitive: roughly speaking, one wants to use *redundancy* to avoid an attacker to create *ex-nihilo* signatures of messages (first example), or to create a valid signature from another one (second example), or to create a valid ciphertext from another one (third example).

Needs for randomness For encryption, there is an additional issue. Imagine that Alice wants to cipher to Bob whether he needs to buy or cell a firm stock. Eve wants to recover the information, as she knows that Alice has some secret information about the firm. If she looks at the ciphertext c, she can *guess* what was the original message ("buy" or "sell"), by just encrypting the two values and comparing them to c. This is an attack against the *indistinguishability*.

This last example shows that (encryption) paddings need to be *probabilistic*, that is, contain a random of sufficient size, so that it is impossible to decide whether a ciphertext is the encryption of a given message.

Paddings Paddings are the solution that is used to achieve both redundancy and randomness, and obtain security of signature and encryption schemes, against most powerful attackers.

1.3 RSA-OAEP encryption scheme

OAEP (for Optimal Asymmetric Encryption Padding) is the most famous padding for encryption, especially for RSA. It was proposed by Bellare and Rogaway in 1994 [16], and proved to have the highest security level [74, 42], namely against indistinguishability under chosen-ciphertext attacks, in the random oracle model.

RSA-OAEP is tuned with two security parameters κ_1 and κ_2. It makes use of two hash functions $\mathcal{G} : \{0,1\}^{\kappa_2} \to \{0,1\}^{|N|_2-\kappa_2-1}$ and $\mathcal{H} : \{0,1\}^{|N|_2-\kappa_2-1} \to \{0,1\}^{\kappa_2}$. Messages being encrypted are of size $\ell = |N|_2 - \kappa_1 - \kappa_2 - 1$.

1.3.1 RSA-OAEP encryption

The set of plaintexts is $\mathcal{M} = \{0,1\}^{\ell}$. The encryption algorithm uses a random coin r from the set $\mathcal{R} = \{0,1\}^{\kappa_2}$ and outputs a ciphertext c. On input a plaintext $m \in \mathcal{M}$, it computes $s = (m\|0^{\kappa_1}) \oplus \mathcal{G}(r)$ and $t = r \oplus \mathcal{H}(s)$, and finally the ciphertext $c = (0\|s\|t)^e \bmod N$.

1.3.2 RSA-OAEP decryption

On input a ciphertext c, the decryption algorithm first computes $b\|s\|t = c^d \bmod N$, where $s \in \{0,1\}^{|N|_2-\kappa_2-1}$, $t \in \{0,1\}^{\kappa_2}$ and b is a single bit. Then, it computes $r = t \oplus \mathcal{H}(s)$ and $(m\|v) = s \oplus \mathcal{G}(r)$. If $v = 0^{\kappa_1}$ and $b = 0$, the ciphertext is valid and message m is returned, else $\perp$ is returned indicating that the ciphertext is invalid.

Remark that κ_1 is the parameter that tunes the redundancy while κ_2 is the parameter that tunes the randomness.

1.3.3 Requirements

As we can see, the padding itself is absolutely not very consuming (in terms of both memory and speed), compared with the exponentiation itself: it basically requires two calls to hash functions. As a result, noting that a hash evaluation can be carried out in the buffer required for an RSA exponentiation and neglecting the time needed for a hash evaluation, the analysis given in Section 1.1 remains valid.

1.4 RSA-PSS signature scheme

PSS (for Probabilistic Signature Scheme) is the most famous padding for signature with RSA. It was proposed and proved to be secure in the random oracle model by Bellare and Rogaway in 1996 [17].

RSA-PSS is tuned with two security parameters κ_1 and κ_2. It make use of three hash functions $\mathcal{F} : \{0,1\}^{\kappa_1} \to \{0,1\}^{\|n\|-\kappa_1-\kappa_2-1}$, $\mathcal{G} : \{0,1\}^{\kappa_1} \to \{0,1\}^{\kappa_2}$ and $\mathcal{H} : \{0,1\}^{\star} \times \{0,1\}^{\kappa_2} \to \{0,1\}^{\kappa_1}$. Messages being signed are of arbitrary length, contrary to the encryption case (here, we only consider PSS with appendix and refer the reader to [17] for PSS *with message recovery*)

1.4.1 RSA-PSS signing

The set of plaintexts is $\mathcal{M} = \{0,1\}^{*}$. The signature algorithm makes use of a random coin r from the set $\mathcal{R} = \{0,1\}^{\kappa_2}$ and outputs a signature s: on a plaintext $m \in \mathcal{M}$. On input a

message m, it computes $u = \mathcal{H}(m, r)$, $v = r \oplus \mathcal{G}(u)$ and $w = \mathcal{F}(u)$, and finally the signature $s = (0\|u\|v\|w)^d \bmod N$.

1.4.2 RSA-PSS verification

To verify a signature s on a message $m \in \{0,1\}^*$, the verification algorithm computes $(b\|u\|v\|w) = s^e \bmod N$, where b is a single bit, $u \in \{0,1\}^{\kappa_1}$ and $v \in \{0,1\}^{\kappa_2}$. Next, it computes $r = v \oplus \mathcal{G}(u)$, and accepts the signature iff $u = \mathcal{H}(m, r)$ and $w = \mathcal{F}(u)$.

As in OAEP, we remark that κ_1 is the parameter tuning the redundancy while κ_2 is the parameter tuning the randomness.

1.4.3 Requirements

Again, as for OAEP, the PSS padding is not consuming and, neglecting the calls to the hash functions, the analysis given in Section 1.1 applies.

1.5 GQ signature scheme

The GQ signature scheme (named after its inventors Guillou and Quisquater) [49] is based on RSA.

Let κ and ℓ be two security parameters. Let $N = pq$ be the product of two (large) $\frac{\kappa}{2}$-bit primes p and q. Let $e \geq 2^\ell$ be a public exponent such that $\gcd(e, (p-1)(q-1)) = 1$ and $d \equiv e^{-1} \pmod{\lambda(N)}$ be the corresponding private exponent. Let also J be a random element $\mathbb{Z}_N^*$ and B be the e-th root of J^{-1} modulo N, i.e., $B = J^{-d} \bmod N$. Finally, let $\mathcal{G}$ be a hash function $\{0,1\}^* \to \{0,1\}^\ell$.

The public key is (e, J, N) while the private key is B. Advantageously, modulus N and public exponent e can be shared between users and J can represent a unique identifier for a given user, which simplifies the management of public key infrastructure.

1.5.1 GQ signing

To sign a message m, the signer first generates a random $k \in \mathbb{Z}_N^*$ and computes $r = k^e \bmod N$. Then, he/she computes $h = \mathcal{G}(m, r)$, and finally $s = kB^h \bmod N$. The signature is $\sigma = (h, s)$.

1.5.2 GQ verification

The verification of the signature is as follows. The verifier computes $r' = s^e J^h \bmod N$. Then, if $\mathcal{G}(m, r') = h$, the signature is accepted.

1.5.3 Requirements

Compared to RSA, GQ presents the advantage of only involving exponentiations with ℓ-bit exponents (typically $\ell = 160$) in the signature generation. An analysis similar as the one in Section 1.1 shows that, neglecting the call of the hash function, the generation of a signature requires on average

$4 + 3\,\ell$ Montgomery multiplications (modulo N)

that is,

$$\boxed{k(2k+1)(4+3\ell) \text{ CPU multiplications}}$$

and

$$\boxed{(5k+2+\ell/\Omega) \text{ words of working memory}}$$

where k denotes the number of words of modulus N.

In a recent version, Guillou and Quisquater propose a similar scheme (called GQ2), but non-identity based, which allows to use Chinese remaindering and so gives rise to further speed-ups and memory savings.

Chapter 2

Primitives Based on Discrete Logarithm in a Finite Field

2.1 Reduction modulo pseudo-Mersenne numbers

A pseudo-Mersenne (PM) number m is a number of the form $m = \beta^t - c$, where c is a small integer. These numbers, introduced by Crandall [32], are highly suited for modular reduction due to their special (custom) form. They allow to employ very fast reduction techniques which are not applicable to general numbers. The efficiency of the reduction operation modulo a PM number $m = \beta^t - c$ is based on the relation

$$\beta^t \equiv c \bmod m$$

which means that any occurence of β^t in an integer $Z \geq \beta^t$ can be substituted by the much smaller "offset" c.

To give an example, let us assume that Z is the product of two integers which are smaller than $m = \beta^t - c$ and c is of length at most $t/2$ digits. Thus, the product Z has at most $2t$ digits. Let us write the k-digit integer Z as $Z_H \cdot \beta^t + Z_L$, where Z_H represents the $k - t$ most significant digits and Z_L the t least significant digits of Z. The basic reduction step is accomplished by multiplying Z_H and c together and "folding" the product $Z_H \cdot c$ into Z_L:

$$Z = Z_H \cdot \beta^t + Z_L \equiv Z_H \cdot c + Z_L \pmod{m} \ .$$

This leads to a new expression for the residue class with a length of at most $3/2 \cdot t$ digits. By repeating this substitution a few times and performing a final subtraction of m one obtains the fully reduced result $Z \mod m$. An implementation of this repeated substitution is depicted in Algorithm 1.

2.1.1 Complexity

The implementation depicted in Algorithm 1 is adapted from the iterative division algorithm in [62] (a similar method is described in [60]). Note that $Z \bmod \beta^t$ can be efficiently computed by assigning the t least significant digits of Z to Z_L. Similarly, $Z_H \leftarrow \lfloor Z/\beta^t \rfloor$ is trivial to compute by shifting Z t digits to the right and assigning the result to Z_H.

The number of loop iterations depends on the magnitude of A. It can be shown that the loop iterates at most twice when A is of size $k \leq 2t$ digits and c of size $l \leq t/2$ digits. In

Algorithm 1 Reduction modulo a pseudo-Mersenne number.

Require: t-digit modulus $m = \beta^t - c$ with $\beta = 2^\Omega$,
l-digit positive integer c for some $l < t$,
and a k-digit operand $A \geq m$
Ensure: $\text{PMRED}(A, m, c) = A \bmod m$
1: $Z \leftarrow A$
2: **while** $Z \geq \beta^t$ **do**
3: $Z_L \leftarrow Z \bmod \beta^t$
4: $Z_H \leftarrow \lfloor Z/\beta^t \rfloor$
5: $Z \leftarrow Z_H \cdot c + Z_L$
6: **end while**
7: **if** $Z \geq m$ **then**
8: $Z \leftarrow Z - m$
9: **end if**
10: **return** Z

each iteration one multiplication by c is required. Assuming the above restrictions on k and l, the first iteration of the loop requires at most $t \cdot l$ CPU multiplications, when schoolbook-multiplication is used (what is reasonable, since c usually consists of very few digits) to compute $Z_H \cdot c$. In the second iteration at most l^2 CPU multiplications are required, since the product Z resulting from the previous iteration of the loop is of size at most $t+l$ digits. Hence, the reduction $\text{PMRED}(A, m, c)$ of a $2t$-digit integer A modulo a t-digit pseudo-Mersenne number $m = \beta^t - c$, where c is of size $l \leq t/2$ digits, requires at most

$$\boxed{(t \cdot l + l^2) \text{ CPU multiplications}.}$$

Thus, if we assume that c is a single-precision integer (i.e., $l = 1$), as proposed by Crandall [32], applying a reduction modulo m costs at most

$$\boxed{(t + 1) \text{ CPU multiplications}.}$$

Larger values of the operand A may require additional iterations. In general, any iteration of the loop decreases the length of Z by $t - l$ digits. However, it is reasonable to restrict our consideration to a $2t$-digit operand A, since usually we apply this reduction after each multiplication or addition of two numbers which are smaller than m.

2.1.2 Memory

Let A be a k-digit integer. Then we need k words of memory for the variable Z, t words for Z_L and $k - t$ words for Z_H. Neglecting the memory needed to store A, m and c the reduction $\text{PMRED}(A, m, c)$ of a k-digit integer A modulo a t-digit pseudo-Mersenne number $m = \beta^t - c$ requires

$$\boxed{(2k) \text{ words of working memory}.}$$

2.1.3 Analysis of the algorithms

The most time consuming instructions used to implement modular multiplication operations are multiplication and addition In Table 2.1, we tabulate the numbers of these instructions

Table 2.1: Comparison of instructions for Montgomery multiplication algorithms.

Algorithm	# MUL	# ADD
Mont. multiplication (CIOS method)	$2k^2 + 2k$	$4k^2 + 6k + 2$
KCM multiplication	$k^2 + k + X_1$	$2k^2 + 4k + 1 + X_2$
Mult. mod. PM (Karat., $\log_\beta c \leq 1$)	$\frac{3}{4}k^2 + k + 1$	$4k^2 + 2k + 2$

needed to implement CIOS and KCM methods for modular multiplication and squaring operations (cf. Appendix A.2).

2.2 Modular exponentiation

The plain square & multiply method is described in Algorithm A.7 (in appendix). Algorithm A.8 depicts the same algorithm for the use with Montgomery residues. In the worst case, both algorithms require l squarings and l multiplications. On average, $l/2$ multiplications are required.

2.2.1 Fixed window method

The fixed window method is also referred to as k-ary exponentiation algorithm. Different from the plain square & multiply algorithm, Algorithm 2 processes one operand in chunnks of w bits.

Algorithm 2 Fixed window method.

Require: $N, x < N$ and $e = \sum_{j=0}^{\ell-1} e_j \, (2^w)^j$ where $e_j \in \{0, 1, ..., 2^w - 1\}$ and $e_{\ell-1} = 1$
Ensure: $y = x^e \bmod N$
Precomputation:
1: $x_0 \leftarrow 1$
2: **for** $j = 1$ to $(2^w - 1)$ **do**
3: $\quad x_j \leftarrow x_{j-1} \cdot x \bmod N$
4: **end for**
Main algorithm:
5: $A \leftarrow 1$
6: **for** $j = \ell$ down to 0 **do**
7: $\quad A \leftarrow A^{2^w} \bmod N$
8: $\quad A \leftarrow A \cdot x_{e_j} \bmod N$
9: **end for**
10: **return** A

2.2.2 Variable window method

The variable window method (Algorithm 3) is also known as sliding-window method and reduces the amount of precomputations as well as the average number of multiplications compared to Algorithm 2.

Algorithm 3 Variable window method.

Require: $N, x < N$, an integer $w \geq 1$, and
$e = \sum_{j=0}^{\ell-1} e_j\, 2^j$ where $e_j \in \{0,1\}$ and $e_{\ell-1} = 1$
Ensure: $y = x^e \bmod N$
Precomputation:
1: $x_1 \leftarrow x$
2: **for** $j = 1$ to $(2^{w-1} - 1)$ **do**
3: $x_{2j+1} \leftarrow x_{2j-1} \cdot x_2 \bmod N$
4: **end for**
Main algorithm:
5: $A \leftarrow 1$, $j \leftarrow \ell$
6: **while** $j \geq 0$ **do**
7: **if** $e_j = 0$ **then**
8: $A \leftarrow A^2$, $j \leftarrow j - 1$
9: **else**
10: $(e_j \neq 0)$, find longest bitstring $e_j e_{j-1} \ldots e_m$ such that $\begin{cases} j - m + 1 \leq w \\ e_m = 1 \end{cases}$
and compute $A \leftarrow A^{2^{j-m+1}} \cdot g_{e_j e_{j-1} \ldots e_m}$, $j \leftarrow m - 1$
11: **end if**
12: **end while**
13: **return** A

The sliding window exponentiation algorithm is potentially very fast. It is very difficult to convey an accurate idea about the complexity of this algorithm. [69] provides a good overview of the complexity of this algorithm.

2.2.3 Comparison of exponentiation algorithms

Table 2.2 lists the number of required modular multiplications (including squarings) for the most important exponentiation methods displayed in Algorithms A.8, 2, and 3.

Table 2.2: Comparison of instructions for modular exponentiation algorithms

Algorithm	Precomputation	Exponentiation (avg.)
A.8	0	$\frac{3}{2}\ell$
2	2^{w-1}	$\left\lfloor \frac{\ell-1}{w} \right\rfloor (w + 1 - 2^{-w}) + 1 - 2^{-((\ell-1) \bmod w)}$
3	2^{w-1}	$2^{w-1} + \frac{\ell}{w+1} + \ell - w$

2.3 Primitives over prime fields

2.3.1 Diffie-Hellman key agreement

The Diffie-Hellman key agreement (a.k.a. Diffie-Hellman key exchange) is used for unauthenticated key agreement [34]. Several variations of unauthenticated Diffie-Hellman key agreement,

providing shared symmetric keys for subsequent cryptographic use are standardized in ANSI X9.42 [2]. We consider the basic protocol which provides protection from eavesdroppers, but not from active adversaries capable of intercepting, modifying, or injecting messages.

The global setup consists of an appropriate prime p and an element $\alpha \in \mathbb{Z}_p^*$ of order q, where $q \mid (p-1)$ is an ℓ-bit integer. Two parties, A and B, can then construct a common secret key K as follows. Party A chooses a random secret x $(1 \leq x \leq q-1)$, computes $K_x = \alpha^x \pmod{p}$, and sends it to B. Likewise, B chooses a random secret y $(1 \leq y \leq q-1)$, computes $K_y = \alpha^y \pmod{p}$, and sends it to A. Parties A and B can then form the secret K from their respective secrets as $K = (K_y)^x \pmod{p}$ and $K = (K_x)^y \pmod{)p}$.

The cost for a Diffie-Hellman key exchange is

2 modular exponentiations per party (ℓ-bit exponent) .

If the value of α^x (resp. α^y) is precomputed, the protocol reduces to 1 modular exponentiation per party (ℓ-bit exponent).

With x, y being an ℓ-bit exponent, the computational complexity is similar to a plain RSA exponentiation (no CRT, no short exponents, cf. Section 1.1). Depending on the exponentiation algorithm and the multiplication algorithm, the overall complexity can be computed with Tables 2.2 and 2.1.

2.3.2 ElGamal public-key encryption

The ElGamal encryption scheme [37] can be considered as extension of the Diffie-Hellman key exchange. Prior to encryption, each party has to create a matching pair of public key and private key. To set up the system, Alice generates a large prime p and an element $\alpha \in \mathbb{F}_p^*$ of order q, where q is an ℓ-bit integer. Alice then randomly chooses $a \in [1, q-1]$ and computes $y = \alpha^a \pmod{p}$. Alice's public key is $\{p, \alpha, y = \alpha^a\}$ while her private is a.

If now Bob wants to send a message to Alice, he first obtain Alice's public key $\{p, \alpha, y\}$ and represent the message as an integer $m \in [0, p-1]$. Next, Bob computes $\gamma = \alpha^k \pmod{p}$ and $\delta = m \cdot y^k \pmod{p}$. The ciphertext is the pair $c = (\gamma, \delta)$.

Upon receiving ciphertext $c = (\gamma, \delta)$, Alice can recover the corresponding plaintext using her private key as $m = \gamma^{-a} \cdot \delta \pmod{p}$.

The ElGamal encryption requires

2 modular exponentiations (ℓ-bit exponent)

for the computation of $\alpha^k \bmod p$ and $y^k \bmod p$. Depending on the exponentiation algorithm and the multiplication algorithm, the overall complexity can be computed with Tables 2.2 and 2.1. These exponentiations can be sped up by selecting random exponents having low Hamming weights. Remark that the ciphertext is twice as long as the corresponding plaintext.

The ElGamal decryption requires

1 modular exponentiation (ℓ-bit exponent)

for the computation of $\gamma^{-a} \equiv \gamma^{q-a} \pmod{p}$.

2.3.3 Digital signature algorithm (DSA)

The digital signature algorithm (DSA) was proposed by NIST in 1991 and has been standardized in the digital signature standard (DSS) in FIPS 186 [4]. The generation of the public and private key is similar to the one described in § 2.3.2 (see also [60, Algorithm 11.54]). The signature generation and verification goes as follows.

DSA signing To sign a message m, Alice does the following.

a) Select a random (secret) integer $k, 1 \leq k \leq q-1$;

b) Compute $r = (\alpha^k \bmod p) \bmod q$;

c) Compute $k^{-1} \bmod q$;

d) Compute $s = k^{-1}[\mathsf{H}(m) + a \cdot r] \bmod q$;

e) Alice's signature on m is the pair (r, s).

DSA verification The signature can be verified as follows.

a) Obtain Alice's public key $\{p, q, \alpha, y\}$;

b) Verify that $0 < r, s < q$ and if not, reject the signature;

c) Compute $w = s^{-1} \bmod q$ and $\mathsf{H}(m)$;

d) Compute $u_1 = w \cdot \mathsf{H}(m) \bmod q$ and $u_2 = r \cdot w \bmod q$;

e) Compute $v = (\alpha^{u_1} y^{u_2} \bmod p) \bmod q$;

f) Accept the signature iff $v = r$.

Related to FIPS 186, the size of q is fixed at 160 bits. However, the 160-bit operations are relatively minor compared to the exponentiation but DSA has the advantage, that this exponentiation can be precomputed.

The computational expensive part for the signature generation is

2 modular exponentiations (160-bit exponent), one modulo p and one modulo q.

Chapter 3

Primitives Based on Cyclotomic Subgroups

3.1 XTR primitive

In common with CEILIDH (cf. Section 3.2), XTR [57] (see also [77]) is based on the cyclotomic subgroup $\mathbb{G}_{p,6}$. Let g be a generator for this subgroup. In XTR elements of $\langle g \rangle$ are represented by their trace over $\mathbb{F}_{p^2}$

$$\mathrm{Tr}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(g) = g + g^{p^2} + g^{p^4} \in \mathbb{F}_{p^2}\,,$$

and hence need only two elements of $\mathbb{F}_p$ to specify. The set of traces constitute the XTR 'group'. Clearly,

$$\mathrm{Tr}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(g) = \mathrm{Tr}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(g^{p^2}) = \mathrm{Tr}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(g^{p^4})\,,$$

and so given an element in the XTR group one can not distinguish between g and its conjugates g^{p^2} and g^{p^4}. Hence decompression to $\mathbb{F}_{p^6}$ is not unique, though this can easily be resolved.

The analogue of the DLP is to compute n given $\mathrm{Tr}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(g)$ and $\mathrm{Tr}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(g^n)$. One can convert this to an ordinary DLP by mapping both back to $\mathbb{F}^*_{p^6}$ by finding the correct root of

$$X^3 - \mathrm{Tr}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(g)X^2 + \mathrm{Tr}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(g)^p X - 1 = (X-g)(X-g^{p^2})(X-g^{p^4})\,,$$

and similarly for $\mathrm{Tr}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(g^n)$.

The real benefit of the XTR representation is the ease with which arithmetic can be performed. Let $c_n = \mathrm{Tr}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(g^n)$. To compute c_n given c_1, one uses some properties of third order addition chains over $\mathbb{F}_{p^2}$ applied to the recurrence

$$c_{u+v} = c_u c_v - c_u^p c_{u-v} + c_{u-2v}\ .$$

As a consequence, exponentiations can be performed faster than with an optimal representation of $\mathbb{F}_{p^6}$ [78, 79]. The main drawback of XTR is that one cannot perform straightforward multiplication since the set of traces is not a group. By keeping track of the correct conjugate however, this can be accomplished. Rather than being based on the torus T_6, XTR may be viewed algebraically as a quotient of T_6 by an action of the symmetric group S_3 [70]. What makes XTR possible is that the trace map from this quotient variety to $\mathbb{F}_{p^2}$ provides an explicit rational parameterisation. For various algorithmic improvements and implementation

tricks for XTR we refer the reader to [78]. Examining the results of Stam [77], we find that an XTR exponentiation by the ℓ-bit exponent N using the standard, binary algorithm costs roughly

$$\boxed{8\ell\,\mathsf{M}_{\mathbb{F}_p}}$$

where $\mathsf{M}_{\mathbb{F}_p}$ denote a multiplication in $\mathbb{F}_p$, while using the Euclidean algorithm this is reduced to

$$\boxed{\sim 6.62\ell\,\mathsf{M}_{\mathbb{F}_p}\,.}$$

3.2 CEILIDH primitive

Much of this section is a slightly abridged version of the original description of CEILIDH [70]. We point out that the construction given there for the rationality of T_6 also provides an explicit rational parametrisation for the torus T_2 *en passant*.

Fix $x \in \mathbb{F}_{p^2} \setminus \mathbb{F}_p$, so $\mathbb{F}_{p^2} = \mathbb{F}_p(x)$, and let $\{\alpha_1, \alpha_2, \alpha_3\}$ be a basis for $\mathbb{F}_{p^3}$ over $\mathbb{F}_p$. Then $\{\alpha_1, \alpha_2, \alpha_3, x\alpha_1, x\alpha_2, x\alpha_3\}$ is a basis for $\mathbb{F}_{p^6}$ over $\mathbb{F}_p$. Let $\sigma \in \mathrm{Gal}(\mathbb{F}_{p^6}/\mathbb{F}_p)$ be the element of order two. Define $\psi_0 : \mathbb{A}^3(\mathbb{F}_p) \hookrightarrow \mathbb{F}_{p^6}$ by

$$\psi_0(u_1, u_2, u_3) = \frac{\gamma + x}{\gamma + \sigma(x)}$$

where $\gamma = u_1\alpha_1 + u_2\alpha_2 + u_3\alpha_3$. Then $\mathrm{N}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^3}}(\psi_0(\mathbf{u})) = 1$ for every $\mathbf{u} = (u_1, u_2, u_3)$. Let $U = \{\mathbf{u} \in \mathbb{A}^3 : \mathrm{N}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(\psi_0(\mathbf{u})) = 1\}$. Then, $\psi_0(\mathbf{u}) \in T_6(\mathbb{F}_p)$ if and only if $\mathbf{u} \in U$, so restricting ψ_0 to U gives a morphism $\psi_0 : U \longrightarrow T_6$. It follows from Hilbert's Theorem 90 that every element of $T_6(\mathbb{F}_p) \setminus \{1\}$ is in the image of ψ_0, and so ψ_0 defines an isomorphism

$$\psi_0 : U \xrightarrow{\sim} T_6 \setminus \{1\}\ .$$

The equation defining U is a quadratic hypersurface in u_1, u_2, u_3. Fix a point $\mathbf{a} = (a_1, a_2, a_3) \in U(\mathbb{F}_p)$. By adjusting the basis $\{\alpha_1, \alpha_2, \alpha_3\}$ of $\mathbb{F}_{p^6}$ if necessary, one can assume without loss of generality that the tangent plane at $\mathbf{a}$ to the surface U is just the plane $u_1 = a_1$. If $(v_1, v_2) \in \mathbb{F}_p \times \mathbb{F}_p$, then the intersection of U with the line $\mathbf{a} + t(1, v_1, v_2)$ consists of two points, namely $\mathbf{a}$ and a point of the form $\mathbf{a} + \frac{1}{f(v_1,v_2)}(1, v_1, v_2)$ where $f(v_1, v_2) \in \mathbb{F}_p[v_1, v_2]$ is an explicit polynomial independent of p. The map that takes (v_1, v_2) to the latter point is a birational isomorphism

$$g : \mathbb{A}^2 \setminus V(f) \xrightarrow{\sim} U \setminus \{\mathbf{a}\}$$

where $V(f)$ denotes the subvariety of $\mathbb{A}^2$ defined by $f(v_1, v_2) = 0$. Thus $\psi_0 \circ g$ defines an isomorphism

$$\psi : \mathbb{A}^2 \setminus V(f) \xrightarrow{\sim} T_6 \setminus \{1, \psi_0(\mathbf{a})\}\ .$$

For the inverse isomorphism, suppose that $\beta = \beta_1 + \beta_2 x \in T_6(\mathbb{F}_p) \setminus \{1, \psi_0(\mathbf{a})\}$ with $\beta_1, \beta_2 \in \mathbb{F}_{p^3}$. One can check that $\beta_2 \neq 0$, and if $\gamma = (1 + \beta_1)/\beta_2$, then $\gamma/\sigma(\gamma) = \beta$. Write $(1 + \beta_1)/\beta_2 = u_1\alpha_1 + u_2\alpha_2 + u_3\alpha_3$ with $u_i \in \mathbb{F}_p$, and define

$$\rho(\beta) = \left(\frac{u_2 - a_2}{u_1 - a_1}, \frac{u_3 - a_3}{u_1 - a_1}\right)\ .$$

Table 3.1: Costs for CEILIDH arithmetic operations.

Operation	Cost
Multiply	$18\mathsf{M}_{\mathbb{F}_p} + 53\mathsf{A}_{\mathbb{F}_p}$
Square	$6\mathsf{M}_{\mathbb{F}_p} + 21\mathsf{A}_{\mathbb{F}_p}$
Inverse	$2\mathsf{A}_{\mathbb{F}_p}$
Frobenius	$1\mathsf{A}_{\mathbb{F}_p}$

Then $\rho : T_6(\mathbb{F}_p) \setminus \{1, \psi_0(\mathbf{a})\} \xrightarrow{\sim} \mathbb{A}^2(\mathbb{F}_p) \setminus V(f)$ is the inverse isomorphism of ψ, and hence we have an efficient compression and decompression mechanism for all (bar two) elements of $T_6(\mathbb{F}_p)$.

Using the results of Granger et al. [47] which yield the arithmetic costs shown in Table 3.1, we find that a CEILIDH exponentiation by the ℓ-bit exponent N using a Joint Sparse Form (JSF) [76] representation costs roughly

$$\boxed{\tfrac{\ell}{2} \cdot (6\mathsf{M}_{\mathbb{F}_p} + 21\mathsf{A}_{\mathbb{F}_p}) + \tfrac{\ell}{4} \cdot (18\mathsf{M}_{\mathbb{F}_p} + 53\mathsf{A}_{\mathbb{F}_p})}$$

where $\mathsf{M}_{\mathbb{F}_p}$ and $\mathsf{A}_{\mathbb{F}_p}$ respectively denote a multiplication and an addition in $\mathbb{F}_p$.

Chapter 4

Primitives Based on Pairings

4.1 Pairing evaluation

Let E be an elliptic curve over a finite field $\mathbb{F}_q$, and let $\mathcal{O}$ denote the identity element of the associated group of rational points $E(\mathbb{F}_q)$. We include a table of suitable supersingular curve parameterisations in Table 4.1. For a positive integer $l \mid \#E(\mathbb{F}_q)$ coprime to q, let $\mathbb{F}_{q^k}$ be the smallest extension field of $\mathbb{F}_q$ which contains the l-th roots of unity in $\overline{\mathbb{F}}_q$. Also, let $E(\mathbb{F}_q)[l]$ denote the subgroup of $E(\mathbb{F}_q)$ of all points of order dividing l, and similarly for the degree k extension of $\mathbb{F}_q$. To unify the notation used in most protocols, we use two groups $\mathbb{G}_1$ and $\mathbb{G}_2$. The group $\mathbb{G}_1$ will always be a subgroup of an elliptic curve group, whereas the group $\mathbb{G}_2$ is a subgroup of the multiplicative group of a finite field.

For the benchmarks of pairing evaluation, we use the notation described in Appendix B.2.

Table 4.1: Parameters for pairing evaluation in different characteristics.

Name	Condition	Curve	Order	k
		Characteristic 2		
Type-A	$q = \pm 1 \pmod 8$	$y^2 + y = x^3 + x$	$2^q + 1 + f$	4
Type-B	$q \neq \pm 1 \pmod 8$	$y^2 + y = x^3 + x$	$2^q + 1 + g$	4
Type-C	$q = \pm 1 \pmod 8$	$y^2 + y = x^3 + x + 1$	$2^q + 1 - f$	4
Type-D	$q \neq \pm 1 \pmod 8$	$y^2 + y = x^3 + x + 1$	$2^q + 1 - g$	4
		Characteristic 3		
Type-A	$q = \pm 1 \pmod{12}$	$y^2 = x^3 - x + 1$	$3^q + 1 + f$	6
Type-B	$q \neq \pm 1 \pmod{12}$	$y^2 = x^3 - x + 1$	$3^q + 1 + g$	6
Type-C	$q = \pm 1 \pmod{12}$	$y^2 = x^3 + x - 1$	$3^q + 1 - f$	6
Type-D	$q \neq \pm 1 \pmod{12}$	$y^2 = x^3 + x - 1$	$3^q + 1 - g$	6
		Characteristic p		
Type-A		$y^2 = x^3 + 1$	$p + 1$	2
Type-B		$y^2 = x^3 + x$	$p + 1$	2

4.1.1 BKLS algorithm

From an efficiency perspective, k is usually chosen to be even [14]. For a thorough treatment of the following, we refer the reader to [14] and also [43], and to [75] for an introduction to divisors.

The reduced Tate pairing of order l is the map

$$e_l : E(\mathbb{F}_q)[l] \times E(\mathbb{F}_{q^k})[l] \to \mathbb{F}_{q^k}^*/(\mathbb{F}_{q^k}^*)^l ,$$

given by $e_l(P,Q) = f_{P,l}(\mathcal{D})$. Here $f_{P,l}$ is a function on E whose divisor is equivalent to $l(P) - l(\mathcal{O})$, $\mathcal{D}$ is a divisor equivalent to $(Q) - (\mathcal{O})$, whose support is disjoint from the support of $f_{P,l}$, and $f_{P,l}(\mathcal{D}) = \prod_i f_{P,l}(P_i)^{a_i}$, where $\mathcal{D} = \sum_i a_i P_i$. It satisfies the following properties:

- For each $P \neq \mathcal{O}$ there exists $Q \in E(\mathbb{F}_{q^k})[l]$ such that $e_l(P,Q) \neq 1 \in \mathbb{F}_{q^k}^*/(\mathbb{F}_{q^k}^*)^l$ (*non-degeneracy*).
- For any integer n, $e_l([n]P,Q) = e_l(P,[n]Q) = e_l(P,Q)^n$ for all $P \in E(\mathbb{F}_q)[l]$ and $Q \in E(\mathbb{F}_{q^k})[l]$ (*bilinearity*).
- Let $L = hl$. Then $e_l(P,Q)^{(q^k-1)/l} = e_L(P,Q)^{(q^k-1)/L}$.
- It is efficiently computable.

The non-degeneracy condition requires that Q is not a multiple of P, i.e., that Q is in some order l subgroup of $E(\mathbb{F}_{q^k})$ disjoint from $E(\mathbb{F}_q)[l]$. When one computes $f_{P,l}(\mathcal{D})$, the value obtained belongs to the quotient group $\mathbb{F}_{q^k}^*/(\mathbb{F}_{q^k}^*)^l$, and not $\mathbb{F}_{q^k}^*$. In this quotient, for a and b in $\mathbb{F}_{q^k}^*$, $a \sim b$ if and only if there exists $c \in \mathbb{F}_{q^k}^*$ such that $a = bc^l$. Clearly, this is equivalent to

$$a \sim b \text{ if and only if } a^{(q^k-1)/l} = b^{(q^k-1)/l} ,$$

and hence one ordinarily uses this value as the canonical representative of each coset. The isomorphism between $\mathbb{F}_{q^k}^*/(\mathbb{F}_{q^k}^*)^l$ and the elements of order l in $\mathbb{F}_{q^k}^*$ given by this exponentiation makes it possible to compute $f_{P,l}(Q)$ rather than $f_{P,l}(\mathcal{D})$.

The BKLS algorithm [14] takes advantage of this fact to evaluate the pairing using Miller's algorithm but without the costly denominators. Using this algorithm, we operate on tuples we call T-points such that (P,α). We then define addition of these objects

$$(P_3,\gamma) = (P_1,\alpha) + (P_2,\beta)$$

using Algorithm 4. Note that this is essentially normal point addition with some auxiliary operations to compute γ and that a method for tripling T-points follows directly from the above. In characteristic three we use tripling rather than doubling as the efficiency of cubing in $\mathbb{F}_q$ allows a particularly concise point tripling formula; in other characteristics doubling would obviously be more appropriate. Also note as mentioned above, we can ignore the division by $V(x_{P_3}, y_{P_3})$ which vastly accelerates evaluation.

Once we have the requisite tripling and addition methods as constructed for T-points, to compute the pairing we simply perform the equivalent of a point multiplication of P using these methods in place of conventional ECC point arithmetic and the curve order as the multiplier. At the end of this multiplication, we will have calculated $(\mathcal{O},\gamma)$ such that when we power γ by $(q^k-1)/l$ we have the result we want.

Algorithm 4 An algorithm for adding T-points within BKLS evaluation.

Require: T-point $\mathcal{P}_1 = (P_1, \alpha)$, point $\mathcal{P}_2 = (P_2, \beta)$
Ensure: T-point $\mathcal{P}_3 = (P_3, \gamma) = \mathcal{P}_1 + \mathcal{P}_2$
1: $P_3 \leftarrow P_1 + P_2$
2: **let** $L(X, Y) = 0$ denote the line through P_1 and P_2
3: **if** $P_3 \neq \mathcal{O}$ **then**
4: **let** $V(X, Y) = 0$ denote the line through P_3 and $\mathcal{O}$
5: **else**
6: **let** $V(X, Y) = 1$ denote the line through P_3 and $\mathcal{O}$
7: **end if**
8: $\gamma \leftarrow \frac{\alpha \cdot \beta \cdot L(x_{P_3}, y_{P_3})}{V(x_{P_3}, y_{P_3})}$
9: **return** (P_3, γ)

Characteristic 2

For curves over the field $\mathbb{F}_{2^m}$, i.e., $q = 2^m$, we use the parameterisations described in Table 4.1, using h to denote the co-factor of the curve and where

$$\begin{cases} f = +2^{(m+1)/2}\,, \\ g = -2^{(m+1)/2}\,. \end{cases}$$

Note that each curve is one one two types based on the value of m mod 8. The order of the curve is of the form

$$N = l \cdot h = 2^m + 1 \pm 2^{(m+1)/2}\ .$$

Using these parameters, the BKLS algorithm performs

$$\boxed{m\mathsf{TD} + 2\mathsf{TA}}$$

to compute the pairing with costs of TD and TA described in Table 4.2. The final powering is by the exponent

$$(2^{4m} - 1)/N = (2^{2m} - 1)(2^m + 1 \mp 2^{(m+1)/2})\ .$$

Given this special form, we can power the BKLS output to give the final pairing value with operations whose cost is again detailed in Table 4.2.

Characteristic 3

For curves over the field $\mathbb{F}_{3^m}$, i.e., $q = 3^m$, we use the parameterisations described in Table 4.1, using h to denote the co-factor of the curve and where

$$\begin{cases} f = +3^{(m+1)/2}\,, \\ g = -3^{(m+1)/2}\,. \end{cases}$$

Note that each curve is one one two types based on the value of m mod 12. The order of the curve is of the form

$$N = l \cdot h = 3^m + 1 \pm 3^{(m+1)/2}\ .$$

Table 4.2: Costs for BKLS pairing evaluation in characteristic 2.

	T-point double (TD)	T-point add (TA)	Powering
Type-A Type-D	$(4 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(4 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(7 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(3 \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(2 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(3 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(3m \cdot \mathsf{S}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_{q^k}})$
Type-B Type-C	$(4 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(4 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(7 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(3 \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(2 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(3 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(3m \cdot \mathsf{S}_{\mathbb{F}_{q^k}}) +$ $(2 \cdot \mathsf{I}_{\mathbb{F}_{q^k}})$

Using these parameters, the BKLS algorithm performs

$$\boxed{m\mathsf{TT} + 2\mathsf{TA}}$$

to compute the pairing with costs of TT and TA described in Table 4.3. The final powering is by the exponent

$$(3^{6m} - 1)/N = (3^m + 1) \cdot (3^{3m} - 1)(3^m + 1 \mp 3^{(m+1)/2}) \ .$$

Given this special form, we can power the BKLS output to give the final pairing value with operations whose cost is again detailed in Table 4.3.

Characteristic p

For curves over the field $\mathbb{F}_p$, i.e., $q = p$, we use the parameterisations described in Table 4.1, using h to denote the co-factor of the curve. The order of the subgroup is therefore of the form

$$N = (p + 1)/h \ .$$

Using these parameters, the BKLS algorithm performs roughly

$$\boxed{(N-1)\mathsf{TD} + \tfrac{|N|_2}{2}\,\mathsf{TA}}$$

Table 4.3: Costs for BKLS pairing evaluation in characteristic 3.

	T-point triple (TT)	T-point add (TA)	Powering
Type-A Type-D	$(2 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(4 \cdot \mathsf{C}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{N}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{C}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(7 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(2 \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{C}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_q}) +$ $(2 \cdot \mathsf{N}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(2 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(4 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(5m \cdot \mathsf{C}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_{q^k}})$
Type-B Type-C	$(2 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(4 \cdot \mathsf{C}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{N}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{C}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(7 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(2 \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{C}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_q}) +$ $(2 \cdot \mathsf{N}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(2 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(4 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(5m \cdot \mathsf{C}_{\mathbb{F}_{q^k}}) +$ $(2 \cdot \mathsf{I}_{\mathbb{F}_{q^k}})$

to compute the pairing with costs of TD and TA described in Table 4.4. Clearly sparse values of N are preferable in order to reduce this cost. The final powering is by the exponent

$$(p^2 - 1)/q = h \cdot (p - 1) \ .$$

Given this special form, we can power the BKLS output to give the final pairing value with operations whose cost is again detailed in Table 4.4 (on page 27).

4.1.2 Duursma-Lee and Kwon algorithms

Duursma and Lee introduced their algorithm [36] in the context of pairings on a family of supersingular hyperelliptic curves but can also be applied in the elliptic curve case. The performance of their method was improved upon by Kwon [54] who also provided similar algorithms for other base field characteristics. The Duursma-Lee and Kwon methods are detailed in Algorithms 5 and 6 respectively.

Algorithm 5 The Duursma-Lee algorithm.

Require: point $P = (x_1, y_1)$, point $Q = (x_2, y_2)$
Ensure: $f_P(\phi(Q)) \in \mathbb{F}^*_{q^6}/\mathbb{F}^*_{q^3}$

$f \leftarrow 1$
for $i = 1$ to m **do**
  $x_1 \leftarrow x_1^3$
  $y_1 \leftarrow y_1^3$
  $\mu \leftarrow x_1 + x_2 + b$
  $\lambda \leftarrow -y_1 y_2 \sigma - \mu^2$
  $g \leftarrow \lambda - \mu\rho - \rho^2$
  $f \leftarrow f \cdot g$
  $x_2 \leftarrow x_2^{1/3}$
  $y_2 \leftarrow y_2^{1/3}$
end for
return f

Algorithm 6 The Kwon algorithm.

Require: point $P = (x_1, y_1)$, point $Q = (x_2, y_2)$
Ensure: $f_P(\phi(Q)) \in \mathbb{F}^*_{q^6}/\mathbb{F}^*_{q^3}$

$f \leftarrow 1$
$x_2 \leftarrow x_2^3$
$y_2 \leftarrow y_2^3$
$d \leftarrow mb$
for $i = 1$ upto m **do**
  $x_1 \leftarrow x_1^9$
  $y_1 \leftarrow y_1^9$
  $\mu \leftarrow x_1 + x_2 + d$
  $\lambda \leftarrow y_1 y_2 \sigma - \mu^2$
  $g \leftarrow \lambda - \mu\rho - \rho^2$
  $f \leftarrow f^3 \cdot g$
  $y_2 \leftarrow -y_2$
  $d \leftarrow d - b$
end for
return f

Table 4.4: Costs for BKLS pairing evaluation in characteristic p.

	T-point double (TD)	T-point add (TA)	Powering
Type-A	$(9 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(3 \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(2 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_q}) +$ $(2 \cdot \mathsf{N}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(2 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(7 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(3 \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_q}) +$ $(2 \cdot \mathsf{N}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(3 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(2 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(2 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(3 \cdot \mathsf{S}_{\mathbb{F}_{q^k}}) +$ $(2 \cdot \mathsf{I}_{\mathbb{F}_{q^k}})$
Type-B	$(8 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(3 \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(2 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_q}) +$ $(2 \cdot \mathsf{N}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(7 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(3 \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_q}) +$ $(2 \cdot \mathsf{N}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_{q^k}}) +$ $(2 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{A}_{\mathbb{F}_q,\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_q,\mathbb{F}_{q^k}})$	$(1 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(3 \cdot \mathsf{S}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_{q^k}})$

Let $q = 3^p$ and $E(\mathbb{F}_q) : Y^2 = X^3 - X + b$, with $b = \pm 1$, and let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be points of order l. Let $\mathbb{F}_{q^3} = \mathbb{F}_q[\rho]/(\rho^3 - \rho - b)$, with $b = \pm 1$ depending on the curve equation, and let $\mathbb{F}_{q^6} = \mathbb{F}_{q^3}[\sigma]/(\sigma^2 + 1)$. Then the modified Tate pairing on E is the mapping $f_P(\phi(Q))$ where $\phi : E(\mathbb{F}_q) \rightarrow E(\mathbb{F}_{q^6})$ is the distortion map $\phi(x_2, y_2) = (\rho - x_2, \sigma y_2)$. The method for computing this is shown in Figure 5, noting that the final result is powered by $q^3 - 1$ to form a compatible result with the BLKS method.

Characteristic 3

The costs of applying the Duursma-Lee or Kwon algorithm and associated final powering are described in Table 4.5. In terms of actual cost however, several arithmetic optimisations are possible that mainly effect the multiplication in $\mathbb{F}_{q^k}$ which is the dominant cost.

Note that the result of both algorithms is raised to the power $3^{3p} - 1$. This special form clearly offers a more efficient method of powering than a standard exponentiation algorithm

Table 4.5: Costs for Duursma-Lee and Kwon pairing evaluation in characteristic 3.

	Duursma-Lee	Kwon	Powering
Type-A Type-D	$(4p \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(p \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(p \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $(2p \cdot \mathsf{C}_{\mathbb{F}_q}) +$ $((2p-2) \cdot R_{\mathbb{F}_q}) +$ $(p \cdot \mathsf{M}_{\mathbb{F}_{q^k}})$	$(5p \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(p \cdot \mathsf{M}_{\mathbb{F}_q}) +$ $(p \cdot \mathsf{S}_{\mathbb{F}_q}) +$ $((6p+2) \cdot \mathsf{C}_{\mathbb{F}_q}) +$ $(p \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(p \cdot \mathsf{C}_{\mathbb{F}_{q^k}})$	$(8 \cdot \mathsf{A}_{\mathbb{F}_q}) +$ $(2 \cdot \mathsf{N}_{\mathbb{F}_q}) +$ $(1 \cdot \mathsf{M}_{\mathbb{F}_{q^k}}) +$ $(1 \cdot \mathsf{I}_{\mathbb{F}_{q^k}})$

since over the field defined by $\theta^6 + \theta + 2$ if

$$\gamma = \sum_{i=0}^{5} a_i \, \theta^i$$

then

$$\gamma^{3^{3p}} = \sum_{i=0}^{5} b_i \, \theta^i$$

where

$$\begin{cases} b_0 = a_0 - a_1 + a_2 - a_4 - a_5 \\ b_1 = -a_1 - a_5 \\ b_2 = -a_1 - a_3 \\ b_3 = a_1 - a_2 + a_5 \\ b_4 = -a_1 + a_2 - a_3 - a_4 - a_5 \\ b_5 = a_5 \end{cases} .$$

The reader is referred to [13] for the latest techniques in pairing evaluation.

4.2 Pairing based schemes

When evaluating the cost of pairing based schemes, we use the notation detailed in Appendix B.2. We mark those operations *without* pre-computation using $-PC$ and those *with* pre-computation using $+PC$. In both cases, the operation costs are constructed as a sum of high level primitives which can be further decomposed to obtain some more meaningful value that depends on the implementation of each primitive.

- Since pairings are used in such a diverse way, benchmarking all schemes is infeasible. Therefore, we concentrate on those core schemes that are deeded useful in an operational context: we refer to [35] for a wider survey.
- Related to the above, our focus is on fairly general pairing forms, i.e. a map $\mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$ rather than the more specific forms afforded by MNT curves [61, 73] which result in

the map $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Clearly this is not ideal but is vital to limit ourselves to an achievable remit.

- We ignore the cost of computing the distortion map Φ in the BKLS method since this inconsequential in comparison to the rest of the operation.
- Although the possibility for using multi-exponentiation techniques does occur in a few situations, since it is uncommon and in order to present a simpler result, we do not consider it in the cost analysis. In a similar manner, we do not consider techniques like delayed reduction that could reduce the costs of some operations.
- We ignore low cost operations like application of XOR and symmetric encryption primitives.
- Although we include several hash functions in the analysis, the diverse nature of some protocols demands hashes with many different signatures. We only consider the most important of these, aggregating the rest in a single, generic hash class.

4.2.1 Encryption

Identity based encryption (IBE) from the Weil pairing [22]

Encrypt	$-PC$	$\mathsf{H}_{\mathbb{Z}_q} + 2 \cdot \mathsf{E}_{\mathbb{G}_1} + \mathsf{P}$
Encrypt	$+PC$	$\mathsf{H}_{\mathbb{Z}_q} + \mathsf{E}_{\mathbb{G}_1} + \mathsf{E}_{\mathbb{G}_2}$
Decrypt	$-PC$	$\mathsf{H}_{\mathbb{Z}_q} + \mathsf{P}$
Decrypt	$+PC$	N/A

If the sender often encryptions data for a given recipient, he can perform pre-computation by calculating and storing the pairing value for each such public key. Also note the trade of $\mathsf{E}_{\mathbb{G}_2}$ when pre-computation is used against $\mathsf{E}_{\mathbb{G}_1}$ when it is not since $\mathsf{E}_{\mathbb{G}_1}$ will generally be cheaper.

Efficient selective-ID secure IBE without random oracles [20]

Encrypt	$-PC$	$\mathsf{M}_{\mathbb{Z}_q} + 4 \cdot \mathsf{E}_{\mathbb{G}_1} + \mathsf{A}_{\mathbb{G}_1} + \mathsf{M}_{\mathbb{G}_2} + \mathsf{P}$
Encrypt	$+PC$	$\mathsf{M}_{\mathbb{Z}_q} + 3 \cdot \mathsf{E}_{\mathbb{G}_1} + \mathsf{A}_{\mathbb{G}_1} + \mathsf{E}_{\mathbb{G}_2} + \mathsf{M}_{\mathbb{G}_2}$
Decrypt	$-PC$	$\mathsf{E}_{\mathbb{G}_1} + \mathsf{A}_{\mathbb{G}_1} + \mathsf{M}_{\mathbb{G}_2} + \mathsf{I}_{\mathbb{G}_2} + \mathsf{P}$
Decrypt	$+PC$	N/A

Note that the pre-computation is free of conditions: the user either calculates the pairing result, stores it and uses $\mathsf{E}_{\mathbb{G}_2}$ or saves some space but pays the cost of computing the pairing and $\mathsf{E}_{\mathbb{G}_1}$.

4.2.2 Signatures

Short signatures from the Weil pairing [24]

Sign	$-PC$	$\mathsf{H}_{\mathbb{G}_1} + \mathsf{E}_{\mathbb{G}_1}$
Sign	$+PC$	N/A
Verify	$-PC$	$\mathsf{H}_{\mathbb{G}_1} + 2 \cdot \mathsf{P}$
Verify	$+PC$	N/A

Aggregate and verifiably encrypted signatures from bilinear maps [23]

Sign	$-PC$	$n \cdot \mathsf{H}_{\mathbb{G}_2} + n \cdot \mathsf{E}_{\mathbb{G}_2}$
Sign	$+PC$	N/A
Aggregate	$-PC$	$(n-1) \cdot \mathsf{M}_{\mathbb{G}_2}$
Aggregate	$+PC$	N/A
Verify	$-PC$	$n \cdot \mathsf{H}_{\mathbb{G}_2} + (n+1) \cdot \mathsf{P}$
Verify	$+PC$	N/A

Note that we are dealing with the aggregation of n BLS [24] signatures by different users into one signatures: we note the cost of the BLS signing separately in our total. Also note that we mix-notation here a little with regard to $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ to fit the scheme into our cost model.

Short signatures without random oracles [21]

Sign	$-PC$	$\mathsf{M}_{\mathbb{Z}_q} + 2 \cdot \mathsf{A}_{\mathbb{Z}_q} + \mathsf{I}_{\mathbb{Z}_q} + \mathsf{E}_{\mathbb{G}_1}$
Sign	$+PC$	N/A
Verify	$-PC$	$2 \cdot \mathsf{E}_{\mathbb{G}_1} + 2 \cdot \mathsf{A}_{\mathbb{G}_1} + 2 \cdot \mathsf{P}$
Verify	$+PC$	$2 \cdot \mathsf{E}_{\mathbb{G}_1} + 2 \cdot \mathsf{A}_{\mathbb{G}_1} + \mathsf{P}$

Note that the pre-computation here is without conditions: the verifier can either calculate and store a pairing value or save some space but calculate it online.

Efficient identity based signature schemes based on pairings [50]

Sign	$-PC$	$\mathsf{H}_{\mathbb{Z}_q} + 2 \cdot \mathsf{E}_{\mathbb{G}_1} + \mathsf{A}_{\mathbb{G}_1} + \mathsf{P}$
Sign	$+PC$	$\mathsf{H}_{\mathbb{Z}_q} + 2 \cdot \mathsf{E}_{\mathbb{G}_1} + \mathsf{A}_{\mathbb{G}_1} + \mathsf{E}_{\mathbb{G}_2}$
Verify	$-PC$	$\mathsf{H}_{\mathbb{Z}_q} + \mathsf{N}_{\mathbb{G}_1} + \mathsf{E}_{\mathbb{G}_1} + \mathsf{M}_{\mathbb{G}_2} + 2 \cdot \mathsf{P}$
Verify	$+PC$	$\mathsf{H}_{\mathbb{Z}_q} + \mathsf{E}_{\mathbb{G}_2} + \mathsf{M}_{\mathbb{G}_2} + \mathsf{P}$

If the verifier often receives signed data from a given signer, he can perform precomputation by calculating and storing a pairing value for each such public key.

4.2.3 Signcryption

Note that with all these schemes, the high cost in terms of the number of online pairing evaluations that are required makes them unattractive in the sense that performing separate encryption and signing operations is less expensive.

Identity-based signcryption [59]

Signcrypt	$-PC$	$\mathsf{H}_{\mathbb{G}_1} + 2 \cdot \mathsf{H} + 3 \cdot \mathsf{E}_{\mathbb{G}_1} + \mathsf{A}_{\mathbb{G}_1} + \mathsf{P}$
Signcrypt	$+PC$	N/A
Unsigncrypt	$-PC$	$\mathsf{H}_{\mathbb{G}_1} + 2 \cdot \mathsf{H} + \mathsf{E}_{\mathbb{G}_2} + 4 \cdot \mathsf{P}$
Unsigncrypt	$+PC$	N/A

New identity based signcryption schemes from pairings [58]

Signcrypt	$-PC$	$\mathsf{H}_{\mathbb{G}_1} + 2 \cdot \mathsf{H} + 2 \cdot \mathsf{E}_{\mathbb{G}_1} + \mathsf{N}_{\mathbb{G}_1} + \mathsf{E}_{\mathbb{G}_2} + 2 \cdot \mathsf{P}$
Signcrypt	$+PC$	N/A
Unsigncrypt	$-PC$	$\mathsf{H}_{\mathbb{G}_1} + 2 \cdot \mathsf{H} + 2 \cdot \mathsf{E}_{\mathbb{G}_2} + 4 \cdot \mathsf{P}$
Unsigncrypt	$+PC$	N/A

Chapter 5

Primitives Based on (Hyper-)Elliptic Curves

Public key cryptography based on curves over finite fields has gained a lot of interest over the years, in particular since there is no subexponential algorithm known for solving the discrete logarithm problem in the Jacobian of the curve. As a consequence the key size can be chosen much smaller than in systems based on the DL in finite fields or on RSA. For suggested numbers we refer to the ECRYPT report on key sizes and algorithms [3].

In the following we first present benchmarks for elliptic curves and then consider the generalization to hyperelliptic curves.

5.1 Elliptic curve cryptography

ECC (Elliptic Curve Cryptography) relies on a group structure induced on an elliptic curve. A set of points on an elliptic curve (with one special point added, the so-called point at infinity $\mathcal{O}$) together with a point addition as a binary operation has the structure of an abelian group.

The point or scalar multiplication is the basic operation for cryptographic protocols and it is easily performed via repeated group operations. One can visualize these operations in a hierarchical structure as shown in Fig. 5.1. Point multiplication is at the top level. At the next (lower) level are the point operations, which are closely related to the coordinates used to represent the points. The lowest level consists of finite field operations such as addition, subtraction, multiplication and inversion required to perform the group operations. Typically used fields are either of odd characteristic (i.e., $\mathbb{F}_p$) where p is a "large" prime or of characteristic 2 (i.e., $\mathbb{F}_{2^n}$). In Figure 5.1, a binary field is denoted with GF(2^n).

5.1.1 Point multiplication

Point multiplication is a special case of the general problem of exponentiation in abelian groups. As such, it benefits from all the techniques available for the *shortest addition chain* problem. Certain properties of elliptic curve can be taken into account to obtain faster algorithms. These properties are as follows:

1. Elliptic curve subtraction has the same cost as addition, so the search space for fast algorithms can be expanded to include *shortest addition-subtraction chains* and *signed representations* [18].

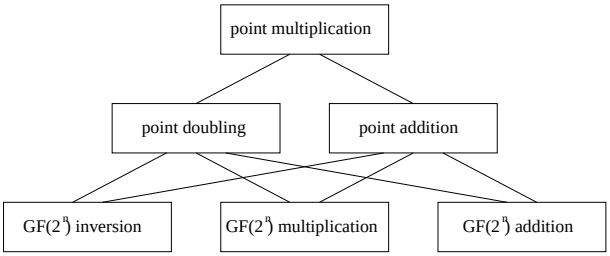

Figure 5.1: Scheme of the hierarchy for ECC operations.

2. The relative complexities of general point addition and doubling have to be considered.
3. For certain families of elliptic curves, specific shortcuts are available that can significantly reduce the computational cost.

The easiest way to calculate the point or scalar multiplication is by means of the basic double-and-add algorithm [60].

Algorithm 7 Elliptic curve point multiplication.

Require: EC point $P = (x, y)$ and scalar $k = (k_{n-1}, k_{n-2}, \ldots, k_0)_2$ with $k_{n-1} = 1$
Ensure: $Q = (x', y') = kP$

```
1: Q ← P
2: for i = n − 2 downto 0 do
3:     Q ← 2Q
4:     if k_i = 1 then
5:         Q ← Q + P
6:     end if
7: end for
8: return Q
```

The expected number of ones in the binary representation of k is $n/2$, hence the expected number of point additions and doublings, namely, A and D, respectively is:

$$\boxed{\tfrac{1}{2}\,(n-1)A + (n-1)D\,.}$$

Another option for point multiplication is the addition-subtraction method (Algorithm 8) (see e.g. [18]). It takes advantage of the fact that subtraction has the same cost as addition in elliptic curve additive group. In this case the expected computation cost is:

$$\boxed{\tfrac{1}{3}\,(n-1)A + (n-1)D\,.}$$

5.1.2 Elliptic curves over $\mathbb{F}_p$

An elliptic curve, defined over the field $\mathbb{F}_p$, is the set of solutions $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ to an equation of the form:

$$y^2 = x^3 + ax + b \tag{5.1}$$

where $a, b \in \mathbb{F}_p$ with $4a^3 + 27b^2 \neq 0 \pmod{p}$.

Algorithm 8 Left to right NAF for ECC scalar multiplication.

Require: EC point $P = (x, y)$ and
scalar k in NAF form $k = (k_{n-1}, k_{n-2}, \ldots, k_0)$ with $k_i \in \{-1, 0, 1\}$, $k_{n-1} = 1$
(NAF means that $k_i \cdot k_{i+1} = 0$)
Ensure: $R = (x', y') = kP$
1: $R \leftarrow P$
2: **for** $i = n - 2$ downto **do**
3: $R \leftarrow 2R$
4: **if** $k_i \neq 0$ **then**
5: $R \leftarrow R + k_i P$
6: **end if**
7: **end for**
8: **return** R

When E is a curve defined as above, the inverse of the point $P = (x_1, y_1)$ is $-P = (x_1, -y_1)$. The sum $P+Q$ of the points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ (assume that $P, Q \neq \mathcal{O}$, and $P \neq \pm Q$) is the point $R = (x_3, y_3)$ where:

$$\begin{aligned} \lambda &= \tfrac{y_2 - y_1}{x_2 - x_1}, \\ x_3 &= \lambda^2 - x_1 - x_2, \\ y_3 &= (x_1 - x_3)\lambda - y_1 \ . \end{aligned} \tag{5.2}$$

For $P = Q$, we get the following "doubling" formula:

$$\begin{aligned} \lambda &= \tfrac{3x_1^2 + a}{2y_1}, \\ x_3 &= \lambda^2 - 2x_1, \\ y_3 &= (x_1 - x_3)\lambda - y_1 \ . \end{aligned} \tag{5.3}$$

The point at infinity $\mathcal{O}$ plays a role analogous to that of the number 0 in ordinary addition. Thus, $P + \mathcal{O} = P$ and $P + (-P) = \mathcal{O}$ for all points P.

The total costs for general addition in affine coordinates is

$$\boxed{\mathsf{I}_{\mathbb{F}_p} + 3\mathsf{M}_{\mathbb{F}_p}}$$

and for doubling

$$\boxed{\mathsf{I}_{\mathbb{F}_p} + 4\mathsf{M}_{\mathbb{F}_p} \ .}$$

There are many types of coordinates in which an elliptic curve may be represented. In Eq. (5.1) affine coordinates are used, but so-called projective coordinates have some implementation advantages. The main conclusion is that point operations can be done in projective coordinates using only field multiplications, with no inversions required. Thus, inversions are deferred, and only one needs to be performed at the end of a point multiplication operation. A projective point (X, Y, Z) on the curve satisfies the homogeneous Weierstrass equation:

$$Y^2 Z = X^3 + aXZ^2 + bZ^3 \tag{5.4}$$

and, when $Z \neq 0$, it corresponds to the affine point $(X/Z, Y/Z)$. It appears that other projective representations result in more efficient implementations of the group operation. In

particular, in the case of the weighted projective representation (also referred to as Jacobian representation) [18] a triplet (X, Y, Z) corresponds to the affine coordinates $(X/Z^2, Y/Z^3)$ for $Z \neq 0$. In this case we have a weighted projective curve equation of the form:

$$E : Y^2 = X^3 + aXZ^4 + bZ^6 \ . \tag{5.5}$$

Weighted projective coordinates provide very fast arithmetic. In this case, conversion from projective to affine coordinates costs

$$\boxed{\mathsf{I}_{\mathbb{F}_p} + 4\mathsf{M}_{\mathbb{F}_p}}$$

while vice versa is trivial. If point addition and doubling are implemented by use of weighted projective coordinates the total costs for general addition and doubling are $16\mathsf{M}_{\mathbb{F}_p}$ and $10\mathsf{M}_{\mathbb{F}_p}$ respectively.

5.1.3 Elliptic curves over $\mathbb{F}_{2^n}$

Here we consider a finite field of characteristic 2, i.e., $\mathbb{F}_{2^n}$. A non-supersingular elliptic curve E over $\mathbb{F}_{2^n}$ is defined as the set of solutions $(x, y) \in \mathbb{F}_{2^n} \times \mathbb{F}_{2^n}$ to the equation: $y^2 + xy = x^3 + ax^2 + b$ where $a, b \in \mathbb{F}_{2^n}, b \neq 0$, together with $\mathcal{O}$.

The point addition in affine coordinates is performed according to the following formulae. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points on an elliptic curve E. Assume $P_1, P_2 \neq \mathcal{O}$ and $P_1 \neq -P_2$. The sum $P_3 = (x_3, y_3) = P_1 + P_2$ is computed as follows [18]:
If $P_1 \neq P_2$,

$$\begin{aligned} \lambda &= (y_2 + y_1) \cdot (x_2 + x_1)^{-1}, \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a, \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \ . \end{aligned}$$

If $P_1 = P_2$,

$$\begin{aligned} \lambda &= y_1/x_1 + x_1, \\ x_3 &= \lambda^2 + \lambda + a, \\ y_3 &= (x_1 + x_3)\lambda + x_3 + y_1 \ . \end{aligned}$$

In either case, the computation requires:

$$\boxed{\mathsf{I}_{\mathbb{F}_{2^n}} + 2\mathsf{M}_{\mathbb{F}_{2^n}} + \mathsf{S}_{\mathbb{F}_{2^n}} \ .}$$

Conversion from projective to affine coordinates costs, in the case of weighted projective coordinates,

$$\boxed{\mathsf{I}_{\mathbb{F}_{2^n}} + 3\mathsf{M}_{\mathbb{F}_{2^n}} + \mathsf{S}_{\mathbb{F}_{2^n}} \ .}$$

5.2 Elliptic curve based schemes

Here we give the operation costs for various elliptic curve based primitives. The operation costs are given as a sum of previously decomposed operations. The schemes include sometimes a hash function that we simply denote with H. We do not take into account any other symmetric cipher nor modular addition and reduction. The elliptic curve point multiplication is denoted with PM.

5.2.1 Signatures

The Elliptic Curve Digital Signature (ECDSA) is specified by an elliptic curve E defined over $\mathbb{F}_q$ and a publicly known point $P \in E$ of prime order m. For simplicity, we assume here that q is a prime, although the construction can easily be adapted to a prime power q as well.

A user's private key is a scalar d and the corresponding public key is $Q = dP \in E$. A signature for a message M is generated as follows:

ECDSA signature generation [19]

To sign a message M, Alice does the following:

1. Select a statistically unpredictable integer $k \in [1, m-1]$;
2. Compute $kP = (x_1, y_1)$;
3. Compute $r = x_1 \bmod m$ where x_1 is an integer between 0 and $q-1$; If $r = 0$ then go back to Step 1.
4. Compute $s = k^{-1}\left(\mathsf{H}(M) + d\,r\right) \bmod m$ where H is the Secure Hash Algorithm (SHA-1); If $s = 0$, then go to Step 1.
5. The signature for the message M is the pair (r, s).

ECDSA signature verification [19]

To verify Alice's signature (r, s) on M, Bob should do the following:

1. Obtain an authenticated copy of Alice's public key Q;
2. Verify that r and s are integers in the interval $[1, m-1]$;
3. Compute $w = s^{-1} \bmod m$ and $\mathsf{H}(M)$;
4. Compute $u_1 = \mathsf{H}(M) \cdot w \bmod m$ and $u_2 = r \cdot w \bmod m$;
5. Compute $u_1 P + u_2 Q = (x_0, y_0)$ and $v = x_0 \bmod m$;
6. Accept the signature if and only if $v = r$.

The operation costs for the ECDSA signature schemes are as follows:

Sign	$\mathsf{PM} + \mathsf{H} + 2 \cdot \mathsf{M}_{\mathbb{F}_m} + \mathsf{I}_{\mathbb{F}_m}$
Verify	$2 \cdot \mathsf{PM} + \mathsf{H} + 2 \cdot \mathsf{M}_{\mathbb{F}_m} + \mathsf{I}_{\mathbb{F}_m}$

5.2.2 Key exchange

Elliptic curve Diffie-Hellman key exchange [19]

The ECDH primitive requires only 2 point multiplications, i.e.,

$$\boxed{2\mathsf{PM}\,.}$$

5.2.3 Encryption

Elliptic curve integrated encryption scheme (ECIES) [19]

The operation costs for this primitive are:

$$\boxed{2\mathsf{PM} + 2\mathsf{H}\,.}$$

5.3 Hyperelliptic curves

Hyperelliptic curves are a generalization of elliptic curves. A genus g curve C over a field K with one point at infinity can be given by an equation of the form

$$C : y^2 + h(x)y = f(x), \quad h, f \in K[x], \deg f = 2g+1, \deg h \leq g\,, \tag{5.6}$$

where there is no point $(a, b) \in C(\overline{K})$ over the algebraic closure $\overline{K}$ satisfies both partial derivative equations simultaneously.

By this definition we include elliptic curves as curves of genus 1 and in fact the algorithmic properties we concentrate on for cryptography are very similar. However, the points of the curve do not form a group and so one uses the divisor class group Pic^0_C of degree zero as group. Like for elliptic curves the negative of an element is easily obtained such that signed expansions should be used to compute scalar mutliples.

The elements defined over K can be represented by two polynomials $\bar{D} = [u(x), v(x)]$, $u(x), v(x)$, where u is monic and $\deg v < \deg u \leq g$, the negative is given by $-\bar{D} = [u(x), -v(x)]$. This means that $2g$ field elements are used to represent a group element.

For finite fields $\mathbb{F}_q$, the theorem of Hasse-Weil states that the group order behaves as

$$|\mathrm{Pic}^0_C(\mathbb{F}_q)| = O(q^g)\ .$$

Hence, the field size can be chosen much smaller to achieve the same group size. Therefore, the amount of storage needed for a group element is equal for all choices. Clearly, for an actual implementation on a device with a given word size the changes are discrete such that one or the other choice might be more interesting depending on the level of security.

For genus $g = 2$ or 3 the security of the DLP in Pic^0_C is assumed to be equal to that of elliptic curves for the same group size, for larger genera index calculus attacks [44, 38, 80, 45] are faster than the generic attacks and so larger group sizes are needed to guarantee the same level of security.

In the following we briefly highlight the results for even and odd characteristic. Like for elliptic curves the scalar multiplication is performed as a sequence of doublings and additions and the same algorithms (Algorithm 7 and Algorithm 8) can be applied, where clearly the addition and doubling algorithms are updated.

So far no standards for hyperelliptic curve cryptography (HECC) have been proposed, but it is clear how to generalize key-exchange, ECDSA and ECIES. The operation count in the previous section lists only the number of group operations and thus remains valid for HECC. In general this field is still moving and new results appear frequently, so we expect that this part will be updated in the following version of the report. Extensive background on hyperelliptic curves can be found in the recent book [11].

5.3.1 Hyperelliptic curves over $\mathbb{F}_p$

In the case of odd characteristic $p > 5$ the equation of the curve in (5.6) can be transformed to $C : y^2 = f(x)$ by completing the square. The group operations get more complex compared to the case of elliptic curves but in return the field size is smaller such that implementations as in [12] show break even for the timings of scalar multiplication. For example, for genus 2 curves in the representation $[u(x), v(x)] = [x^2 + u_1x + u_0, v_1x + v_0]$ introduced above the most general case for the doubling operation reads as follows (the algorithm was obtained from [55] by putting $h = 0$):

Table 5.1: Doubling, $\deg u = 2$.

Input	$[u, v], u = x^2 + u_1x + u_0, v = v_1x + v_0$	
Output	$[u', v'] = 2[u, v]$	
Step	Expression	Operations
1	compute $\tilde{v} \equiv 2v \bmod u = \tilde{v}_1x + \tilde{v}_0$: $\tilde{v}_1 = 2v_1$, $\tilde{v}_0 = 2v_0$;	
2	compute resultant $r = \mathrm{res}(\tilde{v}, u)$: $w_0 = v_1^2$, $w_1 = u_1^2$, $w_2 = \tilde{v}_1^2$, $w_3 = u_1\tilde{v}_1$, $r = u_0w_2 + \tilde{v}_0(\tilde{v}_0 - w_3)$;	2S, 3M ($w_2 = 4w_0$)
3	compute almost inverse $inv' = invr$: $inv'_1 = -\tilde{v}_1$, $inv'_0 = \tilde{v}_0 - w_3$;	
4	compute $k' = (f - v^2)/u \bmod u = k'_1x + k'_0$: $w_3 = f_3 + w_1$, $w_4 = 2u_0$, $k'_1 = 2(w_1 - f_4u_1) + w_3 - w_4v_1$; $k'_0 = u_1(2w_4 - w_3 + f_4u_1) + f_2 - w_0 - 2f_4u_0$;	1M
5	compute $s' = k'inv' \bmod u$: $w_0 = k'_0inv'_0$, $w_1 = k'_1inv'_1$; $s'_1 = (inv'_0 + inv'_1)(k'_0 + k'_1) - w_0 - w_1(1 + u_1)$, $s'_0 = w_0 - u_0w_1$;	5M
6	compute $s'' = x + s_0/s_1$ and s_1: $w_1 = 1/(rs'_1)(= 1/r^2s_1)$, $w_2 = rw_1(= 1/s'_1)$, $w_3 = s'^2_1w_1(= s_1)$; $w_4 = rw_2(= 1/s_1)$, $w_5 = w_4^2$, $s''_0 = s'_0w_2$;	I, 2S, 5M
7	compute $l' = s''u = x^3 + l'_2x^2 + l'_1x + l'_0$: $l'_2 = u_1 + s''_0$, $l'_1 = u_1s''_0 + u_0$, $l'_0 = u_0s''_0$;	2M
8	compute $u' = s^2 + 2vs/u + (v^2 - f)/u^2$: $u'_0 = s''^2_0 + 2w_4v_1 + w_5(2u_1 - f_4)$; $u'_1 = 2s''_0 - w_5$;	S, 2M
9	compute $v' \equiv -(l + v) \bmod u' = v'_1x + v'_0$: $w_1 = l'_2 - u'_1$, $w_2 = u'_1w_1 + u'_0 - l'_1$, $v'_1 = w_2w_3 - v_1$; $w_2 = u'_0w_1 - l'_0$, $v'_0 = w_2w_3 - v_0$;	4M
Total		I, 5S, 22M

To avoid inversions other coordinate systems have been proposed. In *projective coordinates* $\mathcal{P}$ the pentuple $[U_1, U_0, V_1, V_0, Z]$ corresponds to the *affine* group element $[x^2 + U_1/Zx + U_0/Z, V_1/Zx + V_0/Z]$. In *new coordinates* $\mathcal{N}$ the sextuple $[U_1, U_0, V_1, V_0, Z_1, Z_2]$ corresponds to the *affine* group element $[x^2 + U_1/Z_1^2x + U_0/Z_1^2, V_1/(Z_1^3Z_2)x + V_0/(Z_1^3Z_2)]$. The following table taken from [11] lists the number of elementary field operations needed for doublings and additions; the notation $\mathcal{C}_1 + \mathcal{C}_2 = \mathcal{C}_3$ refers to an addition having inputs in $\mathcal{C}_1$ and $\mathcal{C}_2$ providing an element in system $\mathcal{C}_3$.

Table 5.2: Addition and doubling in different systems and in odd characteristic.

Doubling		Addition	
Operation	Costs	Operation	Costs
$2\mathcal{N} = \mathcal{P}$	$38\mathsf{M}_{\mathbb{F}_p} + 7\mathsf{S}_{\mathbb{F}_p}$	$\mathcal{N} + \mathcal{N} = \mathcal{P}$	$51\mathsf{M}_{\mathbb{F}_p} + 7\mathsf{S}_{\mathbb{F}_p}$
$2\mathcal{P} = \mathcal{P}$	$38\mathsf{M}_{\mathbb{F}_p} + 6\mathsf{S}_{\mathbb{F}_p}$	$\mathcal{N} + \mathcal{P} = \mathcal{P}$	$51\mathsf{M}_{\mathbb{F}_p} + 4\mathsf{S}_{\mathbb{F}_p}$
$2\mathcal{P} = \mathcal{N}$	$35\mathsf{M}_{\mathbb{F}_p} + 6\mathsf{S}_{\mathbb{F}_p}$	$\mathcal{N} + \mathcal{N} = \mathcal{N}$	$47\mathsf{M}_{\mathbb{F}_p} + 7\mathsf{S}_{\mathbb{F}_p}$
$2\mathcal{N} = \mathcal{N}$	$34\mathsf{M}_{\mathbb{F}_p} + 7\mathsf{S}_{\mathbb{F}_p}$	$\mathcal{N} + \mathcal{P} = \mathcal{N}$	$48\mathsf{M}_{\mathbb{F}_p} + 4\mathsf{S}_{\mathbb{F}_p}$
—	—	$\mathcal{P} + \mathcal{P} = \mathcal{P}$	$47\mathsf{M}_{\mathbb{F}_p} + 4\mathsf{S}_{\mathbb{F}_p}$
—	—	$\mathcal{P} + \mathcal{P} = \mathcal{N}$	$44\mathsf{M}_{\mathbb{F}_p} + 4\mathsf{S}_{\mathbb{F}_p}$
—	—	$\mathcal{A} + \mathcal{N} = \mathcal{P}$	$40\mathsf{M}_{\mathbb{F}_p} + 5\mathsf{S}_{\mathbb{F}_p}$
—	—	$\mathcal{A} + \mathcal{P} = \mathcal{P}$	$40\mathsf{M}_{\mathbb{F}_p} + 3\mathsf{S}_{\mathbb{F}_p}$
—	—	$\mathcal{A} + \mathcal{N} = \mathcal{N}$	$36\mathsf{M}_{\mathbb{F}_p} + 5\mathsf{S}_{\mathbb{F}_p}$
—	—	$\mathcal{A} + \mathcal{P} = \mathcal{N}$	$37\mathsf{M}_{\mathbb{F}_p} + 3\mathsf{S}_{\mathbb{F}_p}$
$2\mathcal{A} = \mathcal{A}$	$\mathsf{I}_{\mathbb{F}_p} + 22\mathsf{M}_{\mathbb{F}_p} + 5\mathsf{S}_{\mathbb{F}_p}$	$\mathcal{A} + \mathcal{A} = \mathcal{A}$	$\mathsf{I}_{\mathbb{F}_p} + 22\mathsf{M}_{\mathbb{F}_p} + 3\mathsf{S}_{\mathbb{F}_p}$

For curves of genus 3 explicit formulas for affine and projective coordinates exist but more progress can be expected.

5.3.2 Hyperelliptic curves over $\mathbb{F}_{2^n}$

In even characteristic the polynomial h has to be nonzero to guarantee that the curve is nonsingular. Apart from that any degree $1 \leq \deg h \leq g$ can be assumed; for $g = 2$ one should avoid constant h since these curves are supersingular and thus lead to a weaker DLP. Lower degrees of h allow faster group operation but the curves are more special; so far no attack on the special choices is known whereas the gain in speed is quite remarkable (see [56] for genus 2).

Like in the case of odd characteristic different systems of coordinates were proposed, including inversion-free systems. For a recent overview see [11].

Chapter 6

Primitives Based on Non-Hyperelliptic Curves

For many years, the only curves considered for cryptographic primitives were (hyper-)elliptic curves. Moreover, subsequently to the papers of Gaudry, Thériault and others, the only curves considered being suitable for cryptographic purposes have genus $1, 2$ or 3. However, whereas all genus 2 curves are hyperelliptic, "most" genus 3 curves are in fact not hyperelliptic. It is therefore an interesting question whether non-hyperelliptic genus 3 curves might also be cryptographically suitable.

In this chapter we show that one can perform calculations in Jacobians of genus 3 curves in an efficient way, and we outline how "cryptographically suitable" genus 3 curves can be constructed. Based on these results it might be tempting to use non-hyperelliptic genus 3 curves as cryptographic primitives. However, we also outline a new index calculus attack against the DLP in the Jacobians of these curves. A heuristic analysis of the attack indicates that in terms of running time non-hyperelliptic genus 3 curves do not provide a substantially greater security level than genus 2 curves *over the same field*. (But in the former case the key size is 50% larger.)

6.1 Fast arithmetic

Let C be a non-singular curve of genus g over a field k and D^∞ be an effective k-rational divisor of degree g. A consequence of Riemann-Roch theorem is the following representation of divisors: for a degree 0 divisor D of C defined over k (i.e., an element of $\mathrm{Div}^0_k(C)$) there exists an effective divisor E over k of degree g such that $E - D^\infty \sim D$. Generically, the divisor E is unique.

We now restrict ourselves to the case where C is a genus 3 non-hyperelliptic curve. Thanks to the canonical embedding, we may assume that C is a smooth plane quartic. We denote by x, y, z (or sometimes x_1, x_2, x_3) the chosen coordinates in $\mathbb{P}^2$.

In the following let k be the finite field $\mathbb{F}_q$ with $q = p^n$ elements. Generically there is a rational line l^∞ which crosses C in four (not necessarily distinct, but with multiplicity then) k-points $P_1^\infty, P_2^\infty, P_3^\infty, P_4^\infty$. Let D^∞ be the divisor $P_1^\infty + P_2^\infty + P_3^\infty$. For an element D in $\mathrm{Div}^0_k(C)$, let D^+ be an effective divisor (generically unique) such that $D^+ - D^\infty \sim D$. Let $D_1, D_2 \in \mathrm{Div}^0_k(C)$. Then $D_1 + D_2$ is equivalent to a divisor $D = D^+ - D^\infty$, where the points

in the support of D^+ are given by the following algorithm:

1. Take the unique cubic E which goes (with multiplicity) through the support of D_1^+, D_2^+ and $P_1^\infty, P_2^\infty, P_4^\infty$. This cubic also crosses C in the residual effective divisor D_3.

2. Take the unique conic Q which goes through the support of D_3 and P_1^∞, P_2^∞. This conic also crosses C in the residual effective divisor D^+.

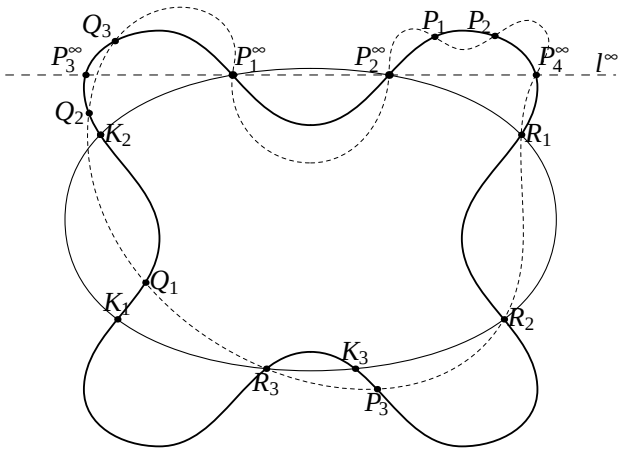

Figure 6.1: Description of the algorithm.

We first look for simple representations of the curve and its divisors: we may suppose (after a k-linear transformation) that P_1^∞ is a point at infinity (i.e., such that its z-coordinate is 0), and that l^∞ is the line $z = 0$. Let $f(x, y) = 0$ be an affine equation of C. We now come to the representation, called *Mumford representation*, of a divisor $D \in \mathrm{Div}_k^0(C)$ by a couple (u, v) of polynomials in $k[x]$. It is unique under the following generic assumptions on D, which define a *typical divisor*:

1. The three points in the support of D^+ are non-collinear. In this case D^+ is unique: in fact if $P_1 + P_2 + P_3 + (f) = Q_1 + Q_2 + Q_3$ then $f \in \mathcal{L}(P_1 + P_2 + P_3)$ and f has to be constant by the Riemann-Roch theorem.

2. There is no point at infinity in the support of D^+. Let $P_i = (x_i : y_i : 1)$ $(i = 1, 2, 3)$ be the three points in the support of D^+ and $u = \prod(x - x_i)$. Since D^+ is a rational divisor, $u \in k[x]$.

3. The $(x_i)_{i=1,2,3}$ are distinct. In this case, there exists a unique polynomial $v \in k[x]$ of degree 2 such that $y_i = v(x_i)$ for $i = 1, 2, 3$ (it is just the interpolation polynomial).

Conversely, given a couple (u, v) such that

– $u, v \in k[x]$,

– $u = \prod(x - x_i)$ is monic of degree 3 and with simple roots,

– $\deg(v) = 2$,

– $u | f(x, v(x))$,

then $P_1 + P_2 + P_3 - D^\infty$ is a rational typical divisor of C (where, for $i \in \{1, 2, 3\}$, we have $P_i = (x_i : v(x_i) : 1)$).

At last, it is obvious that the addition of two typical divisors is generically a typical divisor. As we have cryptographic applications in mind, we implement our algorithm only in that case.

In the following, we will restrict to the case that P_1^∞ is a flex. In this case we can assume (after linear transformation) that the curve is on of the form:

$$y^3 + h_1(x)y^2 + h_2(x)y = f(x)\,,$$

where $h_1, h_2, f \in \mathbb{F}_q[x]$ with $\deg(h_1) \leq 1, \deg(h_2) \leq 3$ and $\deg(f) \leq 4$. The cubic E is generically of the form

$$y^2 + s \cdot y + t\,,$$

where s and t are polynomials in x, with $\deg(s) \leq 1$ and $\deg(t) \leq 3$.
The conic Q is of the form

$$y - v\,,$$

where $v \in \mathbb{F}_q[x]$ and $\deg(v) = 2$.

The algebraic interpretation of the addition on the Jacobian [40, 41] is depicted in Algorithm 9.

To make the algorithm of the theorem more efficient, we used the following optimisations:

1. In order to reduce the number of field inversions, we used Montgomery's trick to compute simultaneous inversions. For the same reason, we computed almost inverses (using Bézout matrix), rather than inverses.

2. We used either Karatsuba or Toom-Cook (in case $p \neq 2, 3, 5$) trick to multiply two polynomials, and we computed only the coefficients we needed in the algorithm. For instance, as we only need to know the quotient of the resultant of ω and C by $u_1 u_2$, the degree ≤ 5 part of this resultant is irrelevant. Note that using Toom-Cook algorithm leads to divisions and multiplications by $2, 3$ and 5. These operations are not counted in the complexity since they are "easy".

Furthermore, if $\mathrm{Char}(k) > 3$, we can also assume that C is on the following form

$$y^3 + h_2(x)y = f_4(x)\,,$$

where h_2 is of degree 3 and f_4 has no x^3 term. In that case an addition requires $148\mathsf{M}+15\mathsf{S}+2\mathsf{I}$ and a doubling $165\mathsf{M} + 20\mathsf{S} + 2\mathsf{I}$. The following table represented explicit formulae for the addition on the Jacobian of a generic non-hyperelliptic curve of genus 3.

Algorithm 9 Algorithm for addition.

Require: $D_1 = (u_1, v_1)$ and $D_2 = (u_2, v_2)$

Ensure: $D_1 + D_2 = (u_{D_1+D_2}, v_{D_1+D_2})$

1: *Computation of the cubic E*

- *Addition*
  1. Compute the inverse t_1 of $v_1 - v_2$ modulo u_2;
  2. Compute the remainder r of $(u_1 - u_2)t_1$ by u_2;
  3. Solve the linear equations given by the following conditions

$$\begin{cases} \deg_x(-v_1(v_1+s) + u_1\delta_1) = 2 & (2 \text{ eq.}) \\ v_1 + v_2 + s \equiv r\delta_1 \quad [u_2] & (3 \text{ eq.}) \end{cases}$$

  where $s, \delta_1 \in k[x]$ with $\deg(s) = 2$ and $\deg(\delta_1) = 1$. Then

$$E = (y - v_1)(y + v_1 + s) + u_1\delta_1 \ .$$

- *Doubling*
  1. Compute $\omega_1 = (v_1^3 + v_1^2 h_1 + v_1 h_2 - f_4)/u_1$;
  2. Compute the inverse t_1 of ω_1 modulo u_1;
  3. Compute the remainder r of $(3v_1^2 + 2v_1h_1 + h_2)t_1$ by u_1;
  4. Solve the linear equations given by the following conditions

$$\begin{cases} \deg_x(-v_1(v_1+s) + u_1\delta_1) = 2 & (2 \text{ eq.}) \\ 2v_1 + s \equiv r\delta_1 \quad [u_1] & (3 \text{ eq.}) \end{cases}$$

  where $s, \delta_1 \in k[x]$ with $\deg(s) = 2$ and $\deg(\delta_1) = 1$. Then

$$E = (y - v_1)(y + v_1 + s) + u_1\delta_1 \ .$$

2: *Computation of the conic Q*

1. Compute $u' := Res^*(E, C, y)/(u_1 u_2)$;
2. Compute the inverse α_1 of $t - s^2 - h_2 + sh_1$ modulo u';
3. Compute the remainder v' of $\alpha_1(st - th_1 - f_4)$ by u'.

3: *Computation of $D_1 + D_2$*

1. $v_{D_1+D_2} := v'$;
2. $u_{D_1+D_2} := ((v^3 + v^2 h_1 + vh_2 - f_4)/(u'))^*$;
3. $D_1 + D_2 = (u_{D_1+D_2}, v_{D_1+D_2})$.

Table 6.1: Addition, $\deg u_1 = \deg u_2 = 3$.

Input	$D_1 = [u_1, v_1]$ and $D_2 = [u_2, v_2]$ $u_i = x^3 + u_{i2}x^2 + u_{i1}x + u_{i0}, v_i = v_{i2}x^2 + v_{i1}x + v_{i0}$ $C : y^3 + h(x)y - f(x) = 0$ with $h(x) := h_3x^3 + h_2x^2 + h_1x + h_0,\ f(x) := x^4 + f_2x^2 + f_1x + f_0$	
Output	$D = [u_{D_1+D_2}, v_{D_1+D_2}] = D_1 + D_2$ with $u_{D_1+D_2} = x^3 + u_2x^2 + u_1x + u_0$ and $v_{D_1+D_2} = v_2x^2 + v_1x + v_0$	
Step	Expression	Operations
1.1	compute the inverse t_1 of $v_1 - v_2$ modulo u_2 $a_1 = (v_{12} - v_{22})u_{22} - (v_{11} - v_{21}),\ a_2 = (v_{12} - v_{22})^2,\ a_3 = a_2u_{20} - a_1(v_{10} - v_{20});$ $a_4 = a_2(u_{22} + u_{21} + u_{20} + 1) - (v_{12} - v_{22} + a_1)(v_{12} + v_{11} + v_{10} - (v_{22} + v_{21} + v_{20})) - a_3;$ $a_5 = a_4(v_{12} - v_{22}),a_6 = a_4(v_{11} - v_{21}) - a_3(v_{12} - v_{22});$ $a_7 = a_4^2,res_1 = a_7(v_{10} - v_{20}) - a_6a_3,t_{10} = a_1a_6,t_{12} = (v_{12} - v_{22})a_5;$ $t_{11} = (a_1 + v_{12} - v_{22})(a_6 + a_5) - (t_{10} + t_{12}),t_{10} = t_{10} + a_7;$ $t_1 = t_{12}x^2 + t_{11}x + t_{10}$	$13\mathsf{M} + 2\mathsf{S}$
1.2	compute the remainder r of $(u_1 - u_2)t_1$ by u_2 $b_1 = (u_{12} + u_{11} + u_{10} - (u_{22} + u_{21} + u_{20}))(t_{12} + t_{11} + t_{10});$ $b_2 = (u_{12} - u_{11} + u_{10} - (u_{22} - u_{21} + u_{20}))(t_{12} - t_{11} + t_{10});$ $b_3 = (4(u_{12}-u_{22})+2(u_{11}-u_{21})+u_{10}-u_{20})(4t_{12}+2t_{11}+t_{10});$ $b_4 = (u_{12} - u_{22})t_{12},b_5 = (u_{10} - u_{20})t_{10},\ b_6 = (b_1 + b_2)/2 - (b_5 + b_4);$ $b_7 = ((b_3 + b_2 - b_1 - b_5)/2 - 2(4b_4 + b_6))/3,b_8 = b_1 - (b_5 + b_6 + b_7 + b_4);$ $b_9 = b_7 - b_4u_{22},r_2 = b_5 - b_9u_{20};$ $b_{10} = b_4 + b_7 + b_6 + b_8 + b_5 - (b_9 + b_4)(u_{22} + u_{21} + u_{20} + 1);$ $r_1 = (b_{10} - (b_4 + b_6 + b_5 - (b_7 + b_8) - (b_9 - b_4)(u_{22} - u_{21} + u_{20} - 1)))/2;$ $r_0 = b_{10} - (r_2 + r_1);$ $r = r_0x^2 + r_1x + r_2$	$9\mathsf{M}$
1.3	compute the cubic $E = y^2 + sy + t$ $c_1 = v_{12}^2,c_2 = r_0c_1,c_3 = res_1(v_{12}+v_{22})-(r_1c_1)+(c_2u_{22}),c_4 = c_3res_1;$ $c_5 = res_1r_0,c_6 = r_0c_2,,c_7 = r_2c_3 - (c_6u_{20}) - c_5(v_{10} + v_{20});$ $c_8 = (r_0 + r_1 + r_2)(c_2 + c_3) - c_6(1 + (u_{22} + u_{21} + u_{20})) - c_5(v_{22} + v_{21} + v_{20} + v_{12} + v_{11} + v_{10}) - c_7;$ $c_9 = c_4 + u_{12}c_5c_1 - v_{12}(c_8 + 2c_5v_{11}),c_{10} = c_5c_9,c_{11} = c_5^2;$	$39\mathsf{M}+3\mathsf{S}+\mathsf{I}$
*1	$c_{12} = c_9^2,\ c_{13} = c_{12} + h_3(-2c_{10} + h_3c_{11}),\ inv_1 = (c_{10}c_{13})^{-1},$ $c_{14} = c_{13}inv_1;$ $c_{15} = c_9c_{14},\ c_{16} = c_{12}inv_1c_{10};$	$(7\mathsf{M} + \mathsf{S} + \mathsf{I})$
	$s_0 = c_7c_{15},\ s_1 = c_8c_{15},\ c_{17} = c_4c_{15};$ $c_{18} = (1 + u_{12} + u_{11} + u_{10})(c_1 + c_{17}) - (v_{12} + v_{11} + v_{10})(v_{12} + v_{11} + v_{10} + s_1 + s_0);$	

	$t_3 = c_9c_{15}$, $t_0 = u_{10}c_{17} - v_{10}(v_{10} + s_0)$; $t_2 = (c_{18} + (-1 + u_{12} - u_{11} + u_{10})(-c_1 + c_{17}) - (v_{12} - v_{11} + v_{10})(v_{12} - v_{11} + v_{10} - s_1 + s_0))/2 - t_0$; $t_1 = c_{18} - (t_0 + t_2 + t_3)$, $k_1 = c_{11}c_{14}$, $c_{19} = t_0k_1$, $c_{20} = t_1k_1$, $c_{21} = t_2k_1$; $E = y^2 + (s_1x + s_0)y + t_3x^3 + t_2x^2 + t_1x + t_0$	
2.1	compute $res(E, C, y)$ and $u' := res(E, C, y)^*/(u_1u_2)$	37M + 5S
	$d_0 = c_{21}^2$, $d_1 = 3c_{21}$, $d_2 = 3(c_{20} + d_0)$, $d_3 = c_{21}(6c_{20} + d_0) + 3c_{19}$; $d_4 = s_1^2$, $d_5 = s_0^2$, $d_6 = (s_1 + s_0)^2 - (d_4 + d_5)$, $d_7 = (s_1 + s_0)(t_3 + t_2 + t_1 + t_0)$; $d_8 = (s_0 - s_1)(t_2 + t_0 - (t_3 + t_1))$, $d_9 = (2s_1 + s_0)(8t_3 + 4t_2 + 2t_1 + t_0)$; $d_{10} = s_1t_3$, $d_{11} = s_0t_0$, $d_{12} = -(d_{11} + d_{10}) + (d_7 + d_8)/2$; $d_{13} = -2d_{10} + (d_{11} - d_7 + (d_9 - d_8)/3)/2$, $d_{14} = d_7 - (d_{11} + d_{12} + d_{13} + d_{10})$; $d_{15} = s_1d_4$, $d_{16} = 3d_4s_0$, $d_{17} = 1 - 3d_{10}$, $d_{18} = d_{15} - 3d_{13}$; $d_{19} = f_2 + d_{16} + (1 - 3d_{10})f_2 - 3d_{12}$;	
*2	$d_{20} = (t_3 + t_2 + t_1 + t_0)(h_3 + h_2 + h_1 + h_0 + d_4 + d_6 + d_5 - 2(t_3 + t_2 + t_1 + t_0))$; $d_{21} = (-t_3 + t_2 - t_1 + t_0)(2(t_3 - t_2 + t_1 - t_0) - h_3 + h_2 - h_1 + h_0 + d_4 - d_6 + d_5)$; $d_{22} = (8t_3 + 4t_2 + 2t_1 + t_0)(8(-2t_3 + h_3) + 4(d_4 - 2t_2 + h_2) + 2(d_6 - 2t_1 + h_1) + d_5 - 2t_0 + h_0)$; $d_{23} = (-8t_3 + 4t_2 - 2t_1 + t_0)(-8(-2t_3 + h_3) + 4(d_4 - 2t_2 + h_2) - 2(d_6 - 2t_1 + h_1) + d_5 - 2t_0 + h_0)$; $d_{24} = (27t_3 + 9t_2 + 3t_1 + t_0)(27(-2t_3 + h_3) + 9(d_4 - 2t_2 + h_2) + 3(d_6 - 2t_1 + h_1) + d_5 - 2t_0 + h_0)$; $d_{25} = t_0(d_5 - 2t_0 + h_0)$, $d_{26} = t_3(-2t_3 + h_3)$,$d_{32} = f_2s_1$; $d_{27} = -5d_{26} + ((-(d_{20} + d_{21}) + (3d_{25} + (d_{23} + d_{22})/2)/2)/2)/3$; $d_{28} = 15d_{26} + (((5d_{25} - 7d_{20} + (-d_{24} + 7d_{22} - d_{23} - d_{21})/2)/2)/2)/3$; $d_{29} = -3d_{26} + (((d_{20} - d_{25} + (d_{21} - d_{22} + (d_{24} - d_{23})/5)/2)/2)/2)/3$; $d_{33} = (h_3 + h_2 + h_1 + h_0)(d_{26} + d_{29} + d_{27} + d_{28} + s_1 + s_0 + d_{32})$; $d_{34} = (-h_3 + h_2 - h_1 + h_0)(-d_{26} + d_{29} - d_{27} + d_{28} + s_1 - s_0 + d_{32})$; $d_{35} = (8h_3 + 4h_2 + 2h_1 + h_0)(8d_{26} + 4(d_{29} + s_1) + 2(d_{27} + s_0) + d_{28} + d_{32})$; $d_{36} = (-8h_3 + 4h_2 - 2h_1 + h_0)(-8d_{26} + 4(d_{29} + s_1) - 2(d_{27} + s_0) + d_{28} + d_{32})$; $d_{37} = (27h_3 + 9h_2 + 3h_1 + h_0)(27d_{26} + 9(d_{29} + s_1) + 3(d_{27} + s_0) + d_{28} + d_{32})$; $d_{38} = h_0(d_{28} + d_{32})$, $d_{44} = h_3d_{26}$; $d_{42} = -5d_{44} + ((-(d_{33} + d_{34}) + (3d_{38} + (d_{36} + d_{35})/2)/2)/2)/3$; $d_{41} = 15d_{44} + (((5d_{38} - 7d_{33} + (-d_{37} + 7d_{35} - d_{36} - d_{34})/2)/2)/2)/3$;	(15M)

	$d_{43} = -3d_{44} + (((d_{33} - d_{38} + (d_{34} - d_{35} + (d_{37} - d_{36})/5)/2)/2)/2)/3$; $d_{40} = (d_{33} + d_{34})/2 - (d_{38} + d_{42} + d_{44})$; $d_{39} = d_{33} - (d_{38} + d_{40} + d_{41} + d_{42} + d_{43} + d_{44})$;	
	$d_{45} = k_1^3$, $d_{46} = d_{45}(d_{19} + d_{41}) + d_3$, $d_{47} = d_{45}(d_{18} + d_{42}) + d_2$; $d_{48} = d_{45}(d_{17} + d_{43}) + d_1$;	
$*_3$	$d_{46} = d_{46}c_{16}$, $d_{47} = d_{47}c_{16}$, $d_{48} = d_{48}c_{16}$;	(3M)
	$d_{49} = u_{12} + u_{22}$, $d_{50} = u_{21} + u_{11} + u_{12}u_{22}$; $d_{51} = u_{20} + u_{10} + u_{12}u_{21} + u_{11}u_{22}$, $u'_2 = d_{48} - d_{49}$; $u'_1 = d_{47} - d_{50} - d_{49}u'_2$, $u'_0 = -d_{49}u'_1 + d_{46} - d_{51} - d_{50}(d_{48} - d_{49})$; $u' = x^3 + u'_2x^2 + u'_1x + u'_0$	
2.2	compute the inverse α_1 of $t - s^2 - h$ modulo u' $g_1 = t_3 - h_3$, $g_0 = g_1(1 + u'_2 + u'_1 + u'_0)$, $g_2 = t_0 - (d_5 + h_0 + g_1u'_0)$; $g_3 = t_3 + t_2 + t_1 + t_0 - (h_3 + h_2 + h_1 + h_0 + d_4 + d_6 + d_5 + g_0)$; $g_5 = (t_3 + t_2 + t_1 + t_0 - (h_3 + h_2 + h_1 + h_0 + d_4 + d_6 + d_5 + g_0) - (-t_3 + t_2 - t_1 + t_0 + h_3 - h_2 + h_1 - h_0 - d_4 + d_6 - d_5 - g_1(-1 + u'_2 - u'_1 + u'_0)))/2$; $g_6 = g_3 - g_5 - g_2$, $g_7 = g_6u'_2 - g_5$, $g_8 = g_6^2$, $g_{10} = g_8u'_0 - g_7g_2$; $g_{11} = g_8(1 + u'_2 + u'_1 + u'_0) - (g_6 + g_7)(g_6 + g_5 + g_2) - g_{10}$, $g_{12} = g_{11}g_6$; $g_{13} = g_{11}g_5 - g_{10}g_6$, $g_9 = g_{11}^2$, $res_2 = g_9g_2 - g_{13}g_{10}$,$\alpha_{10} = g_7g_{13}$; $\alpha_{12} = g_6g_{12}$,$\alpha_{11} = (g_6 + g_7)(g_{12} + g_{13}) - \alpha_{10} - \alpha_{12}$, $\alpha_{10} = \alpha_{10} + g_9$; $\alpha_1 = \alpha_{12}x^2 + \alpha_{11}x + \alpha_{10}$	16M + 2S
2.3	compute the remainder v of $\alpha_1(st - f_4)$ by u' $i_1 = d_{10} - 1$, $i_2 = d_{13} - (d_{10} - 1)u'_2$, $i_3 = d_{11} - f_0 - i_2u'_0$; $i_4 = d_{10} + d_{13} + d_{12} + d_{14} + d_{11} - (1 + f_2 + f_1 + f_0) - (i_2 + i_1)(u'_2 + u'_1 + u'_0 + 1)$; $i_5 = (i_4 - ((d_{10} - d_{13} + d_{12} - d_{14} + d_{11} - 1 - f_2 + f_1 - f_0) - (i_2 - i_1)(u'_2 - u'_1 + u'_0 - 1)))/2$; $i_6 = i_4 - i_3 - i_5$, $i_7 = (i_6 + i_5 + i_3)(\alpha_{12} + \alpha_{11} + \alpha_{10})$, $i_9 = i_6\alpha_{12}$, $i_{10} = i_3\alpha_{10}$; $i_8 = (i_6 - i_5 + i_3)(\alpha_{12} - \alpha_{11} + \alpha_{10})$, ,$i_{11} = (i_7 + i_8)/2 - (i_{10} + i_9)$; $i_{12} = (((4i_6 + 2i_5 + i_3)(4\alpha_{12} + 2\alpha_{11} + \alpha_{10}) - i_7 + i_8 - i_{10})/2 - 2(4i_9 + i_{11}))/3$; $i_{13} = i_7 - (i_{10} + i_{11} + i_{12} + i_9)$, $i_{14} = i_9$, $i_{15} = i_{12} - i_9u'_2$, $i_{16} = i_{10} - i_{15}u'_0$; $i_{17} = (i_9 + i_{12} + i_{11} + i_{13} + i_{10}) - (i_{15} + i_{14})(u'_2 + u'_1 + u'_0 + 1)$; $i_{18} = (i_{17} - (i_9 - i_{12} + i_{11} - i_{13} + i_{10}) + (i_{15} - i_{14})(u'_2 - u'_1 + u'_0 - 1))/2$; $i_{19} = i_{17} - i_{16} - i_{18}$, $inv_2 = (res_2i_{19})^{-1}$, $i_{20} = inv_2i_{19}$; $v_0 = i_{20}i_{16}$, $v_1 = i_{20}i_{18}$, $v_2 = i_{20}i_{19}$; $v = v_2x^2 + v_1x + v_0$	18M + I
3	compute $u := u_{D_1+D_2}$	16M + 3S

	$j_1 = inv_2 res_2^2$, $j_2 = j_1^3$, $j_3 = j_1 v_1$, $j_4 = j_3^2$, $j_5 = j_1 v_0$, $j_6 = j_3(j_4 + 6j_5)$;	
$*_4$	$j_7 = (v_2 + v_1 + v_0)(h_3 + h_2 + h_1)$, $j_8 = (v_2 - v_1 + v_0)(h_3 - h_2 + h_1)$, $j_9 = v_2 h_3$; $j_{10} = v_0 h_1$,$j_{11} = (j_7 + j_8)/2 - (j_{10} + j_9)$, $j_{12} = 3j_3 + j_2 j_9$, $j_{14} = j_6 + j_2 j_{11}$; $j_{13} = 3(j_5 + j_4) - j_2 + j_2((((4v_2 + 2v_1 + v_0)(4h_3 + 2h_2 + h_1) - j_7 + j_8 - j_{10})/2 - 2(4j_9 + j_{11}))/3)$;	(8M)
	$u_2 = j_{12} - u_2'$, $u_1 = j_{13} - u_1' - u_2' u_2$, $u_0 = -u_2' u_1 + j_{14} - u_0' - u_1'(j_{12} - u_2')$; $u = x^3 + u_2 x^2 + u_1 x + u_0$	
Total		148M, 15S, 2I

Table 6.2: Doubling, $\deg u_1 = 3$.

Input	$D_1 = [u_1, v_1]$ $u_1 = x^3 + u_{12}x^2 + u_{11}x + u_{10}$, $v_1 = v_{12}x^2 + v_{11}x + v_{10}$ $C : y^3 + h(x)y - f(x) = 0$ with $h(x) := h_3 x^3 + h_2 x^2 + h_1 x + h_0$, $f(x) := x^4 + f_2 x^2 + f_1 x + f_0$	
Output	$D = [u_{2D_1}, v_{2D_1}] = 2D_1$ with $u_{2D_1} = x^3 + u_2 x^2 + u_1 x + u_0$ and $v_{2D_1} = v_2 x^2 + v_1 x + v_0$	
Step	Expression	Operations
1.1	compute w_1 such that $u_1 w_1 = v_1^3 + h(x)v_1 - f(x)$	12M + 5S
	$l_1 = (v_{12} + v_{11} + v_{10})^2$, $l_2 = (v_{12} - v_{11} + v_{10})^2$, $l_3 = v_{12}^2$, $l_4 = v_{10}^2$; $l_5 = (l_1 + l_2)/2 - (l_4 + l_3)$; $l_6 = (((4v_{12} + 2v_{11} + v_{10})^2 - l_1 + l_2 - l_4)/2 - 2(4l_3 + l_5))/3$; $l_7 = l_1 - (l_4 + l_5 + l_6 + l_3)$,$l_8 = (v_{12} + v_{11} + v_{10})(l_3 + l_6 + l_5 + l_7 + h_3 + h_2 + h_1)$; $l_9 = (v_{12} - v_{11} + v_{10})(-l_3 + l_6 + h_3 - (l_5 + h_2) + l_7 + h_1)$; $l_{10} = (4v_{12} + 2v_{11} + v_{10})(8l_3 + 4(l_6 + h_3) + 2(l_5 + h_2) + l_7 + h_1)$; $l_{11} = (4v_{12} - 2v_{11} + v_{10})(-8l_3 + 4(l_6 + h_3) - 2(l_5 + h_2) + l_7 + h_1)$; $l_{12} = v_{10}(l_7 + h_1)$, $l_{13} = v_{12} l_3$,$l_{14} = -5l_{13} + ((l_9 - l_8 + (l_{10} - l_{11})/2)/2)/3$; $l_{15} = ((-(l_8 + l_9) + (3l_{12} + (l_{10} + l_{11})/2)/2)/2)/3$; $l_{16} = (l_8 + l_9)/2 - (l_{12} + l_{15})$,$l_{14} = l_{14} - 1$, $w_{13} = l_{13}$,$w_{12} = l_{15} - w_{13} u_{12}$; $w_{11} = l_{14} - w_{13} u_{11} - w_{12} u_{12}$,$w_{10} = l_{16} - w_{13} u_{10} - w_{12} u_{11} - w_{11} u_{12}$; $w_1 = w_{13} x^3 + w_{12} x^2 + w_{11} x + w_{10}$	
1.2	compute the inverse t_1 of w_1 modulo u_1	16M + 2S
	$a_1 = w_{13}$, $a_2 = w_{10} - a_1 u_{10}$; $a_3 = w_{13} + w_{12} + w_{11} + w_{10} - a_1(1 + u_{12} + u_{11} + u_{10})$; $a_4 = (a_3 - (-w_{13} + w_{12} - w_{11} + w_{10} - a_1(-1 + u_{12} - u_{11} + u_{10})))/2$; $a_5 = a_3 - a_4 - a_2$, $a_6 = a_5 u_{12} - a_4$, $a_7 = a_5^2$, $a_8 = a_7 u_{10} - a_6 a_2$;	

	$a_9 = a_7(1 + u_{12} + u_{11} + u_{10}) - (a_5 + a_6)(a_5 + a_4 + a_2) - a_8$, $a_{10} = a_9 a_5$; $a_{11} = a_9 a_4 - a_8 a_5$, $a_7 = a_9^2$, $res_1 = a_7 a_2 - a_{11} a_8$, $t_{10} = a_6 a_{11}$; $t_{12} = a_5 a_{10}$, $t_{11} = (a_5 + a_6)(a_{10} + a_{11}) - t_{10} - t_{12}$, $t_{10} = t_{10} + a_7$; $t_1 = t_{12} x^2 + t_{11} x + t_{10}$	
1.3	compute the remainder r of $(3v_1^2 + h)t_1$ by u_1	13M
	$b_1 = 3l_6 + h_3 - 3l_3 u_{12}$, $b_2 = 3l_4 + h_0 - b_1 u_{10}$; $b_3 = (3l_3 + 3l_6 + h_3 + 3l_5 + h_2 + 3l_7 + h_1 + 3l_4 + h_0) - (b_1 + 3l_3)(u_{12} + u_{11} + u_{10} + 1)$; $b_4 = (b_3 - ((3l_3 - (3l_6 + h_3) + 3l_5 + h_2 - (3l_7 + h_1) + 3l_4 + h_0) - (b_1 - 3l_3)(u_{12} - u_{11} + u_{10} - 1)))/2$; $b_5 = b_3 - b_2 - b_4$, $b_6 = (b_5 + b_4 + b_2)(t_{12} + t_{11} + t_{10})$; $b_7 = (b_5 - b_4 + b_2)(t_{12} - t_{11} + t_{10})$, $b_8 = b_5 t_{12}$, $b_9 = b_2 t_{10}$; $b_{10} = (b_6 + b_7)/2 - (b_9 + b_8)$; $b_{11} = (((4b_5 + 2b_4 + b_2)(4t_{12} + 2t_{11} + t_{10}) - b_6 + b_7 - b_9)/2 - 2(4b_8 + b_{10}))/3$; $b_{12} = b_6 - (b_9 + b_{10} + b_{11} + b_8)$, $b_{13} = b_{11} - b_8 u_{12}$, $r_2 = b_9 - b_{13} u_{10}$; $b_{14} = (b_8 + b_{11} + b_{10} + b_{12} + b_9) - (b_{13} + b_8)(u_{12} + u_{11} + u_{10} + 1)$; $r_1 = (b_{14} - (b_8 + b_{10} + b_9) + (b_{11} + b_{12}) + (b_{13} - b_8)(u_{12} - u_{11} + u_{10} - 1))/2$; $r_0 = b_{14} - (r_2 + r_1)$; $r = r_0 x^2 + r_1 x + r_2$	
1.4	compute the cubic $E = y^2 + sy + t$	39M + 2S + I
	$c_1 = l_3$, $c_2 = r_0 c_1$, $c_3 = 2res_1 v_{12} - (r_1 c_1 - c_2 u_{12})$, $c_4 = c_3 res_1$, $c_5 = res_1 r_0$; $c_6 = r_0 c_2$, $c_7 = r_2 c_3 - c_6 u_{10} - 2c_5 v_{10}$; $c_8 = (r_0 + r_1 + r_2)(c_2 + c_3) - c_6(1 + u_{12} + u_{11} + u_{10}) - 2c_5(v_{12} + v_{11} + v_{10}) - c_7$; $c_9 = c_4 + u_{12} c_5 c_1 - v_{12}(c_8 + 2c_5 v_{11})$, $c_{10} = c_5 c_9$, $c_{11} = c_5^2$;	
*1	$c_{12} = c_9^2$, $c_{13} = c_{12} + h_3(-2c_{10} + h_3 c_{11})$, $inv_1 = (c_{10} c_{13})^{-1}$, $c_{14} = c_{13} inv_1$; $c_{15} = c_9 c_{14}$, $c_{16} = c_{12} inv_1 c_{10}$;	(7M + S + I)
	$s_0 = c_7 c_{15}$, $s_1 = c_8 c_{15}$, $c_{17} = c_4 c_{15}$; $c_{18} = (1 + u_{12} + u_{11} + u_{10})(c_1 + c_{17}) - (v_{12} + v_{11} + v_{10})(v_{12} + v_{11} + v_{10} + s_1 + s_0)$; $t_3 = c_9 c_{15}$, $t_0 = u_{10} c_{17} - v_{10}(v_{10} + s_0)$; $t_2 = (c_{18} + (-1 + u_{12} - u_{11} + u_{10})(-c_1 + c_{17}) - (v_{12} - v_{11} + v_{10})(v_{12} - v_{11} + v_{10} - s_1 + s_0))/2 - t_0$; $t_1 = c_{18} - (t_0 + t_2 + t_3)$, $k_1 = c_{11} c_{14}$, $c_{19} = t_0 k_1$, $c_{20} = t_1 k_1$, $c_{21} = t_2 k_1$; $E = y^2 + (s_1 x + s_0) y + t_3 x^3 + t_2 x^2 + t_1 x + t_0$	
2.1	compute $res(E, C, y)$ and $u' := res(E, C, y)^*/(u_1 u_2)$	35M + 6S
	$d_0 = c_{21}^2$, $d_1 = 3c_{21}$, $d_2 = 3(c_{20} + d_0)$, $d_3 = c_{21}(6c_{20} + d_0) + 3c_{19}$;	

	$d_4 = s_1^2$, $d_5 = s_0^2$, $d_6 = (s_1 + s_0)^2 - (d_4 + d_5)$, $d_7 = (s_1 + s_0)(t_3 + t_2 + t_1 + t_0)$; $d_8 = (s_0 - s_1)(t_2 + t_0 - (t_3 + t_1))$, $d_9 = (2s_1 + s_0)(8t_3 + 4t_2 + 2t_1 + t_0)$; $d_{10} = s_1 t_3$, $d_{11} = s_0 t_0$, $d_{12} = -(d_{11} + d_{10}) + (d_7 + d_8)/2$; $d_{13} = -2d_{10} + (d_{11} - d_7 + (d_9 - d_8)/3)/2$, $d_{14} = d_7 - (d_{11} + d_{12} + d_{13} + d_{10})$; $d_{15} = s_1 d_4$, $d_{16} = 3d_4 s_0$, $d_{17} = 1 - 3d_{10}$, $d_{18} = d_{15} - 3d_{13}$; $d_{19} = f_2 + d_{16} + (1 - 3d_{10})f_2 - 3d_{12}$;	
$*_2$	$d_{20} = (t_3 + t_2 + t_1 + t_0)(h_3 + h_2 + h_1 + h_0 + d_4 + d_6 + d_5 - 2(t_3 + t_2 + t_1 + t_0))$; $d_{21} = (-t_3 + t_2 - t_1 + t_0)(2(t_3 - t_2 + t_1 - t_0) - h_3 + h_2 - h_1 + h_0 + d_4 - d_6 + d_5)$; $d_{22} = (8t_3 + 4t_2 + 2t_1 + t_0)(8(-2t_3 + h_3) + 4(d_4 - 2t_2 + h_2) + 2(d_6 - 2t_1 + h_1) + d_5 - 2t_0 + h_0)$; $d_{23} = (-8t_3 + 4t_2 - 2t_1 + t_0)(-8(-2t_3 + h_3) + 4(d_4 - 2t_2 + h_2) - 2(d_6 - 2t_1 + h_1) + d_5 - 2t_0 + h_0)$; $d_{24} = (27t_3 + 9t_2 + 3t_1 + t_0)(27(-2t_3 + h_3) + 9(d_4 - 2t_2 + h_2) + 3(d_6 - 2t_1 + h_1) + d_5 - 2t_0 + h_0)$; $d_{25} = t_0(d_5 - 2t_0 + h_0)$, $d_{26} = t_3(-2t_3 + h_3)$,$d_{32} = f_2 s_1$; $d_{27} = -5d_{26} + ((-(d_{20} + d_{21}) + (3d_{25} + (d_{23} + d_{22})/2)/2)/2)/3$; $d_{28} = 15d_{26} + (((5d_{25} - 7d_{20} + (-d_{24} + 7d_{22} - d_{23} - d_{21})/2)/2)/2)/3$; $d_{29} = -3d_{26} + (((d_{20} - d_{25} + (d_{21} - d_{22} + (d_{24} - d_{23})/5)/2)/2)/2)/3$; $d_{33} = (h_3 + h_2 + h_1 + h_0)(d_{26} + d_{29} + d_{27} + d_{28} + s_1 + s_0 + d_{32})$; $d_{34} = (-h_3 + h_2 - h_1 + h_0)(-d_{26} + d_{29} - d_{27} + d_{28} + s_1 - s_0 + d_{32})$; $d_{35} = (8h_3 + 4h_2 + 2h_1 + h_0)(8d_{26} + 4(d_{29} + s_1) + 2(d_{27} + s_0) + d_{28} + d_{32})$; $d_{36} = (-8h_3 + 4h_2 - 2h_1 + h_0)(-8d_{26} + 4(d_{29} + s_1) - 2(d_{27} + s_0) + d_{28} + d_{32})$; $d_{37} = (27h_3 + 9h_2 + 3h_1 + h_0)(27d_{26} + 9(d_{29} + s_1) + 3(d_{27} + s_0) + d_{28} + d_{32})$; $d_{38} = h_0(d_{28} + d_{32})$, $d_{44} = h_3 d_{26}$; $d_{42} = -5d_{44} + ((-(d_{33} + d_{34}) + (3d_{38} + (d_{36} + d_{35})/2)/2)/2)/3$; $d_{41} = 15d_{44} + (((5d_{38} - 7d_{33} + (-d_{37} + 7d_{35} - d_{36} - d_{34})/2)/2)/2)/3$; $d_{43} = -3d_{44} + (((d_{33} - d_{38} + (d_{34} - d_{35} + (d_{37} - d_{36})/5)/2)/2)/2)/3$; $d_{40} = (d_{33} + d_{34})/2 - (d_{38} + d_{42} + d_{44})$; $d_{39} = d_{33} - (d_{38} + d_{40} + d_{41} + d_{42} + d_{43} + d_{44})$;	(15M)
	$d_{45} = k_1^3$, $d_{46} = d_{45}(d_{19} + d_{41}) + d_3$, $d_{47} = d_{45}(d_{18} + d_{42}) + d_2$; $d_{48} = d_{45}(d_{17} + d_{43}) + d_1$;	
$*_3$	$d_{46} = d_{46} c_{16}$, $d_{47} = d_{47} c_{16}$, $d_{48} = d_{48} c_{16}$;	(3M)
	$d_{49} = 2u_{12}$, $d_{50} = 2u_{11} + u_{12}^2$, $d_{51} = 2u_{10} + 2u_{12}u_{11}$, $u_2' = d_{48} - d_{49}$; $u_1' = d_{47} - d_{50} - d_{49}u_2'$, $u_0' = -d_{49}u_1' + d_{46} - d_{51} - d_{50}(d_{48} - d_{49})$;	

	$u' = x^3 + u'_2x^2 + u'_1x + u'_0$	
2.2	compute the inverse α_1 of $t - s^2 - h$ modulo u' $g_1 = t_3 - h_3, g_0 = g_1(1+u'_2+u'_1+u'_0)$, $g_2 = t_0-(d_5+h_0+g_1u'_0)$; $g_3 = t_3+t_2+t_1+t_0-(h_3+h_2+h_1+h_0+d_4+d_6+d_5+g_0)$; $g_5 = (t_3+t_2+t_1+t_0-(h_3+h_2+h_1+h_0+d_4+d_6+d_5+$ $g_0)-(-t_3+t_2-t_1+t_0+h_3-h_2+h_1-h_0-d_4+d_6-d_5-$ $g_1(-1+u'_2-u'_1+u'_0)))/2$; $g_6 = g_3-g_5-g_2$, $g_7 = g_6u'_2-g_5$, $g_8 = g_6^2$, $g_{10} = g_8u'_0-g_7g_2$; $g_{11} = g_8(1+u'_2+u'_1+u'_0)-(g_6+g_7)(g_6+g_5+g_2)-g_{10}$, $g_{12} = g_{11}g_6$; $g_{13} = g_{11}g_5-g_{10}g_6$, $g_9 = g_{11}^2$, $res_2 = g_9g_2-g_{13}g_{10}$, $\alpha_{10} =$ g_7g_{13}; $\alpha_{12} = g_6g_{12}$, $\alpha_{11} = (g_6+g_7)(g_{12}+g_{13})-\alpha_{10}-\alpha_{12}$, $\alpha_{10} =$ $\alpha_{10}+g_9$; $\alpha_1 = \alpha_{12}x^2+\alpha_{11}x+\alpha_{10}$	16M + 2S
2.3	compute the remainder v of $\alpha_1(st-f_4)$ by u' $i_1 = d_{10}-1$, $i_2 = d_{13}-(d_{10}-1)u'_2$, $i_3 = d_{11}-f_0-i_2u'_0$; $i_4 = d_{10}+d_{13}+d_{12}+d_{14}+d_{11}-(1+f_2+f_1+f_0)-(i_2+$ $i_1)(u'_2+u'_1+u'_0+1)$; $i_5 = (i_4-((d_{10}-d_{13}+d_{12}-d_{14}+d_{11}-1-f_2+f_1-f_0)-$ $(i_2-i_1)(u'_2-u'_1+u'_0-1)))/2$; $i_6 = i_4-i_3-i_5$, $i_7 = (i_6+i_5+i_3)(\alpha_{12}+\alpha_{11}+\alpha_{10})$, $i_9 = i_6\alpha_{12}$, $i_{10} = i_3\alpha_{10}$; $i_8 = (i_6-i_5+i_3)(\alpha_{12}-\alpha_{11}+\alpha_{10})$, ,$i_{11} = (i_7+i_8)/2-(i_{10}+i_9)$; $i_{12} = (((4i_6+2i_5+i_3)(4\alpha_{12}+2\alpha_{11}+\alpha_{10})-i_7+i_8-i_{10})/2-$ $2(4i_9+i_{11}))/3$; $i_{13} = i_7-(i_{10}+i_{11}+i_{12}+i_9)$, $i_{14} = i_9$, $i_{15} = i_{12}-i_9u'_2$, $i_{16} = i_{10}-i_{15}u'_0$; $i_{17} = (i_9+i_{12}+i_{11}+i_{13}+i_{10})-(i_{15}+i_{14})(u'_2+u'_1+u'_0+1)$; $i_{18} = (i_{17}-(i_9-i_{12}+i_{11}-i_{13}+i_{10})+(i_{15}-i_{14})(u'_2-u'_1+$ $u'_0-1))/2$; $i_{19} = i_{17}-i_{16}-i_{18}$, $inv_2 = (res_2i_{19})^{-1}$, $i_{20} = inv_2i_{19}$; $v_0 = i_{20}i_{16}$, $v_1 = i_{20}i_{18}$, $v_2 = i_{20}i_{19}$; $v = v_2x^2+v_1x+v_0$	18M + I
3	compute $u := u_{2D_1}$ $j_1 = inv_2res_2^2$, $j_2 = j_1^3$, $j_3 = j_1v_1$, $j_4 = j_3^2$, $j_5 = j_1v_0$, $j_6 = j_3(j_4+6j_5)$;	16M + 3S
$*_4$	$j_7 = (v_2+v_1+v_0)(h_3+h_2+h_1)$, $j_8 = (v_2-v_1+v_0)(h_3-$ $h_2+h_1)$, $j_9 = v_2h_3$; $j_{10} = v_0h_1$, $j_{11} = (j_7+j_8)/2-(j_{10}+j_9)$, $j_{12} = 3j_3+j_2j_9$, $j_{14} = j_6+j_2j_{11}$; $j_{13} = 3(j_5+j_4)-j_2+j_2((((4v_2+2v_1+v_0)(4h_3+2h_2+h_1)-$ $j_7+j_8-j_{10})/2-2(4j_9+j_{11}))/3)$;	(8M)
	$u_2 = j_{12}-u'_2$, $u_1 = j_{13}-u'_1-u'_2u_2$, $u_0 = -u'_2u_1+j_{14}-u'_0-$ $u'_1(j_{12}-u'_2)$; $u = x^3+u_2x^2+u_1x+u_0$	
Total		165M, 20S, 2I

Table 6.3: If $h_3 = 0$ then replace $*_1, *_2, *_3$ and $*_4$ by

$*_1$	$inv_1 = c_{10}^{-1}$,$c_{14} = inv_1$,$c_{15} = inv_1 c_9$,$c_{16} = 1$;	(M + I)
$*_2$	$d_{27} = (t_3 + t_2 + t_1)(h_1 + h_2 + d_4 + d_6 - 2(t_3 + t_2 + t_1))$; $d_{28} = (t_3 - t_2 + t_1)(h_1 - h_2 + d_6 - d_4 - 2(t_3 - t_2 + t_1))$; $d_{29} = (4t_3 + 2t_2 + t_1)(-8t_3 + 2(d_4 - 2t_2 + h_2) + d_6 - 2t_1 + h_1)$; $d_{30} = -2t_3^2$,$d_{31} = t_1(d_6 - 2t_1 + h_1)$,$d_{32} = (d_{27} + d_{28})/2 - (d_{31} + d_{30})$; $d_{33} = ((d_{29} - d_{27} + d_{28} - d_{31})/2 - 2(4d_{30} + d_{32}))/3$; $d_{35} = (h_2 + h_1 + h_0)(d_{30} + d_{33} + d_{32} + s_0 + s_1)$; $d_{36} = (h_2 - h_1 + h_0)(d_{30} - d_{33} + d_{32} + s_0 - s_1)$; $d_{37} = (4h_2 + 2h_1 + h_0)(4d_{30} + 2(d_{33} + s_1) + d_{32} + s_0)$; $d_{43} = h_2 d_{30}$,$d_{39} = h_0(d_{32} + s_0)$, $d_{41} = (d_{35} + d_{36})/2 - (d_{39} + d_{43})$; $d_{42} = ((d_{37} - d_{35} + d_{36} - d_{39})/2 - 2(4d_{43} + d_{41}))/3$; $d_{40} = d_{35} - (d_{39} + d_{41} + d_{42} + d_{43})$,$d_{44} = 0$;	(9M + S)
$*_3$		
$*_4$	$j_{11} = v_2 h_2$,$j_{12} = 3j_3$,$j_{13} = 3(j_5 + j_4) - j_2 + j_2 j_{11}$; $j_{14} = j_6 + j_2((v_2 + v_1)(h_2 + h_1) - (v_1 h_1 + j_{11}))$;	(5M)

Table 6.4: If $h_3, h_2 = 0$ then replace $*_1, *_2, *_3$ and $*_4$ by

$*_1$	$inv_1 = c_{10}^{-1}$,$c_{14} = inv_1$,$c_{15} = inv_1 c_9$,$c_{16} = 1$;	(M + I)
$*_2$	$d_{37} = -2t_3^2$,$d_{35} = t_2(d_4 - 2t_2)$,$d_{38} = 0$, $d_{39} = 0$,$d_{43} = 0$,$d_{44} = 0$; $d_{36} = (t_3 + t_2)(d_4 - 2(t_3 + t_2)) - (d_{35} + d_{37})$, $d_{42} = h_1 d_{37}$,$d_{40} = h_0(d_{36} + s_1)$; $d_{41} = (h_1 + h_0)(d_{37} + d_{36} + s_1) - (d_{40} + d_{42})$;	(5M + S)
$*_3$		
$*_4$	$j_{12} = 3j_3$,$j_{13} = 3(j_5 + j_4) - j_2$,$j_{14} = j_6 + j_2(h_1 v_2)$;	(2M)

Table 6.5: If $h_3, h_2, h_1 = 0$ then replace $*_1, *_2, *_3$ and $*_4$ by

$*_1$	$inv_1 = c_{10}^{-1}$,$c_{14} = inv_1$,$c_{15} = inv_1 c_9$,$c_{16} = 1$;	(M + I)
$*_2$	$d_{41} = -2h_0 t_3^2$,$d_{38} = 0$,$d_{39} = 0$,$d_{40} = 0$,$d_{42} = 0$, $d_{43} = 0$,$d_{44} = 0$;	(M + S)
$*_3$		
$*_4$	$j_{12} = 3j_3$,$j_{13} = 3(j_5 + j_4) - j_2$,$j_{14} = j_6$;	

Table 6.6: If $h_3, h_2, h_1, h_0 = 0$ then replace $*_1, *_2, *_3$ and $*_4$ by

$*_1$	$inv_1 = c_{10}^{-1}$,$c_{14} = inv_1$,$c_{15} = inv_1 c_9$,$c_{16} = 1$;	(M + I)
$*_2$	$d_{38} = 0$,$d_{39} = 0$,$d_{40} = 0$,$d_{41} = 0$,$d_{42} = 0$, $d_{43} = 0$,$d_{44} = 0$;	
$*_3$		
$*_4$	$j_{12} = 3j_3$,$j_{13} = 3(j_5 + j_4) - j_2$,$j_{14} = j_6$;	

6.2 Construction of secure non-hyperelliptic curves of genus 3

The current methods for point counting on curves over finite fields of small characteristic rely essentially on a p-adic approach (those based on cohomology (Kedlaya), those based on deformation theory (Lauder) and those based on the canonical lift (Satoh). The *AGM method* belongs to the last category. In this section, we give an outline of this method.

Let $k := \mathbb{F}_q$ with $q = 2^N$ and let $\mathbb{Q}_q$ be the unramified extension of degree N of $\mathbb{Q}_2$ with ring of integers $\mathbb{Z}_q$, π a uniformizer and v a valuation such that $v(\pi) = 1$.

Let in the following $\tilde{C}/k$ be an ordinary genus 3 non-hyperelliptic curve. In this case the

curve C has exactly 7 bitangents (defined over over $\bar{k}$) and is isomorphic over $\bar{k}$ to a curve with the following model $(*)$

$$(ax^2 + by^2 + cz^2 + dxy + exz + fyz)^2 = xyz(x + y + z)$$

with

$$abc(a + b + d)(a + c + e)(b + c + f)(a + b + c + d + e + f + 1) \neq 0 \ .$$

Furthermore we assume that its Jacobian is absolutely simple and has its 2^g points of order 2 defined over k (i.e. all its bitangents are defined over k). We give here the general AGM-algorithm [67] with specific details.

1. First lift the curve $\tilde{C}$ to a curve $C/\mathbb{Q}_q$ with Riemann model

   $$\sqrt{x_1 u_1} + \sqrt{x_2 u_2} + \sqrt{x_3 u_3} = 0$$

   which has all its bitangents defined over $\mathbb{Q}_q$. From this model we can explicitely compute the equation of all the other bitangents.

2. Using the Weber formulae

   $$\left(\frac{\vartheta[\chi](0)}{\vartheta(0)}\right)^4 = \frac{[\beta_i, \beta_j, \beta_{ij}][\beta_{ik}, \beta_{jk}, \beta_{ij}][\beta_j, \beta_{jk}, \beta_k][\beta_i, \beta_{ik}, \beta_k]}{[\beta_j, \beta_{jk}, \beta_{ij}][\beta_i, \beta_{ik}, \beta_{ij}][\beta_i, \beta_j, \beta_k][\beta_{ik}, \beta_{jk}, \beta_k]} ,$$

   where the β_i and β_{jk} are the bitangents of a certain Aronhold system and $[\beta_{l_1}, \beta_{l_2}, \beta_{l_3}] = \det(\beta_{l_1}, \beta_{l_2}, \beta_{l_3})$ and where χ is the even theta characteristic $\chi = [\epsilon_i] + [\epsilon_j] + [\epsilon_k]$, we compute (in terms of coefficients of C) the 2^3 constants coefficients which are square roots of the algebraic expressions of the corresponding complex quotients (for a certain Riemann matrix Ω):

   $$\left(\frac{\vartheta \begin{bmatrix} \epsilon \\ \epsilon' \end{bmatrix} (0, \Omega)}{\vartheta \begin{bmatrix} 0 \\ 0 \end{bmatrix} (0, \Omega)}\right)^4_{\epsilon, \epsilon' \in (\mathbb{Z}/2\mathbb{Z})^3}$$

   We denote these quotients by $\left(\vartheta_e^{(0)}\right)_{e \in (\mathbb{Z}/2\mathbb{Z})^3}$.

3. We build a sequence by using formulas which are a generalization of the Arithmetic Geometric Mean. In 2-adic context, they are

   $$\vartheta_e^{(i+1)} = \frac{1}{2^3} \sum_{f \in (\mathbb{Z}/2\mathbb{Z})^3} \vartheta_e^{(i)} \sqrt{\frac{\vartheta_{e+f}^{(i)}}{\vartheta_e^{(i)}}}$$

   The square root of an element $x \in 1 + 8\mathbb{Z}_q$ is chosen to be congruent to 1 modulo 4.

4. Let π_i be the roots of the Frobenius polynomial that are 2-adic units (there are exactly 3 such roots since $\tilde{C}$ is ordinary). The fundamental proposition of the AGM theory is that the quotient $\vartheta_e^{(n+N)}/\vartheta_e^{(n)}$ converges linearly to $\alpha = \pm\pi_1 \cdots \pi_3$.

5. We compute the minimal polynomial of $\beta = \alpha + 2^{3N}/\alpha$ using LLL-algorithm. We need to know β with a precision of $10N$ bits, then it is easy to find the characteristic polynomial of the Frobenius up to a sign problem which we solve with fast addition in the Jacobian.

The complexity in time of the algorithm is $O(N^{2\mu} \log N)$ where $O(N^\mu)$ is the number of bit operations for the multiplication of two N bit length integers. With Karatsuba $\mu = \log_2(3)$, with FFT multiplication $\mu = 1 + \epsilon$. The complexity in space is $O(N^2)$.

6.3 Solving the DLP in Jacobians of non-hyperelliptic genus 3 curves

In this section we outline a recent index calculus algorithm which is particularly well suited for non-hyperelliptic genus 3 curves. A heuristic analysis of the algorithm indicates that "most" instances of the DLP in Jacobian groups of non-hyperelliptic genus 3 curves over $\mathbb{F}_q$ can be solved in a time of $\tilde{O}(q)$. (Here, the $\tilde{O}$-notation indicates that we disregard logarithmic factors.)

6.3.1 General description of index calculus with double large prime variation

The algorithm can be viewed as an adaption of (a simplified version of) the *algorithm by Adleman-DeMarrais-Huang* to the low genus situation. A crucial ingredient of the algorithm is a *double large prime variation*. As in the recent algorithm by Gaudry, Thériault and Thomé [45], the set of large primes is $C(\mathbb{F}_q) - \mathcal{F}$, where $\mathcal{F}$ is the factor base.

Let us before we come to the concrete algorithm describe what "index calculus with double large prime variation" is.

Recall that in the context of discrete logarithm problems in an abelian group, by an "index calculus algorithm", one means a method which is based on (a variant of) the following general description:

Let G be an explicitly given finite abelian group, and let $a, b \in G$ with $a \in \langle b \rangle$. Because we are interested in cryptographic applications, we assume the order $\ell := \#\langle b \rangle$ is known. For simplicity we assume that ℓ is prime. (One can easily reduce to this case by the Chinese Remainder Theorem.) Then one proceeds as follows:

First, one fixes a factor base $\mathcal{F} = \{F_1, F_2, \ldots\} \subset G$. We assume that we have a "sufficiently efficient" method to find relations $\sum_j r_j F_j = \alpha a + \beta b$.

One stores these relations as rows of a "relation matrix" R over $\mathbb{Z}/\ell\mathbb{Z}$ which is usually a sparse matrix, and one also stores vectors of the α's and β's.

Finally, one calculates a random element $v \in \ker(R^t)$. Then one has the relation $0 = \sum_{i,j} v_i r_{ij} F_j = \sum_{i,j} v_i(\alpha_i a + \beta_i b)$. If now $\sum_i v_i \beta_i$ is in $(\mathbb{Z}/\ell\mathbb{Z})^*$, we have

$$b = -\frac{\sum_{i,j} v_i \alpha_i}{\sum_i v_i \beta_i} \cdot a \in G \ .$$

In "index calculus with (single) large prime variation" one proceeds in a similar way. The essential differences are as follows. Additionally to the factor base $\mathcal{F}$, one fixes a set of *large*

primes $\mathcal{L} \subset G - \mathcal{F}$. Then one searches for relations of the form

$$\sum_j r_j F_j + P = \alpha a + \beta b \,,$$

where $P \in \mathcal{L} \cup \mathcal{F}$, and again one stores these relations. If two relations with the same "large prime" P are found, the difference is calculated, and in this way one obtains a relation involving only elements from $\mathcal{F}$.

In "index calculus with (double) large prime variation", one again fixes a set of large primes $\mathcal{L} \subset G - \mathcal{F}$, and one searches for relations of the form

$$\sum_j r_j F_j + P + Q = \alpha a + \beta b \,,$$

where $P, Q \in \mathcal{L} \cup \mathcal{F}$. Such relations are stored in a "graph of large prime variation". This is a graph on the set $\mathcal{L} \cup \{*\}$, where $*$ is an additional point corresponding to all elements of $\mathcal{F}$. A relation as above leads to a directed edge with label $((r_j)_j; \alpha, \beta)$ between P and Q (where P or Q is substituted by $*$ if it is in $\mathcal{F}$). Now cycles in the graph of large prime variation lead to relations over the factor base (no cycles are in fact stored in the graph).

One particular problem with this approach is that it is difficult to control the size of the cycles generated, and it is therefore difficult to control the sparsity of the matrix generated. This problem can be circumvented as follows: One only considers relations as above where P or Q are in the connected component of $*$, and one disregards all other relations (and again one does not include relations in the graph which would lead to cycles). The generated graph is thus in fact a tree, we call it the *tree of large prime variation*. Like this, one can guarantee that the cycle lengths are polynomial in $\log(q)$.

In [45] it is shown (under some heuristic assumptions) that with index calculus with double large prime variation one can solve the DLP in Jacobian groups of genus 3 curves in a time of

$$\tilde{O}(q^{4/3}) \ .$$

Note that this running time is already better than the running time of $\Theta(q^{3/2})$ one obtains with "generic methods" like the ρ-method (provided that the group order is "nearly prime").

(The result in [45] is only stated for hyperelliptic curves but it also applies to non-hyperelliptic curves.)

6.3.2 The algorithm

We now describe the new index calculus algorithm with double large prime variation given in [33]. More precisely, the algorithm we describe here can be viewed as the algorithm in Section 3 of [33] with the additional ingredient of a double large prime variation. We note that in Section 4 of [33] a closely related algorithm is described. The algorithm here can be viewed as a "practical variant" of the algorithm in Section 4 of [33].

As before, let $C/\mathbb{F}_q$ be a genus 3 curve, explicitly given as a curve in $\mathbb{P}^2/\mathbb{F}_q$ by a homogeneous polynomial $F(x, y, z)$ of degree 4. The group order of $\mathrm{Jac}(\mathcal{C})(\mathbb{F}_q)$ is assumed to be known (as is always the case in cryptographic applications).

Let $a, b \in \mathrm{Jac}(C)(\mathbb{F}_q)$, and let $P_0 \in C(\mathbb{F}_q)$ be a fixed point. Let D_∞ be the divisor "at infinity", i.e. the divisor obtained via the equation $Z = 0$ on C (D_∞ an effective divisor of

degree 4). We assume that $a \in \langle b \rangle$. For $D \in \mathrm{Div}(C)$, let $[D]$ denote the corresponding divisor class.

The first step of the algorithm is to find two expressions

$$[P_1] + [P_2] + [P_3] - 3[P_0] = \alpha a \qquad [P_1'] + [P_2'] + [P_3'] - 3[P_0] = \beta b\,,$$

where $P_i, Q_i \in C(\mathbb{F}_q)$ and $\alpha, \beta \neq 0 \in (\mathbb{Z}/\ell\mathbb{Z})^*$.

Only after these relations are found, a factor base $\mathcal{F} = \{F_1, F_2, \ldots\}$ with slightly more than $(2 \cdot q \cdot \log(q))^{\frac{1}{2}}$ elements is fixed.

When the factor base is chosen, it is made sure that it contains the 7 $\mathbb{F}_q$-rational points $P_1, P_2, P_3, P_1', P_2', P_3', P_0$ of C. The two relations already obtained are stored as the first two rows in the relation matrix.

The two relations derived at the beginning are the *only* relations involving a non-trivial "right-hand side" (linear combinations of a and b). All other relations are of the form

$$[F_i] + [F_j] + [P] + [Q] = [D_\infty]\,,$$

where $P, Q \in C(\mathbb{F}_q)$. These relations are found in the following way:

Lines through two elements of the factor base are constructed, and the intersection divisor of these lines with the curve is considered. For this, tuples (i, j) with $i < j$ are fixed and the line $L(x, y, z) = 0$ passing through F_i and F_j is considered. Then the intersection divisor of the line with the curve has the form

$$F_i + F_j + D_{i,j}\,,$$

where $D_{i,j}$ is some divisor of degree 2. Now the divisor $F_i + F_j + D_{i,j} - D_\infty$ is the principal divisor corresponding to the element $L(\frac{x}{z}, \frac{y}{z}, 1)$ in the function field $\mathbb{F}_q(C)$. By definition of the Jacobian group, we have the relation

$$[F_i] + [F_j] + [D_{i,j}] - [D_\infty] = 0\ .$$

If now $D_{i,j}$ splits as $D_{i,j} = P + Q$ (with $P, Q \in \mathrm{Jac}(C)(\mathbb{F}_q)$), we obtain a relation

$$[F_i] + [F_j] + [P] + [Q] = [D_\infty]\ .$$

This is a relation involving (at most) two large primes and otherwise elements from the factor base (the divisor D_∞ can be disregarded). With these relations, we build the tree of large prime variation as described in the previous subsection. (That is, we insert an edge provided that P or Q lie in the connected component of $*$ and this does lead to a cycle. If it lead to a cycle, we store the relation obtained from the cycle as a row in the relation matrix R.)

The relation search is terminated if the number of rows of R exceeds number of the elements in the factor base slightly. (If one has considered all lines through pairs of elements of the factor base before this happens, the algorithm fails and should be rerun with another (possibly larger) factor base.)

The *linear algebra step* is as usual: The relations are stored as rows of a sparse matrix R. Then with an algorithm from sparse linear algebra a random element $v \in \ker(R^t)$ is calculated. We then have a relation of the form

$$v_1 \alpha a + v_2 \beta b = 0\ .$$

The main assumption is now that v_2 lies in $(\mathbb{Z}/\ell\mathbb{Z})^*$. If this assumption is satisfied, we have

$$b = -\frac{v_1\alpha}{v_2\beta} \cdot a \,,$$

and this solves the DLP. If the assumption is not satisfied, more relations should be obtained by considering more lines through elements of the factor base. If all lines are exhausted, the calculation fails, and the algorithm should be rerun with another (possibly larger) factor base.

6.3.3 Analysis of the algorithm

We now sketch a heuristic analysis of the algorithm (for details we refer to [33]).

Let us first recall that $C(\mathbb{F}_q) \sim q$ for $q \longrightarrow \infty$ by the bounds of Hasse-Weil.

Under the assumption that a and b generate $\mathrm{Jac}(C)(\mathbb{F}_q)$, the average number of tries one needs to obtain any of the first two relations is asymptotically equal to $3! = 6$. This means that these relations can be obtained in a time of $\tilde{O}(1)$. Heuristically, this also holds if a and b generate a proper subgroup of $\mathrm{Jac}(C)(\mathbb{F}_q)$.

Each line can be constructed, the corresponding divisor can be tested for complete splitting, and the splitting can be calculated in a time of $\tilde{O}(1)$.

Let us now choose the factor base $\mathcal{F}$ with slightly more than $(2 \cdot q \cdot \log(q))^{\frac{1}{2}}$ elements.

The probability that a divisor of degree 2 splits completely is asymptotically equal to $\frac{1}{2}$ (for $q \longrightarrow \infty$). This motivates the *assumption*: The probability that a line passing through 2 points of $\mathcal{F}$ defines a completely split divisor is roughly $\frac{1}{2}$.

Under some assumptions on the "equidistibutivness" of the relations generated, one can analyse the generated tree of large prime variation via differential equations (see [33]). The outcome of this analysis is that the factor base should be large enough such that a relation matrix R with more rows than columns can be generated as described above. This indicates that the DLP can be solved by calculating a random vector in $\ker(R^t)$.

Under the same heuristic assumptions, it is also possible to analyse the cycle lengths, and just as in [45], they are polynomial in $\log(q)$.

As the factor base has $\tilde{O}(q^{\frac{1}{2}})$ elements, all lines passing through two points of the factor base can be constructed in a time of $\tilde{O}(q)$, and the linear algebra part also has a running time of $\tilde{O}(q)$.

All in all, the heuristic analysis indicates that "most" instances of the DLP in Jacobian groups of non-hyperelliptic genus 3 curves can be solved in a time of $\tilde{O}(q)$.

6.4 Conclusion

In this chapter, it was shown that there exists an efficient arithmetic for non-hyperelliptic genus 3 curves, and that non-hyperelliptic genus 3 curves can be efficiently constructed with the AGM-method. This suggests that the DLP in Jacobian groups of such curves might provide an interesting alternative to the DLP in elliptic curves and genus 2 curves as a public key crypto primitive.

However, with a recent index calculus algorithm, the DLP in Jacobian groups of non-hyperelliptic curves 3 curves can (under some heuristic assumptions) be solved in a time of $\tilde{O}(q)$. This should be compared with the running time of $\Theta(q^{3/2})$ group operations for "generic methods" like the ρ-method (provided that the group order is nearly prime). The

hidden constant (and logarithmic factor) in the $\tilde{O}$-notation for the index calculus method are quite small, and the result is clearly relevant in practice. On the other hand, the relation search phase of the algorithm requires a storage of roughly q elements in $\mathbb{F}_q$, and as usual, the linear algebra part is difficult to parallelise with current computer technology. It it therefore not straightforward to compare the two algorithms for "cryptographically suitable" key sizes from a practical point of view.

Despite the fact that such a comparison is difficult, the new index calculus algorithm clearly shows that non-hyperelliptic genus 3 curves have a weakness in comparison to elliptic curves and genus 2 curves. As we are not aware of any advantages of the usage of genus 3 curves, we recommend against the usage of non-hyperelliptic genus 3 curves in cryptographic applications.

Chapter 7

Primitives Based on Multivariate Polynomials

7.1 Introduction

The area of Multivariate-Polynomial based Cryptosystems (MPC) is rich in constructions and each construction has many variants and parameters. Many MPCs have been broken in the past 10 years, and we will concentrate on a few schemes the security of which has been studied, and for which a correct choice of parameters should remain unbroken.

In general the MPCs have at least one of the two following drawbacks:

D1. The public key may be quite large, which determines the speed of signature verification or encryption with the public key;

D2. The computation of a signature or private-key decryption may be very slow.

In spite of these two drawbacks, in some applications, such as for example very short signatures, or signatures that can be computed on 8-bit smart card with no arithmetic co-processor, no better cryptographic scheme (that would not have at least one of these drawbacks) is known.

In this chapter we are not be as much concerned with D1 (that is never really considered to be critical in the MPC area). With regard to speed (D2), the state of the art on MPC can be summarized as follows:

Speed and signatures For digital signatures a very fast MPC has been proposed: Sflash. It has been studied and recommended by Nessie [7, 6].

Slow schemes for PK-encryption It seems that to propose a fast and secure multivariate public key encryption scheme is harder than for digital signatures. Many schemes have been proposed and broken, and it seems that we don't know a good candidate for a fast and secure public key encryption scheme. We will give an evaluation for a version Sflash adapted for encryption (that will not be as fast as in signature).

Slow schemes for short signatures For this specific application (short means less than 160 bits), no very fast scheme is known, and all of them require many seconds on a powerful PC to compute a single signature. We will give a performance evaluation for Quartz.

Multivariate PK-authentication schemes These schemes work differently than schemes designed for encryption and signature (there is no trapdoor construction). We will give a performance evaluation for the "IP with two secrets scheme" proposed by Patarin and for the MinRank scheme proposed by Courtois. They may appear quite fast compared to some other multivariate schemes (Quartz), but remain slower than the classical authentication schemes such as Fiat-Shamir, GPS or GQ2. Nevertheless they have the advantage of being based on NP-hard problems.

7.2 Multivariate-polynomial based schemes

7.2.1 How to evaluate the performance of a MPC

Signature verification / Encryption All MPCs are very similar in this respect. The public key is in general a set of m quadratic (sometimes other low-degree) polynomials with n variables over GF(q). The size of the public key is then:

$$\boxed{S = m\,\tfrac{n^2}{2} \cdot \log_2(q)\ \text{bits}\,.}$$

and the time to evaluate these polynomials (to verify a signature or a to encrypt a message) is $\mathcal{O}(S)$.

Computation of a signature / decryption Here the method varies greatly from one system to another, depending on the type of algebraic trapdoor used in the cryptosystem.

7.2.2 Sflash signature scheme

We summarize here the main characteristics of Sflash^{v2} following the state of the art, that can be found in [9, 10, 30, 82, 31].

- Length of the signature: 259 bits.
- Length of the public key: 15.4 Kbytes.
- Length of the secret key: the secret key (2.45 Kbytes) can be generated from a small seed of (at least) 128 bits.
- Time to sign a message: less than 1 ms on a PC (pessimistic estimation). Only 59 ms on a Infineon SLE66 component without cryptoprocessor with 10 MHz clock (see the optimized implementation described in [10]).
- Time to verify a signature: a few ms on a PC (approximately $37 \times 37 \times 26$ multiplications and additions in GF(128)).
- Time to generate a pair of public key/secret key: less than 1s (see [81] for details).
- Best known attack: generic algebraic attacks (solving the equations ignoring the trapdoor) requires more than 2^{90} TDES computations (the complexity of the attacks in [30] was overestimated, see [82] for a more precise evaluation). The resistance of Sflash against algebraic and Gröbner bases attacks that exploit the trapdoor, such as described in [28, 39] is also believed to be very good; see [31, Section 7].

7.2.3 Sflash (adapted to encryption)

The signature scheme Sflash^{v2} can be adapted for public key encryption. Sflash^{v2} is a trapdoor function $\mathrm{GF}(2^7)^{37} \to \mathrm{GF}(2^7)^{26}$ [9]. Instead of removing 11 public equations, only 3 are removeed. We get so a trapdoor function $\mathrm{GF}(2^7)^{37} \to \mathrm{GF}(2^7)^{34}$. Since this the trapdoor function is not bijective, the encryption proceeds as follows. To encrypt a message, we choose an AES session key on 128 bits, and 131 bits of redundancy computed with MD5, and get a message on 259 bits encoded as $37 \cdot 7$ bits. When we apply the public key equations we get $34 \cdot 7 = 238$ bits, which still has about 110 bits of redundancy. The owner of the private key searches through all possible (on average) $2^{3.3}$ keys for one that has the right redundancy.

We summarize here the main characteristics of Sflash adapted for encryption:

- Length of the public key: 20 Kbytes.
- Length of the secret key: the secret key (1.2 Kbytes) can be generated from a small seed of (at least) 128 bits.
- Time to encrypt a message: less than 1 ms on a PC.
- Time to decrypt a message: les than 0.3 seconds on a PC. (About 2^{21} trial decryptions, each requires mainly to compute and check the MD5 redundancy.)
- Best known attack: more than 2^{90} TDES computations (the complexity of the attacks described in [30] should be evaluated according to [82]). The resistance of Sflash against algebraic and Gröbner bases attacks that exploit the trapdoor, such as described in [28, 39] should be very good, as it appears in [31, Section 7].

7.2.4 Quartz signature scheme

We summarize here the main characteristics of Quartz following the state of the art [65, 39, 31].

- Length of the signature: 128 bits.
- Length of the public key: 71 Kbytes.
- Length of the secret key: the secret key (3 Kbytes) is generated from a small seed of at least 128 bits.
- Time to generate the public key: about 1 second on a P4 @ 3GHz.
- Time to sign a message: less than 10 seconds on average on a P4 @ 3GHz.
- Time to verify a signature: less than 1 ms.
- Best known attack: would be more than 2^{62} computations according to Faugère and Joux [39], however according to [31] the complexity of this attack may have been overestimated.

7.2.5 IP authentication scheme with two secrets

We summarize here the main characteristics of IP with two secrets following the state of the art [64]. We summarize here the characteristics of the second scheme proposed in Section 2.2. of the Habilitation thesis of Patarin [64], with 20 equations of degree 2 over GF(256), $\lambda = 2$, and 20 iterations of the scheme:

- Length of the public key: 4.1 Kbytes.
- Computation time for the prover: estimated to be about 0.6 seconds in a low-end 8-bit smart card without any arithmetic or cryptographic co-processor, and about 0.06 mili-seconds on a modern CPU such as Pentium 4.
- RAM required for the prover: 50 bytes.
- Computation time for the verifier: estimated to be about 0.1 seconds in a low-end 8-bit smart card without any arithmetic or cryptographic co-processor, and about 0.01 mili-seconds on a modern CPU such as Pentium 4.
- RAM required for the verifier: 50 bytes.
- Size of the algorithm code: 400 bytes.
- The size of the answer: 64 Kbits (can be reduced to 34 K).
- Impersonation probability: 2^{-20}.
- Best known attack: 2^{80} computations.

7.2.6 MinRank authentication scheme

We summarize here the main characteristics of the MinRank authentication scheme following [27]. We consider a version (A) of MinRank with matrices 6×6 over GF(65521), with 35 rounds, as studied in [27, Sections 4.3 and 8.2].

- Length of the public key: 735 bits.
- Length of the secret key: 160 bits.
- Computation time for the prover: essentially the time to apply $35 \cdot 3$ times a cryptographic hash function such as SHA-1.
- Computation time for the verifier: very similar.
- Size of the algorithm code: at most few hundreds bytes.
- The total size of the answer: 35 Kbits (can be reduced to 14 K).
- Impersonation probability: 2^{-20}.
- Best known attack: 2^{106} computations.

Bibliography

[1] ANSI X9.31. Digital signatures using reversible public key cryptography for the financial services industry (rDSA), September 9, 1998.

[2] ANSI X9.42. Agreement of symmetric algorithms using discrete logarithm cryptography, 2000.

[3] ECRYPT. Ecrypt yearly report on algorithms and keysizes. Technical report, 2005.

[4] FIPS PUB 186-2. Digital signature standard (DSS). National Institute for Standards and Technology, 2000.

[5] Intel Corporation. Using streaming SIMD extensions (SSE2) to perform big multiplications. Application note AP-941, order number 248606-001, July 2000.

[6] NESSIE Consortium. Portfolio of recommended cryptographic primitives, February 2003. Avalaible at URL `https://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/decision-final.pdf`.

[7] NESSIE Security Report. Revised final version 2.0, February 2003. Avalaible at URL `http://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/D20-v2.pdf`.

[8] Shamus Software Ltd. *M.I.R.A.C.L. Users Manual*, November 2004. Available for download at `ftp://ftp.computing.dcu.ie/pub/crypto/manual.doc`.

[9] The specification of Sflash^{v2}. Available at URL `http://www.cryptosystem.net/sflash/`.

[10] M.-L. Akkar, N. Courtois, L. Goubin, and R. Duteuil. A fast and secure implementation of Slash. In Y. Desmedt, editor, *Public Key Cryptography – PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 267–278. Springer-Verlag, 2003.

[11] R. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *The Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2005.

[12] R.M. Avanzi. Aspects of hyperelliptic curves over large prime fields in software implementations. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[13] P.S.L.M. Barreto, S. Galbraith, C. O hEigeartaigh, and M. Scott. Efficient pairing computation on supersingular abelian varieties. Report 2004/375, Cryptology ePrint Archive, 2004.

[14] P.S.L.M. Barreto, H. Kim, B. Lynn, and M. Scott. Efficient algorithms for pairing-based cryptosystems. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 354–368. Springer-Verlag, 2002.

[15] P. Barrett. Implementing the Rivest, Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In A.M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer-Verlag, 1987.

[16] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In A. De Santis, editor, *Advances in Cryptology – EUROCRYPT '94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer-Verlag, 1995.

[17] M. Bellare and P. Rogaway. The exact security of digital signatures - How to sign with RSA and Rabin. In U. Maurer, editor, *Advances in Cryptology – EUROCRYPT '96*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer-Verlag, 1996.

[18] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[19] I. Blake, G. Seroussi, and N. Smart. *Advances in Elliptic Curve Cryptography*. Cambridge University Press, 2005.

[20] D. Boneh and X. Boyen. Efficient selective-ID secure identity based encryption without random oracles. In C. Cachin and J. Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 223–238. Springer-Verlag, 2004.

[21] D. Boneh and X. Boyen. Short signatures without random oracles. In C. Cachin and J. Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 56–73. Springer-Verlag, 2004.

[22] D. Boneh and M. Franklin. Identity based encryption from the Weil pairing. In J. Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer-Verlag, 2001.

[23] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In E. Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 416–432. Springer-Verlag, 2003.

[24] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 1976 of *Lecture Notes in Computer Science*, pages 514–532. Springer-Verlag, 2001.

[25] A. Bosselaers, R. Govaerts, and J. Vandewalle. Comparison of three modular reduction functions. In D.R. Douglas, editor, *Advances in Cryptology – CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 175–186. Springer-Verlag, 1994.

[26] P.G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.

[27] N. Courtois. Efficient zero-knowledge authentication based on a linear algebra problem MinRank. In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 1976 of *Lecture Notes in Computer Science*, pages 402–421. Springer-Verlag, 2001.

[28] N. Courtois. The security of Hidden Field Equations (HFE). In D. Naccache, editor, *Topics in Cryptology – CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 266–281. Springer-Verlag, 2001.

[29] N. Courtois. Generic attacks and the security of Quartz. In Y. Desmedt, editor, *Public Key Cryptography – PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 351–364. Springer-Verlag, 2003. A preliminary version was presented at the second Nessie workshop, Royal Holloway, University of London, September 2001.

[30] N. Courtois. Algebraic attacks over GF(2^k): Application to HFE Challenge 2 and Sflash-v2. In F. Bao, R.H. Deng, and J. Zhou, editors, *Public Key Cryptography – PKC 2004*, volume 2947 of *Lecture Notes in Computer Science*, pages 201–217. Springer-Verlag, 2004.

[31] N. Courtois. Short signatures, provable security, generic attacks and computational security of multivariate polynomial schemes such as HFE, Quartz and Sflash. Report 2004/143, Cryptology ePrint Archive, 2004. A very much extended version of [29] with a lot of added material.

[32] R. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent #5159632, 1992.

[33] C. Diem. Index calculus in class groups of plane curves of small degree. Report 2005/119, Cryptology ePrint Archive, 2005.

[34] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[35] R. Dutta, R. Barua, and P. Sarkar. Pairing-based cryptographic protocols: A survey. Report 2004/064, Cryptology ePrint Archive, 2004.

[36] I.M. Duursma and H-S. Lee. Tate pairing implementation for hyperelliptic curves $y^2 = x^p - x + d$. In C.-S. Laih, editor, *Advances in Cryptology – ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 111–123. Springer-Verlag, 2003.

[37] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[38] A. Enge and P. Gaudry. A general framework for subexponential discrete logarithm algorithms. *Acta Arith.*, 102(1):83–103, 2002.

[39] J.-C. Faugère and A. Joux. Algebraic cryptanalysis of Hidden Field Equation (HFE) cryptosystems using Gröbner bases. In D. Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 44–60. Springer-Verlag, 2003.

[40] S. Flon and R. Oyono. Fast arithmetic on Jacobians of Picard curves. In F. Bao, R.H. Deng, and J. Zhou, editors, *Public Key Cryptography – PKC 2004*, volume 2947 of *Lecture Notes in Computer Science*, pages 55–68. Springer-Verlag, 2004.

[41] S. Flon, R. Oyono, and C. Ritzenthaler. Fast addition on non-hyperelliptic genus 3 curves. Report 2004/118, Cryptology ePrint Archive, 2004.

[42] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. *Journal of Cryptology*, 17(2):81–104, 2004.

[43] S.D. Galbraith, K. Harrison, and D. Soldera. Implementing the Tate pairing. In C. Fieker and D.R. Kohel, editors, *Algorithmic Number Theory Symposium (ANTS V)*, volume 2369 of *Lecture Notes in Computer Science*, pages 324–337. Springer-Verlag, 2002.

[44] P. Gaudry. An algorithm for solving the discrete log problem on hyperelliptic curves. In *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 19–34. Springer-Verlag, 2000.

[45] P. Gaudry, N. Thériault, and E. Thomé. A double large prime variation for small genus hyperelliptic index calculus. Report 2004/153, Cryptology ePrint Archive, 2004.

[46] D.M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27:129–146, 1998.

[47] R. Granger, D. Page, and M. Stam. A comparison of CEILIDH and XTR. In D. Buell, editor, *Algorithmic Number Theory Symposium (ANTS VI)*, volume 3076 of *Lecture Notes in Computer Science*, pages 235–249. Springer-Verlag, 2004.

[48] T. Granlund. The GNU Multiple Precision arithmetic library, 2004. Version 4.1.4.

[49] L. Guillou and J.-J. Quisquater. A 'paradoxical' identity-based signature scheme resulting from zero-knowledge. In S. Goldwasser, editor, *Advances in Cryptology – CRYPTO '88*, volume 403 of *Lecture Notes in Computer Science*, pages 213–231. Springer-Verlag, 1990.

[50] F. Hess. Efficient identity based signature schemes based on pairings. In K. Nyberg and H.M. Heys, editors, *Selected Areas in Cryptography (SAC 2002)*, volume 2595 of *Lecture Notes in Computer Science*, pages 310–324. Springer-Verlag, 2002.

[51] A.A. Karatsuba and Yu. Ofman. Multiplication of multiplace numbers on automata. *Dokl. Acad. Nauk SSSR*, 145(2):293–294, 1962.

[52] D.E. Knuth. *The Art of Computer Programming. Vol. 2, Seminumerical Algorithms.* Addison-Wesley, 2nd edition, 1998.

[53] Ç.K. Koç, T. Acar, and B.S. Kaliski, Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[54] D. Kwon. Efficient Tate pairing computation for supersingular elliptic curves over binary fields. Report 2004/303, Cryptology ePrint Archive, 2004.

[55] T. Lange. Formulae for arithmetic on genus 2 hyperelliptic curves. *Appl. Algebra Engrg. Comm. Comput.*, 15(5):295–328, 2005.

[56] T. Lange and M. Stevens. Efficient doubling for genus two curves over binary fields. In H. Handschuh and M.A. Hasan, editors, *Selected Areas in Cryptography – SAC 2004*, volume 3357 of *Lecture Notes in Computer Science*, pages 170–181. Springer-Verlag, 2005.

[57] A.K. Lenstra and E.R. Verheul. The XTR public key system. In M. Bellare, editor, *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2000.

[58] B. Libert and J.-J. Quisquater. New identity based signcryption schemes from pairings. Report 2003/023, Cryptology ePrint Archive, 2003.

[59] J. Malone-Lee. Identity-based signcryption. Report 2002/098, Cryptology ePrint Archive, 2002.

[60] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[61] A. Miyaji, M. Nakabayashi, and S. Takano. New explicit conditions of elliptic curve traces for FR-reduction. *IEICE Transactions on Fundamentals*, E84-A(5):1234–1243, 2001.

[62] S. Mohan and B. Adiga. Fast algorithms for implementing RSA public key cryptosystems. *Electronics Letters*, 21(17):761, 1985.

[63] P.L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[64] J. Patarin. *La Cryptographie Multivariable*. Habilitation thesis (HDR), Paris 7, 1999.

[65] J. Patatin, N. Courtois, and L. Goubin. QUARTZ, 128-bit long digital signatures. In D. Naccache, editor, *Topics in Cryptology – CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 282–297. Springer-Verlag, 2001.

[66] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18:905–907, 1982.

[67] C. Ritzenthaler. Point counting on genus 3 non hyperelliptic curves. In *Algorithmic Number Theory Symposium – ANTS-VI*, volume 3076 of *Lecture Notes in Computer Science*, pages 379–394. Springer-Verlag, 2004.

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

[69] O.G. Rizzo. On the complexity of the 2^k-ary and of the sliding window algorithms for fast exponentiation. Technical report, Dipartimento di Matematica, Universitá di Milano, Milan, March 2004.

[70] K. Rubin and A. Silverberg. Torus-based cryptography. In D. Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 349–365. Springer-Verlag, 2003.

[71] M. Scott. Fast machine code for modular multiplication, January 1995. Manuscript, available for download at `ftp://ftp.computing.dcu.ie/pub/crypto/fast_mod_mult2.ps`.

[72] M. Scott. Comparison of methods for modular exponentiation on 32-bit Intel 80x86 processors, June 1996. Informal draft, available for download at `ftp://ftp.computing.dcu.ie/pub/crypto/timings.ps`.

[73] M. Scott and P.S.L.M. Barreto. Generating more MNT curves. Report 2004/058, Cryptology ePrint Archive, 2004.

[74] V. Shoup. OAEP reconsidered. *Journal of Cryptology*, 15(4):223–249, 2002.

[75] J.H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer-Verlag, 1986.

[76] J.A. Solinas. Low-weight binary representations for pairs of integers. Technical Report CORR-2001-41, Dept of C&O, University of Waterloo, Canada, 2001.

[77] M. Stam. *Speeding up Subgroup Cryptosystems*. PhD thesis, Technische Universiteit Eindhoven, Holland, 2003.

[78] M. Stam and A.K. Lenstra. Speeding up XTR. In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 1976 of *Lecture Notes in Computer Science*, pages 125–143. Springer-Verlag, 2001.

[79] M. Stam and A.K. Lenstra. Efficient subgroup exponentiation in quadratic and sixth degree extensions. In B.S. Kaliski, Jr., Ç.K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 318–332. Springer-Verlag, 2002.

[80] N. Thériault. Index calculus attack for hyperelliptic curves of small genus. In C.-S. Laih, editor, *Advances in Cryptology – ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 75–92. Springer-Verlag, 2003.

[81] C. Wolf. Efficient public key generation for multivariate cryptosystems. Report 2003/089, Cryptology ePrint Archive, 2003.

[82] B.-Y. Yang, J.-M. Chen, and N. Courtois. On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis. In J. Lopez, S. Qing, and E. Okamoto, editors, *Information and Communications Security – ICICS 2004*, volume 3269 of *Lecture Notes in Computer Science*, pages 401–413. Springer-Verlag, 2004.

Appendix A

Basic Algorithms

A.1 Multiple-precision multiplication

In this section we analyze three principal methods to perform a multiple-precision multiplication: the schoolbook method [60], Comba's method [26], and Karatsuba's method [51]. These three methods form the basis of the algorithms for Montgomery multiplication discussed in Section A.2. The schoolbook method represents the most straightforward way to realize a multiple-precision multiplication and is covered in many textbooks. However, the two other methods may perform better in practice. Comba's method requires fewer memory accesses (in particular store operations), whereas Karatsuba's method reduces the number of multiply instructions.

We represent long integers as arrays of Ω-bit digits. A typical choice for Ω is the word-size of the processor. The bitlength of the integers is denoted by n, and k is the number of digits necessary to store them, whereby $k = \lceil n/\Omega \rceil$. For example, a 256-bit integer requires $k = 8$ digits on a 32-bit architecture. We shall denote long integers by uppercase letters and use the corresponding lowercase letters for the individual Ω-bit digits, e.g., $A = (a_{k-1}, \ldots, a_1, a_0)$ with $0 \leq a_i < 2^{\Omega}$.

A.1.1 Schoolbook method

The schoolbook method, shown in Algorithm A.1, consists of two nested loops, each looping through the digits of one operand. In each iteration of the outer loop, a digit b_i of the operand B is multiplied by all digits of the operand A, and the $(n + \Omega)$-bit results are accumulated according to their weight. The schoolbook method is also called *operand scanning method* since the outer loop moves through the digits of an operand.

An ordered pair of the form (u, v) represents the 2Ω-bit (i.e., double-precision) integer $u \cdot 2^{\Omega} + v$. The schoolbook method performs an operation of the form $a \cdot b + p + u$ in its inner loop, whereby a, b, p, and u are all Ω-bit quantities. Therefore, the result of this inner-loop operation is at most 2Ω bits long. This makes the schoolbook method easy to implement in high-level programming languages which provide a double-precision integer datatype. For instance, common extensions of the C and C++ programming language support the datatype `unsigned long long` for 64-bit integers. The Java language provides the `long` type, which has a precision of 64 bits on all platforms.

Algorithm A.1 Multiple-precision multiplication (schoolbook method).

Require: Two k-word operands $A = (a_{k-1}, \ldots, a_1, a_0)$ and $B = (b_{k-1}, \ldots, b_1, b_0)$
Ensure: $2k$-word product $P = A \cdot B = (p_{2k-1}, \ldots, p_1, p_0)$

```
1:  P ← 0
2:  for i = 0 to k − 1 do
3:      u ← 0
4:      for j = 0 to k − 1 do
5:          (u, v) ← a_j · b_i + p_{i+j} + u
6:          p_{i+j} ← v
7:      end for
8:      p_{k+i} ← u
9:  end for
10: return P
```

Algorithm A.2 Integer multiplication with result-in-place.

Require: Two k-word operands $A = (a_{k-1}, \ldots, a_1, a_0)$ and $B = (b_{k-1}, \ldots, b_1, b_0)$
Ensure: $\textsc{IntMult}(A, B) = (C, A) \leftarrow A \cdot B$

```
1:  C ← 0; c ← 0
2:  for i = 0 to k − 1 do
3:      tmp ← a_i
4:      (c, a_i) ← tmp · b_0 + c_0
5:      for j = 0 to k − 2 do
6:          (c, c_j) ← tmp · b_{j+1} + c_{j+1} + c
7:      end for
8:      c_{k−1} ← c
9:  end for
10: return (C, A)
```

The schoolbook multiplication requires

$$\boxed{k^2 \text{ CPU multiplications}.}$$

Let $\beta = 2^{\Omega}$. The square of a long integer can be computed almost twice as fast as the product of two distinct integers, which can be observed from Eq. (A.1):

$$A^2 = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} a_j \cdot a_i \cdot \beta^{(i+j)} = \sum_{i=0}^{k-1} {a_i}^2 \cdot \beta^{2i} + 2 \cdot \sum_{i=0}^{k-2} \sum_{j=i+1}^{k-1} a_j \cdot a_i \cdot \beta^{(i+j)} \quad . \tag{A.1}$$

Long integer squaring is typically performed in two steps. In the first step, all inner-product terms $a_j \cdot a_i$ with $j \neq i$ are calculated and summed up as shown in Eq. (A.1). The second step doubles the result obtained in the first step and adds the inner products from the "main diagonal", i.e., the terms ${a_i}^2$. Doing so, squaring requires only

$$\boxed{\tfrac{1}{2}\, k(k+1) \text{ CPU multiplications}.}$$

When the memory is scarce, the schoolbook method can be accomodated with "result-in-place" to evaluate $A \cdot B$ with rewriting in buffer A:

$$(C, A) \leftarrow A \cdot B$$

where C represents a k-word "register" (see Algorithm A.2). Using this algorithm, the integer multiplication of A and B can be evaluated with a k-word working "register" (i.e., C) and a few CPU registers. Further the value of operand B is still available in memory at the end of the computation.

A.1.2 Comba's method

Algorithm A.3 illustrates an alternative method to accomplish a long integer multiplication. This method, first described by Comba [26], also consists of a nested loop structure with a relatively simple inner loop. The two outer loops of Algorithm A.3 move through the digits p_i of the product P, and therefore Comba's method is also referred to as *product scanning method*. To obtain the i-th digit p_i of $P = A \cdot B$, all inner-product terms $a_j \cdot b_{i-j}$ with $0 \leq j \leq i$ are accumulated and eventually added to carries from the computation of previous digits. The store operation corresponding to each digit of the result takes only place in the outer loop, when the digit is completely evaluated.

Comba's method performs multiply/accumulate (MAC) operations in its inner loop, which means that two Ω-bit digits are multiplied and the 2Ω-bit product is added to a cumulative sum. This sum can easily get longer than 2Ω bits and hence we need three Ω-bit registers for its storage. Algorithm A.3 represents these three registers by the triple (t, u, v). The operation carried out at Lines 7 and 14 is just a Ω-bit right-shift of (t, u, v). However, the extended precision of the cumulative sum makes an implementation of Comba's method rather difficult when using high-level programming languages like C/C++ or Java, since they have neither triple-precision data types, nor built-in support for handling carries in an efficient way. Intel's application note AP-941 [5] describes two solutions for this problem. One possibility is to use less bits in the digits (e.g., 28 instead of 32), so that several products can be accumulated into a 64-bit integer without overflow. Another solution is to store the cumulative sum in two 64-bit integers (see [5] for further details). However, the price of the former is an increase in the number of digits, while the latter entails more additions in the inner loop.

Algorithm A.3 Multiple-precision multiplication (Comba's method).

Require: Two k-word operands $A = (a_{k-1}, \ldots, a_1, a_0)$ and $B = (b_{k-1}, \ldots, b_1, b_0)$
Ensure: $2k$-word product $P = A \cdot B = (p_{2k-1}, \ldots, p_1, p_0)$

```
(t, u, v) ← 0
for i = 0 to k − 1 do
    for j = 0 to i do
        (t, u, v) ← (t, u, v) + a_j · b_{i−j}
    end for
    p_i ← v
    v ← u, u ← t, t ← 0
end for
for i = s to 2k − 2 do
    for j = i − k + 1 to k − 1 do
        (t, u, v) ← (t, u, v) + a_j · b_{i−j}
    end for
    p_i ← v
    v ← u, u ← t, t ← 0
end for
p_{2s−1} ← v
return P
```

A.1.3 Karatsuba's method

Karatsuba's method reduces a multiplication of two k-digit operands to three multiplications of size $k/2$, but at the cost of an increased number of additions [51]. The three half-size multiplications can either be performed with the schoolbook method, Comba's method, or again Karatsuba's method, provided that the operands are large enough. A product of two k-digit operands with methods such as the schoolbook method or Comba's requires to calculate k^2 single-precision multiplications. Karatsuba's method performs only $3k^2/4$ single-precision multiplications. However, when applied recursively, Karatsuba's method results in an algorithm with complexity $O(k^{\log_2 3})$ where $\log_2 3 \approx 1.58$.

In order to explain Karatsuba's method, let us assume, for simplicity, that the number of digits k is even. The operands A and B are split into two parts of equal length, whereby A_L, B_L consist of the $k/2$ least significant digits, and A_H, B_H of the $k/2$ most significant digits of A and B, respectively. Let $\beta = 2^{\Omega}$. Since $A = A_H \cdot \beta^{k/2} + A_L$ and $B = B_H \cdot \beta^2 + B_L$, the product $P = A \cdot B$ can be computed as according to the following equation:

$$P = A_H \cdot B_H \cdot \beta^k + [A_H \cdot B_H + A_L \cdot B_L - (A_H - A_L) \cdot (B_H - B_L)] \cdot \beta^{k/2} + A_L \cdot B_L \ . \quad \text{(A.2)}$$

A graphical representation of Karatsuba's method is given in Figure A.1. It is also possible to perform the calculation with the absolute value for $(A_H - A_L) \cdot (B_H - B_L)$ and to use the sign to decide whether this value is added to or subtracted from $A_H \cdot B_H + A_L \cdot B_L$ [52]. Note that carries may propagate from the most significant words of $A_H \cdot B_H$, $A_L \cdot B_L$, and $(A_H - A_L) \cdot (B_H - B_L)$ when they are added. Karatsuba squaring is similar to multiplication, but with $A = B$ the equation reduces to three $(k/2)$-digit squares that have to be added as shown in Figure A.1. The middle term $(A_H - A_L)^2$ is always positive, which simplifies the implementation of Karatsuba squaring [48].

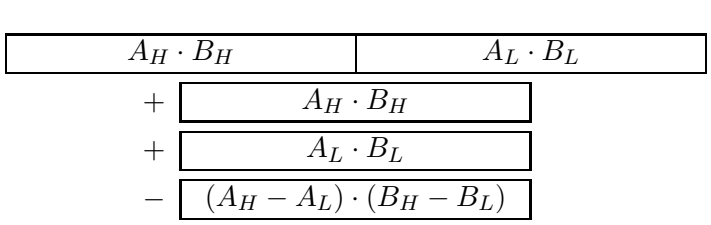

Figure A.1: Graphical representation of Karatsuba's method.

A.1.4 Analysis of the algorithms

Both the execution time and the energy consumption of the algorithms described in this section depend heavily on the concrete implementation. An implementer could, for instance, fully unroll the inner and outer loops of the algorithms. In this case, only the base instructions like multiplies, adds, loads and stores have to be performed. However, while loop unrolling allows to achieve the best possible performance, it can significantly increase the code size, especially when the number of digits is large. On the other hand, an implementation with "rolled" loops represents the other end of the spectrum. Rolled-loop implementations do not only execute the base instructions mentioned above, but also instructions which do not directly contribute to the calculation of the result. We may think about operations such as incrementing loop counters, branch instructions, register moves, or pointer arithmetic. While an implementation with rolled loops has the benefit of small code-size, it can be significantly slower than an optimized variant with unrolled loops. This makes it necessary to find a trade-off between performance (i.e. unrolled loops) and code-size (i.e., rolled loops). One possible solution is to *partially unroll* the loops. For instance, the body of the loop can be replicated multiple times (e.g., 8 or 16 times), which replaces a number of loop iterations by non-iterated straight-line code. Partial loop unrolling eliminates, or substantially reduces, the effects of the loop overhead (i.e., incrementing the loop counter, branch instruction, etc.). In such case, the loop overhead is (almost) negligible, which means that the execution time and the energy consumption are primarily determined by the base instructions.

Table A.1: Comparison of base instructions for long integer multiplication algorithms.

Algorithm	# `MUL`	# `ADD`
Schoolbook Multiplication	k^2	$4k^2$
Comba Multiplication	k^2	$3k^2$
Karatsuba/Schoolbook Multiplication	$3/4\,k^2$	$3k^2 + ks + 2$
Karatsuba/Comba Multiplication	$3/4\,k^2$	$9/4\,k^2 + 4k + 2$

Table A.1 summarizes the number of *base instructions* (i.e., multiplies and adds).for the algorithms described before. The schoolbook method performs exactly k^2 iterations of the inner loop. In each iteration, an operation of the form $a \cdot b + p + u$ is executed, i.e., two Ω-bit digits are multiplied and another two Ω-bit digits are added to the product. Note that adding a single-precision digit to a double-precision digit actually involves two `ADD` instructions since

a single-precision addition may produce a carry which has to be processed properly.[1]

Comba's method also iterates the inner loop exactly k^2 times. Therefore, we have k^2 multiplications and $3k^2$ single-precision additions since the accumulation of a 2Ω-bit product to the running sum in (t, u, v) requires one `ADD` and two `ADC` instructions.

A Karatsuba multiplication of two k-digit operands basically consists of three $(k/2)$-digit multiplications and five $(k/2)$-digit additions or subtractions. Table A.1 shows the number of base instructions when using the schoolbook method or Comba's method for the half-size multiplications (we do not apply Karatsuba's trick recursively). Note that the addition or subtraction of the k-digit products $A_H \cdot B_H$, $A_L \cdot B_L$, and $(A_H - A_L) \cdot (B_H - B_L)$ may produce a carry. For simplicity, we count one `ADD` for the processing of this carry.

A.2 Modular multiplication

Three efficient algorithms are known to compute a modular reduction [25]. This section presents the Montgomery multiplication algorithm [63]. For any R relatively prime to modulus N, the Montgomery method represents equivalence classes of $A \pmod N$ as $\widetilde{A} \stackrel{\text{def}}{=} AR \pmod N$ and redefines the modular multiplication as

$$\textsc{MontMult}(\widetilde{A}, \widetilde{B}, N) = ABR^{-1} \bmod N \ . \tag{A.3}$$

Hence, we see that Montgomery multiplication with the new representation is "isomorphic" to the ordinary modular multiplication:

$$\begin{aligned} & \textsc{MontMult}(AR \bmod N, BR \bmod N, N) = (AB)R \bmod N \\ \iff & \textsc{MontMult}(\widetilde{A}, \widetilde{B}, N) = \widetilde{A \cdot B} \ . \end{aligned}$$

Montgomery algorithm relies on the key observation that

$$PR^{-1} \bmod N = \frac{P + QN}{R} - \epsilon N \tag{A.4}$$

with $\epsilon \in \{0, 1\}$ and $Q = PN' \bmod R$ where $N' = -N^{-1} \bmod R$. In particular, when N is odd and R is a power of $\beta = 2^\Omega$ on a Ω-bit processor, Montgomery modular reduction can be evaluated easily since division by R amounts to a shift operation.

There are five basic algorithms given in [53] to perform the Montgomery multiplication in general-purpose computers. These algorithms can be classified based on two factors. The first factor is whether multiplication and reduction are *separated* or *integrated*. In the former method, operands A and B are first multiplied and then a reduction is performed. In the integrated approach, multiplication and reduction are performed in an interleaved manner. The integration is *coarse-grained* if we do not alternate between multiplication and reduction very frequently. Otherwise, it is called *fine-grained*. The second factor is about how the multiplication and reduction steps are performed. In the *operand scanning* form, an outer loop moves through words of one of the operands. In the *product scanning* form, the loop moves through words of the product.

[1] More precisely, an `ADD` and an `ADC` (add with carry) instruction are required. However, we ignore this distinction in our analysis and count only the number of single-precision additions, regardless of whether or not the carry flag is considered.

Based on this classification, [53] proposes and analyzes five different algorithms:

1. SOS: separated operand scanning;
2. CIOS: coarsely integrated operand scanning;
3. FIOS: finely integrated operand scanning;
4. FIPS: finely integrated product scanning;
5. CIHS: coarsely integrated hybrid scanning.

It was reported in [53] that the CIOS algorithm results in the best performance on a general-purpose computer. The CIOS algorithm is simple to implement and has the same inner loop as the pencil-and-paper method.

We give in Algorithm A.4 a (high-level) description the CIOS version (a.k.a. interleaved version) of Montgomery multiplication algorithm as presented in [60, § 14.3.2]. This version presents the advantage of requiring less memory; on the minus side, however, it does not allow to optimize the modular squaring.

Algorithm A.4 Montgomery multiplication (interleaved version).

Require: $N < \beta^k$, odd, with $\beta = 2^\Omega$, $n'_0 = -N^{-1} \bmod \beta$,
$A = \sum_{i=0}^{k-1} a_i \, \beta^j = (a_{k-1}, \ldots, a_0)$ and $B = \sum_{i=0}^{k-1} b_j \, \beta^j = (b_{k-1}, \ldots, b_0)$
with $0 \leq A, B < N$

Ensure: $\textsc{MontMult}(A, B, N) = ABR^{-1} \bmod N$ with $R = \beta^k$

1: $R_0 \leftarrow 0$
2: **for** $i = 0$ to $k - 1$ **do**
3: $\quad u \leftarrow \big((R_0 \bmod \beta) + (a_0 \cdot b_i \bmod \beta)\big) \cdot n'_0 \bmod \beta$
4: $\quad R_0 \leftarrow (R_0 + A \cdot b_i + N \cdot u) \div \beta$
5: **end for**
6: **if** $R_0 \geq N$ **then**
7: $\quad R_0 \leftarrow R_0 - N$
8: **end if**
9: **return** R_0

It is worth noting that $R_0 < 2N - 1$ at Line 4, $\forall i$. Indeed, we have

$$\frac{R_0 + A\, b_i + N\, u}{\beta} \leq \frac{(2N-2) + (N-1)(\beta-1) + N(\beta-1)}{\beta}$$
$$= 2N - 1 - \frac{1}{\beta} < 2N - 1 \ .$$

A.2.1 Complexity

The evaluation of u (Line 3) requires 2 CPU multiplications. The evaluation of R_0 (Line 4) requires 2 multiplications of a word by a k-word integer, which can be done with $2k$ CPU multiplications (see § A.2.2). Integrating the shifting into the reduction allows to save a multiplication [53] (see the full description given in Algorithm A.5). Hence, with the previous slight improvement, for a k-word modulus N, a Montgomery multiplication requires

$$\boxed{k(2k+1) \text{ CPU multiplications}\,.}$$

A.2.2 Memory

Neglecting a few CPU registers, in addition to the memory needed to store A, B, N and n'_0, the Montgomery multiplication MontMult(A, B, N) requires

$$\boxed{(k+2) \text{ words of working memory}.}$$

A.2.3 Detailed implementations

Coarsely Integrated Operand Scanning (CIOS)

Algorithm A.5 Montgomery multiplication (Coarsely Integrated Operand Scanning).

Require: $N < \beta^k$, odd, with $\beta = 2^\Omega$, $n'_0 = -N^{-1} \bmod \beta$,
$A = \sum_{i=0}^{k-1} a_i \beta^j = (a_{k-1}, \ldots, a_0)$ and $B = \sum_{i=0}^{k-1} b_j \beta^j = (b_{k-1}, \ldots, b_0)$
with $0 \le A, B < N$

Ensure: MontMult$(A, B, N) = ABR^{-1} \bmod N$ with $R = \beta^k$

1: $Z \leftarrow 0$
2: **for** $i = 0$ to $k-1$ **do**
3: $\quad u \leftarrow 0$
4: $\quad$ **for** $j = 0$ to $k-1$ **do**
5: $\quad\quad (u, v) \leftarrow a_j \cdot b_i + z_j + u$
6: $\quad\quad z_j \leftarrow v$
7: $\quad$ **end for**
8: $\quad (u, v) \leftarrow z_k + u$
9: $\quad z_k \leftarrow v$
10: $\quad z_{k+1} \leftarrow u$
11: $\quad q \leftarrow z_0 \cdot n'_0 \bmod \beta$
12: $\quad (u, v) \leftarrow z_0 + n_0 \cdot q$
13: $\quad$ **for** j from 1 by 1 to $k-1$ **do**
14: $\quad\quad (u, v) \leftarrow n_j \cdot q + z_j + u$
15: $\quad\quad z_{j-1} \leftarrow v$
16: $\quad$ **end for**
17: $\quad (u, v) \leftarrow z_k + u$
18: $\quad z_{k-1} \leftarrow v$
19: $\quad z_k \leftarrow z_{k+1} + u$
20: **end for**
21: **if** $Z \ge N$ **then**
22: $\quad Z \leftarrow Z - N$
23: **end if**
24: **return** Z

In CIOS, execution of the algorithm alternates between the iterations of the loops for multiplication (Lines 4–7) and reduction (Lines 12-16). In the first loop, a word of the multiplier a_i is multiplied by the entire multiplicant B and the result is added to the value in Z. Therefore, the value in Z may be $k+2$ words. In the second inner loop, the temporary result is shortened by one word utilizing the precomputed value $n'_0 = -n_0^{-1} \bmod \beta$. Consequently, after each iteration in the outer loop, the intermediate result does not become longer than

$k + 1$ words. Final subtraction in the last step guarantees that the result is in the desired range (i.e., $Z < N$).

Karatsuba-Comba-Montgomery (KCM) Multiplication

Karatsuba-Comba-Montgomery (KCM) multiplication algorithm [71, 72, 8] implements the Montgomery multiplication in three steps, as given by Eq. (A.4),

1. the multiplication $P = A \cdot B$;
2. the calculation of $Q = P \cdot N' \bmod R$; and
3. the calculation of the actual Montgomery residue $Z = (P + Q \cdot N)/R$.

The last two steps implement the Montgomery reduction.

The calculation of $A \cdot B$ is performed using Karatsuba-Comba multiplication, where the recursion in the Karatsuba reduces the number of multiplications while the Comba part reduces the number of memory references. The Montgomery reduction is performed using Comba-style multiplications.

Algorithm A.6 Montgomery reduction (product scanning form).

Require: $N < \beta^k$, odd, with $\beta = 2^\Omega$, $n'_0 = -N^{-1} \bmod \beta$,
$P = \sum_{i=0}^{2k-1} p_i \, \beta^j = (p_{2k-1}, \ldots, p_0)$ with $0 \le P < 2N - 1$
Ensure: Montgomery residue $Z = P\,R \bmod N$ with $R = \beta^k$

1: $(t, u, v) \leftarrow 0$
2: **for** $i = 0$ to $k - 1$ **do**
3: **for** $j = 0$ to $i - 1$ **do**
4: $(t, u, v) \leftarrow (t, u, v) + z_j \cdot n_{i-j}$
5: **end for**
6: $(t, u, v) \leftarrow (t, u, v) + p_i$
7: $z_i \leftarrow v \cdot n'_0 \bmod \beta$
8: $(t, u, v) \leftarrow (t, u, v) + z_i \cdot n_0$
9: $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
10: **end for**
11: **for** $i = k$ to $2k - 2$ **do**
12: **for** $j = i - k + 1$ to $k - 1$ **do**
13: $(t, u, v) \leftarrow (t, u, v) + z_j \cdot n_{i-j}$
14: **end for**
15: $(t, u, v) \leftarrow (t, u, v) + p_i$
16: $z_{i-k} \leftarrow v$
17: $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
18: **end for**
19: $(t, u, v) \leftarrow (t, u, v) + p_{2k-1}$
20: $z_{k-1} \leftarrow v,\ z_k \leftarrow u$
21: **if** $Z \ge N$ **then**
22: $Z \leftarrow Z - N$
23: **end if**
24: **return** P

The first outer loop in Algorithm A.6 (Lines 2–10) calculates k words of $Q = P \cdot N' \bmod R$ and stores them in $(z_{k-1}, \ldots, z_1, z_0)$. Thereafter, the second loop (Lines 11–20) produces the actual Montgomery residue $Z = (P + Q \cdot N)/R$. Both loops implement one multi-precision multiplication operation each. The first multiplication requires the bottom half of the result while the second does the upper half. Therefore, these two modular multiplications can be efficiently computed using Comba method [72]. For more information on KCM multiplication, one can profitably refer to [71, 72, 8].

A.3 Modular exponentiation

Again, there are various ways to evaluate an exponentiation [46]. This section reviews the celebrated square-and-multiply algorithm for computing $y = x^e \bmod N$. Advantageously, the value of x is still available in register R_1 at the end of the computation.

Algorithm A.7 Square & multiply.

Require: $N, 0 \leq x < N$ and e with $e = \sum_{j=0}^{\ell-1} e_j \, 2^j$ where $e_j \in \{0, 1\}$ and $e_{\ell-1} = 1$
Ensure: $y = x^e \bmod N$
1: $R_0 \leftarrow x$; $R_1 \leftarrow R_0$
2: **for** $j = \ell - 2$ downto 0 **do**
3: $\quad R_0 \leftarrow (R_0)^2 \pmod N$
4: $\quad$ **if** $e_j = 1$ **then**
5: $\quad\quad R_0 \leftarrow R_0 \cdot R_1 \pmod N$
6: $\quad$ **end if**
7: **end for**
8: **return** R_0

Normalization and denormalization steps are necessary to use it together with Montgomery multiplication (see Lines 1 and 8 in Algorithm A.8).

Algorithm A.8 Square & Multiply with Montgomery multiplication.

Require: $N < \beta^k$, odd, with $\beta = 2^\Omega$, $n'_0 = -N^{-1} \bmod \beta$, $\mathfrak{R} = R^2 \bmod N$
$\quad 0 \leq x < N$ and e with $e = \sum_{j=0}^{\ell-1} e_j \, 2^j$ where $e_j \in \{0, 1\}$ and $e_{\ell-1} = 1$
Ensure: $y = x^e \bmod N$
1: $R_0 \leftarrow \textsc{MontMult}(x, \mathfrak{R}, N)$; $R_1 \leftarrow R_0$
2: **for** $j = \ell - 2$ downto 0 **do**
3: $\quad R_0 \leftarrow \textsc{MontMult}(R_0, R_0, N)$
4: $\quad$ **if** $e_j = 1$ **then**
5: $\quad\quad R_0 \leftarrow \textsc{MontMult}(R_0, R_1, N)$
6: $\quad$ **end if**
7: **end for**
8: $R_0 \leftarrow \textsc{MontMult}(R_0, 1, N)$
9: **return** R_0

Note also that the value of x is no longer available in R_1 at the end of the computation but we can easily recover it as $R_1 \leftarrow \textsc{MontMult}(R_1, 1, N)$.

Appendix B

Index to Notations

B.1 General symbols

Formal symbolism	Meaning
$\|$	Bit-string concatenation
$\{0,1\}^\ell$	Set of bit-strings of length ℓ
$\#\mathcal{A}$	Cardinality of set $\mathcal{A}$
$\|a\|_2$	Binary length of a
$a \div b$	Integer division: $a \div b = \lfloor a/b \rfloor$
$a \mid b$	a divides b
$\mathbb{F}_q$	Finite field with q elements
$\overline{\mathbb{F}}_q$	Algebraic closure of $\mathbb{F}_q$
$\lambda(N)$	Carmichael's function of N
$\mathrm{N}_{L/K}(a)$	Norm of an element $a \in L$ over K
$\mathrm{Tr}_{L/K}(a)$	Trace of an element $a \in L$ over K
$\mathbb{Z}_N$	Ring of integers modulo N
$\mathbb{Z}_p$	Ring of p-adic integers
$\mathbb{Q}_p$	Field of p-adic integers
$\mathbb{Q}_q$	Unramified extension of $\mathbb{Q}_p$ with $q = p^N$
$\mathrm{Jac}(C)$	Jacobian variety of a curve C
$\mathrm{Jac}(C)(\mathbb{F}_q)$	Jacobian group (class group, Picard group) of a curve $C/\mathbb{F}_q$

B.2 Costs

Formal symbolism	Meaning
$\mathsf{A}_{\mathbb{F}_q}$	Addition in $\mathbb{F}_q$
$\mathsf{A}_{\mathbb{G}_1}$	Addition in $\mathbb{G}_1$
$\mathsf{A}_{\mathbb{Z}_q}$	Addition in $\mathbb{Z}_q$
$\mathsf{A}_{\mathbb{Z}_N}$	Addition in $\mathbb{Z}_N$
$\mathsf{E}_{\mathbb{G}_1}$	Exponentiation in $\mathbb{G}_1$

Formal symbolism	Meaning
$\mathsf{E}_{\mathbb{G}_2}$	Exponentiation in $\mathbb{G}_2$
H	Generic hash class
$\mathsf{H}_{\mathbb{G}_1}$	Hash into $\mathbb{G}_1$
$\mathsf{H}_{\mathbb{G}_2}$	Hash into $\mathbb{G}_2$
$\mathsf{H}_{\mathbb{Z}_q}$	Hash into $\mathbb{Z}_q$
$\mathsf{I}_{\mathbb{G}_2}$	Inversion in $\mathbb{G}_2$
$\mathsf{I}_{\mathbb{Z}_q}$	Inversion in $\mathbb{Z}_q^*$
$\mathsf{M}_{\mathbb{F}_q}$	Multiplication in $\mathbb{F}_q$
$\mathsf{M}_{\mathbb{G}_2}$	Multiplication in $\mathbb{G}_2$
$\mathsf{M}_{\mathbb{Z}_N}$	Multiplication in $\mathbb{Z}_N$
$\mathsf{M}_{\mathbb{Z}_q}$	Multiplication in $\mathbb{Z}_q$
$\mathsf{N}_{\mathbb{G}_1}$	Negation in $\mathbb{G}_1$
P	Pairing evaluation
PM	Point multiplication
TA	T-point addition
TD	T-point doubling
TT	T-point tripling

B.3 Acronyms

Formal symbolism	Meaning
AES	Advanced Encryption Standard
AGM	Arithmetic Geometric Mean
BKLS	Barreto-Kim-Lynn-Scott
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
DES	Data Encryption Standard
DL	Discrete Logarithm
DLP	Discrete Logarithm Problem
IBE	Identity-Based Encryption
ECC	Elliptic Curve Cryptography
GQ	Guillou-Quisquater
KCM	Karatsuba-Comba-Montgomery
MNT	Miyaji-Nakabayashi-Takano
MPC	Multivariate-Polynomial based Cryptosystem
NAF	Non-Adjacent Form
OAEP	Optimal Asymmetric Encryption Padding
PK	Public Key
PM	Pseudo-Mersenne
PSS	Probabilistic Signature Scheme
RSA	Rivest-Shamir-Adleman
TDES	Triple DES