Diffie-Hellman Key Exchange – From Mathematics to Real Life

Suvi Lehtinen

University of Tampere
School of Information Sciences
Computer Science
M.Sc thesis
Supervisor: Martti Juhola
December 2011

University of Tampere
School of Information Sciences
Computer Science
Suvi Lehtinen: Diffie-Hellman Key Exchange - From Mathematics to Real Life
M.Sc thesis, 56 pages
December 2011

Abstract

This thesis gives an introduction to classical Diffie-Hellman Key Exchange and its variant for elliptic curves. The needed mathematical background is given and the discrete logarithm problem is shortly introduced, but the main focus is on algorithms for modular multiplication and correspondingly for scalar multiplication. The theoretical complexity of the needed IEEE algorithms is analyzed and compared to the experimental results achieved from a small-scale performance evaluation of one cryptographic library that has an implementation of these algorithms.

Key words and terms: Diffie-Hellman Key Exchange, Elliptic Curve Cryptography, Modular Multiplication, Scalar Multiplication, Implementation of Public Key Cryptography, Performance Evaluations

Contents

1 Introduction

1.1 Cryptography around us

Most of the information nowadays is in digital format that can be passed around and abused easily, unless it is protected. Cryptographic algorithms provide means for protecting the content (encryption) and integrity (hash functions), and verifying authenticity (digital signatures).

One general way to categorize is to make a division to symmetric (secret) key cryptography and asymmetric (public) key cryptography. As the name suggest, in symmetric key cryptography the same key must be shared between the parties and, hence, kept secret. Examples of algorithms based on secret keys are DES, 3DES, AES and keyed hashing functions (e.g., HMAC). These algorithms tend to be quite fast, but the problem is that they require key-exchange or pre-shared keys that should be partner-wise unique. Asymmetric key cryptography allows one to publish a public key that can be used for encryption or verification of a signature by anyone, while its private counterpart is used for decryption or signing only by the owner of the key pair.

Most commonly used public key algorithms can be divided to those whose security is based on the difficulty of integer factorization and to those, whose security is based on the difficulty of the discrete logarithm problem. Both problems are believed to be hard to solve on large enough numbers. RSA is based on integer factorization whereas Diffie-Hellman, DSA and Elgamal are based on the discrete logarithm problem. It should be noted that even though the security considerations are based on different mathematical problems, from the perspective of implementation all public key algorithms mentioned above are based on how fast an exponentiation one can make.

The problem with traditional public key cryptography is that in order to stay ahead the computational power of modern computers, the key sizes must keep growing. 1024-bit RSA is already getting out-dated, and if you think on a longer scale, you should already start to consider supporting longer than 2048-bit keys (see for example http://www.keylength.com/, link tested 01-Dec-2011). One so-

lution is to use elliptic curves that provide the same security levels with substantially shorter key lengths. Diffie-Hellman, DSA and Elgamal algorithms all have elliptic curve variants and their security is based on the elliptic curve discrete logarithm problem.

The problem with asymmetric (public) key cryptography is that it tends to be much slower than symmetric key cryptography. This can be solved by using public key methods for creating a shared secret that can then be used for creating a symmetric key that can further be used for faster ciphering or HMACs. In 1976, Whitfield Diffie and Martin Hellman published a neat method, Diffie-Hellman key exchange, which is a crucial part of many cryptographic protocols like handshake in Transport Layer Security (TLS) [27]. As in some contexts (mobile) it would be good to keep the key sizes relatively small, the elliptic curve variant of Diffie-Hellman key exchange was also proposed for TLS in 2006 [28]. Authentication mechanisms are usually needed on top of these key exchange algorithms, but understanding the key exchange is a good start. Authentication is needed to tackle man-in-the-middle attacks. This means that (without proper authentication of the parties) a third party might be able to act between the actual parties and learn the secrets that where not meant for her.

1.2 About this work

When put into mathematical formulas, cryptography seemed so simple, nice and neat. All you need is some simple arithmetical operations – a couple of multiplications, some additions and maybe even an exponentiation. With sophisticated mathematical programs even large numbers were not a problem. In real life it was something else.

It all became clear to me when I left the university and started adopting knowledge of the cryptographic functionalities that are hidden in our phones. Before that day, I had practically no experience in cryptography. So, I did what any long-term university student would do: I checked the nearest cryptography-related courses and enrolled on.

Cryptography seemed quite reasonable from mathematical point of view. You

have an underlying mathematical structure, say a *group*. You have certain group operations. You have a mathematical property that is relatively easy to compute one way but hard to reverse, and there you have a cryptographic system. Because you are using big enough numbers with a suitable mathematical structure, your system is safe according to current knowledge, that is, as safe as it can be. You can cipher your secrets and make digital signatures just like that (at least in theory).

In practice there are some issues. Firstly, standard opening of a math course does not work here – you can not say to a computer: "Let's assume that we have a suitable mathematical structure." You will have to define it yourself (or find it in some suitable standard). Secondly, we are talking about 128-bit to at least 2048-bit numbers, whereas standard computer arithmetics can handle 8-bit to 32-bit or even 64-bit numbers. You will need quite many of those in a row before you get a 2048-bit number. So you will need your own arithmetics, preferably a fast one, as we are talking about large numbers.

From mathematical point of view, it would seem that if you get all this fixed, then you will just do some basic calculations, and there you have it. But mostly this is not the case. Of course, if you do it for fun and you are not short of resources, this might do. If you do it for real, you will need to go through at least a couple of standards and check the availability of your resources. Especially in the field of mobile phones, you do not want to waste resources; neither time nor space, not to mention money. Even if you made decisions on these, it might still not be straightforward. You can make it faster by using more space, but what if you want to optimize the use of both time and space? Sometimes, by over-optimizing, you might actually end up losing money, as especially in the field of elliptic curves, many optimization tricks are patented – or even worse, optimization might lead to new security holes.

In this thesis I try to shed some light on Diffie-Hellman Key Exchange, a simple protocol for setting up a common secret between two parties who are often called Alice and Bob. I chose this protocol because its simplicity gives an opportunity to go into great detail with the algorithms.

Algorithm 1 (Diffie-Hellman Key Exchange)

1. Alice and Bob agree on a prime number p and a base g such that certain conditions are satisfied. These numbers are public.

2. Alice and Bob choose secret random positive integers a and b respectively.

3. Alice sends $g^a \bmod p$ to Bob and Bob sends $g^b \bmod p$ to Alice.

4. Both raise the received numbers to the powers of their private number to achieve the shared secret $g^{ab} \bmod p$.

It is easy to see that the main part in this protocol is exponentiation, so understanding this on the algorithmic level helps to understand one significant part of any public key algorithm using exponentiation, for example RSA. I start with this original version of Diffie-Hellman method, and then move on to the elliptic curve variant. In both versions, Montgomery arithmetics [21] plays a key role in making the algorithms faster.

2 Mathematical Background

2.1 Classical Diffie-Hellman Key Exchange

By Wikipedia (http://en.wikipedia.org/wiki/Diffie-Hellman, link tested 01-Dec-2011), the original implementation of Diffie-Hellman protocol uses the *multiplicative group of integers modulo p*, where p is a *prime number* and g a *primitive root modulo p*. What does this mean? Let us go through this word by word.
Group is a set of elements together with a binary operation that fulfills certain properties. It is called multiplicative to denote that the group operation is multiplication (as opposed to being addition). In Definition 1 we put this into more mathematical form.

Definition 1 A structure $(G, \star)$ is a *group* if the following four properties hold:

1. $\star$ is an operation on G, that is, a rule that joins to each pair $(a, b) \in G \times G$ a unique element $a \star b \in G$.

2. $\star$ is associative, that is, $(a \star b) \star c = a \star (b \star c)$.

3. There exists a neutral element $e \in G$ with respect to $\star$, that is, $\forall a \in G : a \star e = e \star a = a$. Note that the neutral element is unique and is quite often denoted by 1 in multiplicative groups.

4. Every element $a \in G$ has an inverse $a' \in G$ such that $a \star a' = a' \star a = e$, where e is the neutral element. Note the inverse of a is quite often denoted by a^{-1} and it is unique.

For example, if we have a two-element set $\Gamma = \{\alpha, \beta\}$ and we define an operation $\star : \Gamma \times \Gamma \to \Gamma$ by the following table

$\star$	α	β
α	α	β
β	β	α

we get a group $(\Gamma, \star)$.

Let m be a fixed non-negative integer. The set of all *residue classes* mod m is denoted by $\mathbf{Z}_m$, that is,

$$\mathbf{Z}_m = \{\bar{0}, \bar{1}, \ldots, \bar{m-1}\},$$

where $\bar{x} = \{y \in \mathbf{Z} \mid \exists k \in \mathbf{Z} : y = x + km\}$.

Remark 1 The structure $(\mathbf{Z}_m, \cdot)$, where

$$\bar{a} \cdot \bar{b} = \overline{ab},$$

is a *commutative monoid*, that is, $\cdot$ is commutative ($\forall a \forall b : a \cdot b = b \cdot a$), associative and the neutral element exists. The structure $(\mathbf{Z}_m, \cdot)$ is not necessarily a group, because some residue classes might not have an inverse.

Definition 2 The *greatest common divisor* of a and b is the largest positive integer that divides both a and b, that is, integer c such that there exists $k_1, k_2 \in \mathbf{Z} : a = k_1 c$ and $b = k_2 c$. It is denoted by GCD(a,b) or (a, b). If $(a, b) = 1$, a and b are *relatively prime* or *coprime*.

Theorem 1 *An element $\bar{a}$ has inverse in monoid $(\mathbf{Z}_m, \cdot)$ iff $(a, m) = 1$ iff the equation $ax \equiv 1 \bmod m$ has a solution iff the Diophantine Equation $ax - my = 1$ has a solution.*

Definition 3 The structure $(G, \star)$ is called *Abelian group* if $\star$ is commutative, associative operation on G and the neutral element and inverses exist.

Definition 4 The elements of the set

$$\mathbf{Z}_m^\star = \{\bar{a} \in \mathbf{Z}_m \mid (a, m) = 1\}$$

are called *reduced residue classes* modulo m.

Theorem 2 *The structure $(\mathbf{Z}_m^\star, \cdot)$ is* Abelian group.

The group $G = (\mathbf{Z}_m^\star, \cdot)$ is **the multiplicative group of integers modulo** m and it has $\phi(m)$ elements (its order, ord(G) is $\phi(m)$), where ϕ is *Euler's totient function* (that gives the number of positive integers less or equal to m that are coprime to m).

Euler's totient function has the following properties

- $\phi(1) = 1$
- $\phi(p^k) = (p-1)p^{k-1}$, for any prime number p and $k \geq 1$
- if m and n are coprime, then $\phi(mn) = \phi(m)\phi(n)$.

We are mostly interested in multiplicative groups of integers modulo some prime.

Remark 2 If p is a prime number, $\mathbf{Z}_p^\star = \{\bar{1}, \ldots, \bar{p-1}\}$ and $\phi(p) = p - 1$.

Definition 5 A group $(G, \star)$ is a *cyclic group*, if

$$\exists g \in G : \langle g \rangle = \{g^k \mid k \in \mathbf{Z}\} = G.$$

Element g is called a *generator* of G.

Remark 3 Given a prime number p, the group $(\mathbf{Z}_p^\star, \cdot)$ is always a cyclic group that is generated by a single element. A generator of this cyclic group is called a *primitive root modulo* p or a *primitive element* of $\mathbf{Z}_p^\star$.

Definition 6 The *order of a group element* g, denoted by $\mathrm{ord}(g)$, is the smallest positive integer k such that $g^k = e$, where e is the neutral element of the group.

Remark 4 It is easy to see that for a generator g of a cyclic group $\mathbf{Z}_p^\star$, it always holds that $\mathrm{ord}(g) = \mathrm{ord}(\mathbf{Z}_p^\star) = p - 1$.

Example 1 Let $p = 5$. Now $(\mathbf{Z}_p^\star, \cdot) = (\{\bar{1}, \ldots, \bar{4}\}, \cdot)$, where $\cdot$ is defined as follows:

$\cdot$	$\bar{1}$	$\bar{2}$	$\bar{3}$	$\bar{4}$
$\bar{1}$	$\bar{1}$	$\bar{2}$	$\bar{3}$	$\bar{4}$
$\bar{2}$	$\bar{2}$	$\bar{4}$	$\bar{1}$	$\bar{3}$
$\bar{3}$	$\bar{3}$	$\bar{1}$	$\bar{4}$	$\bar{2}$
$\bar{4}$	$\bar{4}$	$\bar{3}$	$\bar{2}$	$\bar{1}$

The primitive roots mod p are $\bar{2}$ and $\bar{3}$.

What kinds of conclusions can we draw concerning Algorithm 1, based on the mathematics we have presented so far? First of all, the prime p that is used should be big enough. Otherwise, we could use a simple table lookup to find out Alice and Bob's secret values from the values they sent. If we for example know that $p = 5$ and $g = 2$, we get the following exponentiation table:

e	1	2	3	4
$exp(g, e)$	$\bar{2}$	$\bar{4}$	$\bar{3}$	$\bar{1}$

So if we know that Alice sends out $\bar{3}$ and Bob sends out $\bar{4}$ as their public values, the evil eavesdropper immediately knows that their private values are 3 and 2 respectively.

Also, the generator g has to be decided, as the group $(\mathbf{Z}_p^\star, \cdot)$ might have more than one generator. Because g is a primitive root modulo p, no information is given beforehand on the values $g^a \bmod p$ and $g^b \bmod p$ might have. Furthermore, there is no need to choose a and b bigger than $p - 2$ as we know that $g^{p-1} = 1 \bmod p$, and hence, there always exists $a' : 0 \leq a' < p-1$ such that $g^a \bmod p = g^{a'} \bmod p$.

2.2 Elliptic Curve Diffie-Hellman Key Exchange

How does moving to elliptic curves change the setup we have so far? The use of the word "curve" sets our minds to think about the nice pictures we used to illustrate functions at school. That is, we have an equation, and when we draw the points that satisfy the given equation in the xy-plane, we get a nice picture. In elliptic curve set-up, the equation could look like this:

$$y^2 = x^3 - x + 1$$

and the corresponding picture as in Figure 2.1.

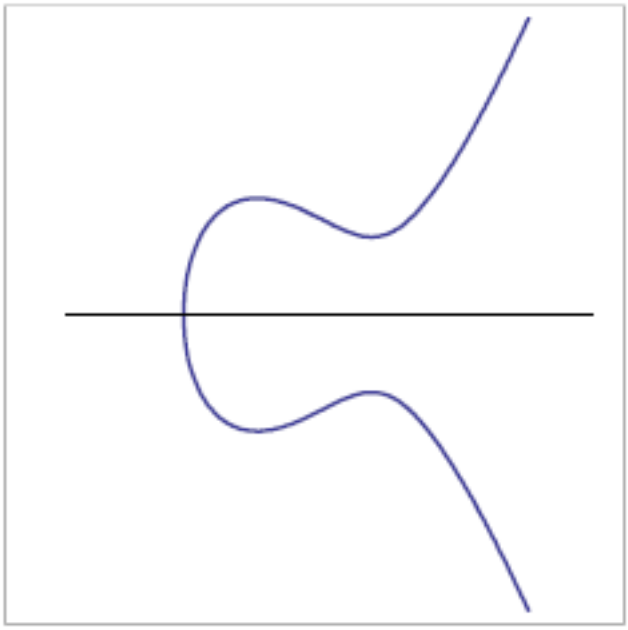

Figure 2.1: An elliptic curve on real numbers.

That is, if we were in the world of real numbers, but we are not. Computers rather operate in discrete world and with a finite number of elements, which means that the pictures do not look that nice anymore.

Regardless of the pictures, what we need is a suitable *group* that is somehow based on elliptic curves. This turns out to be possible. Remember that a group is just *a set of points together with an operation that satisfies certain conditions.* We just take the points (x, y) that satisfy the equation (modulo p), add a point to play the role of the neutral element, and then define an operation that satisfies the group axioms on this set of points.

Definition 7 Let $p > 3$ be a prime and let $a, b \in \mathbf{Z}_p$ be constants such that $4a^3 + 27b^2 \not\equiv 0 \bmod p$. The *elliptic curve* $E_{a,b}$ over $\mathbf{Z}_p$ is the set

$$E_{a,b} = \{(x, y) \in \mathbf{Z}_p \mid y^2 \equiv x^3 + ax + b \bmod p\} \cup \{\bigcirc\} \tag{2.1}$$

(see [35, Definition 6.4, p. 258]).

Remark 5 Different notation is used in different books for the extra point $\bigcirc$ that was added to $E_{a,b}$, for example, Hankerson [14, p. 13] names it as the point at infinity and denotes it by ∞.

We later define a group operation on set $E_{a,b}$, but let us start by an example of what kind of a set we are dealing with. In the context of elliptic curves, we omit the bars above equivalence classes, and assume that the reader knows from the context when a number actually refers to the equivalence class that it defines. Calculations are done in the underlying field $\mathbf{Z}_p$, which means that $+$ refers to addition modulo p and, similarly, multiplications are done modulo p.

Example 2 Let us look at the solutions of equation $y^2 = x^3 - x + 1$ on $\mathbf{Z}_7$. Looking up the solution from the table

x	$x^3 - x + 1 \bmod 7$	y s.t. $y^2 \equiv x^3 - x + 1 \bmod 7$
0	1	$1, 6$
1	1	$1, 6$
2	0	0
3	4	$2, 5$
4	5	$-$
5	2	$3, 4$
6	1	$1, 6$

gives us

$$E_{-1,1} = \{(0,1),(0,6),(1,1),(1,6),(2,0),(3,2),(3,5),(5,3),(5,4),(6,1),(6,6),\bigcirc\}.$$

One thing to notice is the number of points on the set $E_{a,b}$. In the case of our example there are 12 points. In general, this size is outlined by Hasse's theorem

$$|E_{a,b}| \leq p + 1 - t, \tag{2.2}$$

where $|t| \leq 2\sqrt{p}$. In Equation 2.2 value t is called the *trace of Frobenius*. Exact value for $|E_{a,b}|$ can be computed, for example, with Schoof's algorithm [30]. Another thing to note here is that all values $x \in \mathbf{Z}_p$ do not have the corresponding $y \in \mathbf{Z}_p$ that would satisfy the equation. In real life this implies that padding might be needed when numbers are converted to points on the curve, for example, for encryption. This happens on average 50 % of the times as can be concluded from Hasse's theorem. There are $p-1$ non-zero elements in $\mathbf{Z}_p$ and $p-t$ finite points on the curve. Almost all points have non-zero y, which implies that two values correspond to one x. Hence we get that approximately half of the times $x \in \mathbf{Z}_p$ does not have corresponding y that would satisfy the equation that specifies the curve, that is, x does not convert to a point on the curve.
Now we are ready to define an addition operation $\oplus$ on $E_{a,b}$.

Definition 8 Let $E_{a,b}$ be an elliptic curve over $\mathbf{Z}_p$ and let $P_0 = (x_0, y_0), P_1 = (x_1, y_1)$ be points on $E_{a,b}$. We define the addition operation $\oplus : E_{a,b} \times E_{a,b} \rightarrow E_{a,b}$ as follows:

1. $\forall P \in E_{a,b} : P \oplus \bigcirc = P$.

2. If $x_0 = x_1$ and $y_0 = -y_1$, P_1 is called the inverse of P_0 and $P_0 \oplus P_1 = \bigcirc$.

3. Otherwise, $P_0 \oplus P_1 = P_2 = (x_2, y_2)$, where

$$\begin{aligned} x_2 &= \lambda^2 - x_0 - x_1 \\ y_2 &= \lambda(x_0 - x_2) - y_0, \end{aligned}$$

and

$$\lambda = \begin{cases} (3x_0^2 + a)(2y_0)^{-1}, & \text{if } P_0 = P_1 \\ (y_1 - y_0)(x_1 - x_0)^{-1}, & \text{if } P_0 \neq P_1 \end{cases}$$

(see [35, p. 258]). Note that the arithmetic operations above are naturally performed in $\mathbf{Z}_p$. It is also worth noting that λ is well-defined (used inverses exist) as item 2 also covers the case, where $y_0 = 0$ and if $x_0 = x_1$, either item 2 is applicable or $y_0 = y_1$, in which case $P_0 = P_1$.

By combining the two previous definitions we obtain a group as we wanted.

Theorem 3 *Let $E_{a,b}$ be an elliptic curve over $\mathbf{Z}_p$ and let $\oplus : E_{a,b} \times E_{a,b} \to E_{a,b}$ be an operation on $E_{a,b}$ as defined in Definition 8. Then the structure $(E_{a,b}, \oplus)$ is an Abelian group.*

Note that whereas $(\mathbf{Z}_m^\star, \cdot)$ is a multiplicative group, the group $(E_{a,b}, \oplus)$ is additive. This implies that the operation corresponding to exponentiation, that is, applying the group operator repeatedly to a group's element, is *scalar multiplication*. This means that if we look at Algorithm 1 in the context of elliptic curves, base will be a point $P \in E_{a,b}$, the secret values are still random integers a, b, but instead of g^a and g^b we get aP and bP.

Remark 6 Group $(E_{a,b}, \oplus)$ is not always cyclic, but if $|E_{a,b}|$ is prime or product of two distinct primes, $(E_{a,b}, \oplus)$ is cyclic, that is,

$$\exists P \in E_{a,b} : E_{a,b} = \{iP \mid i \in \{0, \ldots, |E_{a,b}| - 1\}\},$$

where $0P = \bigcirc$. In general, $(E_{a,b}, \oplus)$ is isomorphic to $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2}$, for some positive integers n_1, n_2 such that $n_2 | n_1$ and $n_2 | (p-1)$. This implies that even if $(E_{a,b}, \oplus)$ is not cyclic, it always has a cyclic subgroup that can potentially be used as a setup.

3 Discrete Logarithm Problem

For Diffie-Hellman Key Exchange to act reasonably in cryptographic context, a possible eavesdropper should not be able to deduce the secret information from the data going through the public channels. When using classical Diffie-Hellman Key Exchange it should be hard to solve k from the equation

$$g^k \equiv c \bmod p.$$

This is leads to the discrete logarithm problem. There are, of course, infinitely many solutions, but typically we choose the least nonnegative value.

Definition 9 (The Discrete Logarithm Problem (DLP)) Let p be a prime number and g a generator of $\mathbf{Z}_p^\star$. Given an element $c \in \mathbf{Z}_p^\star$, find the integer k such that $0 \leq k < p-1$ and $g^k \equiv c \bmod p$. [20, Definition 3.51, p. 103]

Solution to the discrete logarithm problem gives us a discrete logarithm.

Definition 10 Let p be a prime number and g a generator of $\mathbf{Z}_p^\star$. The *discrete logarithm of* $c \in \mathbf{Z}_p^\star$ *to the base* g, denoted by $\log_g c$, is the unique integer k such that $0 \leq k < p-1$ and $g^k \equiv c \bmod p$.

Remark 7 As $g^k \equiv g^{(k+l\cdot \mathrm{ord}(g))}$ for any $k \in \mathbf{Z}$, discrete logarithm is not unique, if we did not require that $0 \leq k < p-1$.

Remark 8 Familiar properties of ordinary logarithms hold also for discrete logarithms modulo $\mathrm{ord}(g) = p-1$, that is,

$$\log_g(xy) = \log_g(x) + \log_g(y) \bmod \mathrm{ord}(g)$$

and

$$\log_g(x^y) = y \log_g(x) \bmod \mathrm{ord}(g).$$

Note that there are also a lot of related problems [8] around the topic. One example is (computational) Diffie-Hellman Problem (DHP), which asks to compute

$g^{ab} \bmod p$ when $g^a \bmod p$ and $g^b \bmod p$ are given. It is obvious that solving DLP would also solve DHP, but it is not known if the converse holds in general.

Chris Studholme [36] gives a quite detailed explanation on algorithms for solving the discrete logarithm problem. I will leave the details to him (and to [20]) and give an overview of some of the basic ideas here.

The most obvious algorithm for solving the discrete logarithm problem is exhaustive search, sometimes called *trial multiplication*. To solve k from $c = g^k$, one could start multiplying g by itself until the result is equal to c and the number of required multiplications $+1$ would give out the k. In general this is, of course, quite inefficient, as it would require on average $\mathrm{ord}(g)/2$ group operations, but if $\mathrm{ord}(g)$ is known to be small, this might lead to a potential threat.

In 1971 Daniel Shanks [31] presented a better algorithm for solving the discrete logarithm problem. The reasoning behind this *Baby-Step-Giant-Step* algorithm is that if $c = g^x$ and $\mathrm{ord}(g) = n$, we can present x in form $i \cdot m + j$, where $m = \lceil \sqrt{n} \rceil$ and $i, j \in \{0, \ldots, m-1\}$. Thus, when we find two numbers $c * g^{-im}$ and g^j such that

$$c * g^{-im} = g^j,$$

we know that $c = g^{im+j}$, and hence $\log_g c = im + j$. Along this line the algorithm first produces and stores the baby-steps (j, g^j) for $j = 0, \ldots, (m-1)$. Then it proceeds with the computation of giant-steps (i, cg^{-im}), for $i = 0, \ldots, (m-1)$ and subsequently checks if a match is found from the table of baby steps. This method runs in $O(\sqrt{n})$ time, but the problem is that it also requires $O(\sqrt{n})$ space. [20, Section 3.6.2]

One solution to the large space requirements introduced by Baby-Step-Giant-Step algorithm is to use *Pollard's Rho* algorithm [26]. The idea of this algorithm is to define a deterministic random sequence $a_0, a_1, a_2, \ldots$ of group elements (see, for example, [36, p. 7]). As we are dealing with finite groups, a collision will eventually happen, and this is very likely to lead to a solution of the logarithm. The name of the algorithm comes from the fact that after the collision, the sequence will repeat itself in a cycle, which gives a ρ-like shape. The so-called "birthday problem" dictates that, if the sequence is truly random, a collision is

likely (with a probability higher than 0.5) to happen with a number of steps that asymptotically approaches $\sqrt{\pi n/2}$ as n increases [20, Fact 2.27, p. 53]. As such, this approach would require $O(\sqrt{n})$ space, but with the help of Floyd's cycle-finding algorithm, the space requirement can be dropped down to storing only two elements until elements such that $a_i = a_{2i}$ are found. Assuming that the defined sequence behaves like a random sequence, the expected running time of this algorithm is $O(\sqrt{n})$ group operations [20, Fact 3.62, p. 107].
The fastest currently known algorithms for generic cyclic groups are running in $O(\sqrt{n})$ time. It has actually been shown that these "square root methods" are the best that can be expected without any more information about the group structure [32]. A possible extra information about the group structure is factoring of its order, especially when the order is smooth.

Definition 11 Let $B \in \mathbf{N}$ and $F(B) = \{q : q \text{ is prime and } q \leq B\}$. An integer n is $F(B)$-*smooth*, if all its prime factors q belong to $F(B)$, that is,

$$n = \Pi_{q \in F(B)} q^{e_q},$$

where $e_q \geq 0$. $F(B)$ is called a *factor base*. [15]

The idea behind the *Pohlig-Hellman* algorithm is that if we have the prime factorization $p_1^{e_1} \cdots p_r^{e_r}$ of the group of order n, we can use the Chinese remainder theorem to calculate discrete logarithm for the whole group from discrete logarithms in groups of order $p_i^{e_i}$. Given the factorization of the group order, running the Pohlig-Hellman algorithm will take $O(\Sigma_{i=1}^{r} e_i(\lg n + \sqrt{p_i}))$ group multiplications [20, Fact 3.65, p. 108]. If the group order is known to be $F(B)$-smooth, for some small B, this is severe. Due to this, *the order of the group should have at least one large prime factor.*
When a significant proportion of group elements can be efficiently expressed as a product of some selected factor base $F(B)$, it is possible to use sub-exponential time algorithm called *index calculus*. Its basic idea is to first solve discrete logarithms $log_g q$ for all elements in the factor base and then to use this information together with the basic properties of discrete logarithms to solve the actual discrete logarithm. If we find an integer k such that cg^k can be presented as a

product of elements in $F(B)$, that is,

$$cg^k = \Pi_{q \in F(B)} q^{e_q},$$

it follows from the properties stated in Note 8 that

$$log_g c = \Sigma_{q \in F(B)} e_q \log_g q - k.$$

Similarly, if we present enough $F(B)$-smooth elements of the group as a product of elements in $F(B)$, we get a bunch of linear equations and can solve unknown discrete logarithms $\log_g q$ by using methods of linear algebra.

Remark 9 Finding a suitable factor base for index calculus is an optimization problem [20, Note 3.71, p. 112] – the more elements you have in the factor base, the more equations you will have to solve. On the other hand, the bigger the factor base is, the more likely it is that a randomly chosen element of a group can be presented as a product of the elements of the factor base. Solving these linear equations is the most time consuming part of index calculus, but luckily, it can be done using parallel computers, and it only needs to be done once per group [20, Note 3.73].

The best algorithm for computing discrete logarithms in $\mathbf{Z}_p^\star$ is a variation of the index calculus method called the *number field sieve* [20, Note 3.72]. It has expected running time of $L_p[\frac{1}{3}; \frac{63}{9}^{1/3} + o(1)]$, where $L_p[s; c] = e^{c \log p)^s (\log \log p)^{1-s}}$ [36, p. 44]. The latest records can be checked from Wikipedia [9], but in 2007 the record was to compute discrete logarithms over $\mathbf{Z}_p^\star$, where p is a 530-bit safe prime[1].
So far, we have only paid attention to group $\mathbf{Z}_p^\star$, but in fact many of the methods above are also valid in a more generic setting and also for elliptic curves.

Definition 12 (The Generalized Discrete Logarithm Problem) Let G be a finite cyclic group of order n and g a generator of G. Given an element $c \in G$, find the integer k such that $0 \leq k < n$ and $g^k = c$. [20, Definition 3.52, p. 103]

[1] Safe prime is of the form $2p + 1$, where also p is a prime

Solutions to the generalized discrete logarithm problem give discrete logarithms over arbitrary finite cyclic groups. As elliptic curves form an additive (rather than multiplicative) group, it is good to rewrite this definition with additive notation for elliptic curves.

Definition 13 (The Elliptic Curve Discrete Logarithm Problem) Let $E_{a,b}$ be an elliptic curve over $\mathbf{Z}_p$ and let $\oplus : E_{a,b} \times E_{a,b} \rightarrow E_{a,b}$ be an operation on $E_{a,b}$ as defined in Definition 8. Let $Q \in E_{a,b}$ be a generator of a subgroup of $(E_{a,b}, \oplus)$ of order n. Given an element $P \in E_{a,b}$, find the integer k such that $0 \leq k < n$ and $kQ = P$.

Remark 10 In IEEE Std-1363-2000 [16] it is assumed that the order n of G is a prime and n^2 does not divide the order of the curve. These assumptions make sense, because they guarantee that the cyclic subgroup generated by G does not contain small cyclic subgroups and is big enough.

Solutions to the elliptic curve discrete logarithm problem give us discrete logarithms over elliptic curves.

Definition 14 Let G be a finite cyclic subgroup of $(E_{a,b}, \oplus)$ and Q a generator of G. The *discrete logarithm of $P \in G$ to the base Q*, denoted by $\log_Q P$, is the unique integer k such that $0 \leq k < n$ and $kQ = P$, where $n = \mathrm{ord}(G)$.

Most algorithms introduced here for solving discrete logarithms over $\mathbf{Z}_p^\star$ also apply in a general setting and, hence, are applicable for solving discrete logarithms over elliptic curves. Only index calculus is not directly applicable to the elliptic curve setting, which is a good thing, as sub-exponential time algorithm with short key lengths used for elliptic curves would be devastating. Nevertheless, Menezes et al. [19] have shown that for supersingular elliptic curves it is possible to reduce in polynomial time the elliptic curve discrete logarithm problem to the discrete logarithm problem that can be solved using index calculus. Another class of weak elliptic curves was pointed out by Smart [33], when he showed that the elliptic curve discrete logarithm problem can be solved in linear time, provided that the

elliptic curve $E_{a,b}$ over $\mathbf{Z}_p$ has exactly p points, that is, its trace of Frobenius equals one.
Here there is a summary of classes of weak elliptic curves for which the elliptic curve discrete logarithm problem is known to be solvable in such a short time that these curves should *not* be used as a setup for cryptographic system:

- Supersingular curves, that is, curves for which the trace of Frobenius is zero.
- Elliptic curves over $\mathbf{Z}_p$ having exactly p points, that is, curves for which the trace of Frobenius is one.
- Elliptic curves that do not have a subgroup with order that is divisible by at least one large prime factor (Pohlig-Hellman algorithm).

For a more detailed listing, see [17, p. 28–32]. If you avoid these classes of weak curves, you should be safe at least until some new ideas or faster computers come around. In 2009 the record was solving the elliptic curve discrete logarithm problem for an elliptic curve over $\mathbf{Z}_p^\star$, where p was a 112-bit prime. This was done by using common parallelized version of Pollard rho method on over 200 Playstation 3 consoles and it still took about 6 months [5].

4 Diffie-Hellman Key Exchange

Let us assume that the parties have decided on parameters, that is, prime modulus p and generator g of $(\mathbf{Z}_p^\star, \cdot)$. Then Diffie-Hellman Key Exchange for one party looks like this:

Algorithm 2

1. Generate a random number $0 < a < p - 1$.

2. Compute $g^a \bmod p$.

3. Compute $S_2^a \bmod p$, where S_2 was received from the other party.

Note that this is just a very basic approach to elaborate the ideas. More variation and details can be found from the standards ([2], [25]).
As always, when approaching actual implementation and using of a cryptographic protocol, we should pay attention to security issues. The first weakness of Diffie-Hellman is, of course that, as a protocol, it is vulnerable to the man-in-the-middle attack. In the Diffie-Hellman Key Exchange context this means that a third party P would get g^a and g^b, but would send g^c, where c a random generated by P, to the original parties. This way P gets shared secrets g^{ac} and g^{bc} and can decrypt, read and again encrypt for the receiving party all content that is send between the original parties. So, some extra authentication should take place, when man-in-the-middle is a possibility. Another possible weakness might come from bad random number generation. As real random numbers are hard to get, pseudo-random numbers are generally used. In this case it should be verified that the used pseudo-random generator is good enough [10, Chapter 10].
From mathematical point of view, issues might arise from parameter generation. If g fails to be primitive root but rather generates a small subgroup, solving discrete logarithm might not be that big of a problem after all. The same applies if g^a or S_2 happen to fall into a small subgroup. This can be avoided quite easily by using safe primes, but then again it might lead to losses in efficiency. [10, p. 213-215]

Efficient cryptography is often about finding the right balance between security and optimization. The obvious optimization in Diffie-Hellman protocol is to use smaller exponents. This leads to obvious security consideration, because if the exponent is known to be tiny, it is quite easy to find it by consequent multiplication of g by itself.
Using Wiener's table [24, Figure 4] is one option for optimization. This raises some questions, especially as using the same table in ElGamal signatures leads to full exposure of the private key from just one signature [24]. Nevertheless, Practical Cryptography [10, p. 218–] recommends using a subgroup whose size is a 256-bit prime q. This saves a lot of effort. It also emphasizes that given parameters should be checked at least once (and several times if there is a possibility for them to change) and that value S_2 received from the opponent actually is in the used subgroup: $1 < S_2 < p$ and $S_2^q = 1 \bmod p$. This thesis concentrates more on algorithms, but if you are actually implementing this protocol, you should definitely read through chapters 12 and 15 from [10].

4.1 Algorithms and their complexity

After choosing how big exponents we should use, it all boils down to how fast implementation of exponentiation g^a modulo p we can make. Standard text book algorithm for doing this would be by $a - 1$ consecutive multiplication modulo p. Even if you use a bit cleverer algorithm, like the method of Russian peasants, you still end up with $2 \times \log_2 a$ multiplication [29, p. 18] (and reduction modulo p as you would not want the arguments of multiplication to keep growing).
Bosselaers et al. [6] compare three different modular reduction algorithms: classical algorithm, Barrett's algorithm and Montgomery's algorithm and summarize [6, Table 1] their results of reducing a $2k$-digit number modulo a k-digit modulus. It shows that reduction part is at least slightly more demanding than multiplication and the Montgomery reduction wins the competition at least in the case where some pre-/after calculations are ignored and arguments are twice the size of the modulus, which is often close to the truth when implementing Public Key Cryptography and reducing 2k-bit numbers modulo k-bit number. So, when it

comes to exponentiation, cutting down the number of needed reductions is a good thing. This is exactly what Montgomery based exponentiation allows us to do. Let us see how this Montgomery exponentiation works in practice. Basic components of the Montgomery exponentiation are the Montgomery reduction and the Montgomery multiplication.

Definition 15 Let R be an integer greater than modulus m and $gcd(R, m) = 1$. Then the *Montgomery representation* of $x \in [0, m-1]$ is

$$[x] = xR \bmod m,$$

and the *Montgomery reduction* of $u \in [0, Rm-1]$ is

$$M_{\text{red}}(u, m) = uR^{-1} \bmod m.$$

[7, Definition 10.21, p. 180]

As our modulus tends to be a prime, limiting values of R to coprimes of m is not a real limitation. The catch here is to choose R in such a way that the Montgomery reduction becomes efficient. The way to do this is to choose R to be b^k, where b is the base for representing m, that is,

$$m = (m_{k-1} \cdots m_0)_b = \sum_{i=0}^{k-1} m_i b^i,$$

and k is the length of the b base representation of m. As m can be assumed to be odd, we can choose b to be a power of two in which case R will also be a power of two.

Example 3 Let $b = 2^{16}$ and $m = 12345678901234567890$. Now

$$m = 43860 * b^3 + 43404 * b^2 + 60191 * b + 2770$$

and hence the b base (or radix b) representation of m is $(43860, 43404, 60191, 2770)_b$ and its length is four.

From Definition 15 we know how the result of the Montgomery reduction looks like. Algorithm 3 shows how to compute it.

Algorithm 3 (Montgomery Reduction) Let $M = (m_{k-1} \cdots m_0)_b$ be a modulus in such base b that $gcd(M, b) = 1$, and let $X = (x_{2k-1} \cdots x_0)_b$ be an integer such that $X < M \cdot b^k$.

1. Let $R := b^k$, $M' := -M^{-1} \bmod b = -m_0^{-1} \bmod b$
2. Set $T = (t_{2k-1} \cdots t_0)_b := X$.
3. **for** $i := 0$ **to** $k - 1$ **do**
   - 3.1 $u_i := t_i M' \bmod b$
   - 3.2 $T := T + u_i M b^i$
4. $T := T/R$
5. **if** $T \geq M$, **then** $T := T - M$
6. **return** $M_{\text{red}}(X, M, M') := T = XR^{-1} \bmod M$

[7, Algorithm 10.22, p. 181].

If modulus M is clear from the context, we can write $M_{\text{red}}(X) := M_{\text{red}}(X, M, M')$. How does this algorithm work? The loop in item 3, is the key part here [20, Note 14.33, p. 602]. It calculates

$$T = X + UM,$$

where $U = (u_{k-1} \ldots u_0)_b$. The trick is to use the inverse of $M \bmod b$ in calculating each u_i. This will make sure that the i^{th} digit of T after the i^{th} round of the loop is $t_i + u_i M \bmod b = t_i + t_i M' M = t_i - t_i = 0$. Furthermore, as the loop performs addition of b-base numbers starting from the least significant number, it does not modify any digit of T that has index smaller than i. So, in the end of the loop there are k zeros, which makes division by b^k in item 4 a simple "drop some

zeros from the end"-operation. As $gcd(M, b) = 1$, also $gcd(M, R) = 1$ and, hence, $R^{-1} \bmod M$ exists. Now $T = (X + UM)/R = XR^{-1} + UR^{-1}M \equiv_M XR^{-1}$. Also, as we assumed that $0 \leq X < M \cdot b^k = MR$ and know that $0 \leq U = \sum_{i=0}^{k-1} u_i b^i \leq \sum_{i=0}^{k-1}(b-1)b^i =^1 b^k - 1 < b^k = R$, we have $T = (X + UM)/R < (MR + RM)/R = 2M$. So, if $T \geq M$, it is enough to subtract M from T only once and the result will be in $[0, N - 1]$.

Now that we know that Algorithm 3 works, that is, computes the Montgomery reduction as expected, we can start analyzing its complexity. Let us do this step-by-step using the same numeration as in the algorithm:

1. These values can be precomputed for each modulus.

2. $2k$ substitutions.

3. Loop is executed k times.

   3.1 One single precision multiplication and possibly double precision reduction mod b.

   3.2 k single precision multiplications[2] and additions.

4. Dropping k zeros from the end (time consumption depends on the chunk order of the number representation – might require some shifting).

5. Comparison of at most k single precision integers and possible up to k subtractions.

Multiplication is more demanding than addition or subtraction, not to mention substitutions or possible shifting. Even reduction modulo b is easy here as we are handling the numbers in base b – hence, it is enough just to drop the overhead. So altogether, from the complexity perspective we are left with $k(k + 1)$ single precision multiplications [6].

---

[1] Geometric sum

[2] There might be some hidden additions here, if the multiplied numbers are close to the limit b.

Algorithm 4 (Montgomery Multiplication) Let $M = (m_{k-1} \cdots m_0)_b$ be a modulus in such base b that $gcd(M, b) = 1$, and let $X = (x_{k-1} \cdots x_0)_b$ and $Y = (y_{k-1} \cdots y_0)_b$ be integers such that $0 \leq X, Y < M$.

1. Let $R := b^k$, $M' := -M^{-1} \bmod b = -m_0^{-1} \bmod b$

2. Set $T = (t_k t_{k-1} \cdots t_0)_b := 0$.

3. **for** $i := 0$ **to** $k - 1$ **do**

   3.1 $u_i := (t_0 + x_i y_0) M' \bmod b$

   3.2 $T := (T + x_i Y + u_i M)/b$

4. **if** $T \geq M$, **then** $T := T - M$

5. **return** $M_{\text{mult}}(X, Y, M, M') := XYR^{-1} \bmod M$

[20, Algorithm 14.36, p. 602].

If modulus M is clear from the context, we can simply write $M_{\text{mult}}(X, Y) := M_{\text{red}}(X, Y, M, M')$. It is easy to note that $M_{\text{red}}(X) = M_{\text{mult}}(X, 1)$.
The basis for this algorithm is again in loop 3, where Y is multiplied by the i^{th} digit of X and added to zero-initialized T. Furthermore, in each round we want to divide the result by b so that in the end of the loop we have completed the division by $R = b^k$. For this repeated division to be possible and even easy, we act as in Algorithm 3, that is, add on each round also a balancing factor $u_i M$. As we are interested in the result only up to modulus M, this balancing factor does not affect the result. In addition, it can be shown by easy induction on i that after each round $0 \leq T < 2M - 1$ (see [20, Note 14.37, p. 603]). So, step 4 is also justified and we have shown that Algorithm 4 computes what it claims to compute.
Let us then check the efficiency of Algorithm 4 step-by-step using the same numeration as in the algorithm:

1. These values can be precomputed for each modulus.

2. Zero-initialization of length $n + 1$ (in base b).

3. Loop is executed k times.

   3.1 Two single precision multiplications, one single precision addition and possibly three (counting intermediate steps – each intermediate step is reduced to keep the actual calculations single precision) double precision reduction mod b.

   3.2 $2k$ single precision multiplications[3] and additions. Dropping one zero (dividing by b) from the end (hardness depends on the chunk order of the number representation – might require some shifting).

4. Dropping k zeros from the end (hardness depends on the chunk order of the number representation – might require some shifting).

5. Comparison of at most k single precision integers and possible up to k subtractions.

If we again count only multiplications, we end up with $k(2 + 2k) = 2k(k + 1)$ single precision multiplications as indicated in the literature.

Algorithm 5 (Montgomery Exponentiation) Let $N = (n_{k-1} \cdots n_0)_b$ be a modulus in such base b that $gcd(N, b) = 1$, let $e = (e_j \cdots e_0)_2$ be the exponent and further, let x be an integer such that $1 \leq x < M$.

1. Set $R := b^k$ and $N' := -N^{-1} \bmod b = -n_0^{-1} \bmod b$.

2. Calculate $[x] = x * R \bmod N$ and $T = R \bmod N$.

[3] There might be some hidden additions here, if the multiplied numbers are close to the limit b.

3. **for** $i := j;\ i \geq 0;\ i--$ **do**

   3.1 $T := M_{\text{mult}}(T, T, N, N') = T * T * R^{-1} \bmod N$

   3.2 **if** $e_i == 1$, **then** $T := M_{\text{mult}}(T, [x], N, N') = T * [x] * R^{-1} \bmod N = T * x \bmod N$

4. **return** $M_{\text{exp}}(x, e, N) := M_{\text{red}}(T, N, N') = x^e \bmod N$.

[20, Algorithm 14.94, p. 620].

This algorithm combines left-to-right binary exponentiation (see for example [20, Algorithm 14.79, p. 615]) with the Montgomery multiplication. The idea of left-to-right binary exponentiation can be explained with the following equation:

$$x^e = x^{(e_j 2^j + \cdots + e_1 2 + e_0)} = x^{e_j 2^j} \cdots x^{e_1 2} x^{e_0} = (x^{2^j})^{e_j} (x^{2^{j-1}})^{e_{j-1}} \cdots (x^2)^{e_1} x^{e_0}.$$

So, if $e_i = 1$, we multiply x into the loop and the rest of the loop raises it to the power 2^i. As $(x^{2^a} x^{2^b})^2 = (x^{2^a})^2 (x^{2^b})^2$, further multiplications do not mix the previous ones and the final result is as expected. Algorithm 5 adds usage of the Montgomery multiplication to this. As we have already shown how the Montgomery multiplication works, it is easy to see that after each round we have the required power of x multiplied by R. So, in the end we just do one Montgomery reduction and have the result.
Rough complexity analysis of Algorithm 5 looks like this (again the numbers refer to the corresponding steps in the algorithm):

1. These values can be precomputed for each modulus.

2. One multiprecision multiplication modulo N and one reduction modulo N. T can be precomputed for each modulus. If also $R^2 \bmod N$ is precomputed, $x * R = M_{\text{mult}}(x, R^2, N, N')$ might be useful.

3. Loop is executed $j + 1$ times.

   3.1 One Montgomery multiplication, that is, $2k(k + 1)$ single precision multiplications.

   3.2 Possibly one Montgomery multiplication, that is, $2k(k + 1)$ single precision multiplications.

4. One Montgomery reduction, that is, $k(k+1)$ single precision multiplications.

4.2 Testing the performance of one implementation

The cryptographic library[4] (CL), that was used for testing, provides interface for the Montgomery multiplication (Algorithm 4) and the Montgomery exponentiation (Algorithm 5) and implements them in plain C. A natural continuation for theoretical evaluations on the complexity of these algorithms was to test how well given implementations (in CL) perform in practice when compared to each other and to the expected running times.
Basically, adding CPU time measurement around operation is very simple:

```
#include <time.h>

clock_t t;
double time;

t = clock();

<operation>

time = ((double)(clock() - t))/CLOCKS_PER_SEC;
```

[4] Implementation of the cryptographic library is done by S. Sovio [34]

(see: http://www.gnu.org/s/libc/manual/html_node/CPU-Time.html, link tested 01-Dec-2011). Based on my experience, this seems to be rather set-up sensitive, meaning that it does not give proper result in all environments. Results can also be affected by other processes running in the testing environment. In order to have control on the possible other processes, I decided to try out my own laptop. This first try with gcc-cygwin-setup on my laptop was a total failure – running times with this setup could give out anything from zero to even negative values.

The second option was to use our teams normal testing environment, where tests are written in C and run on Armulator (ARM Realview Debugger simulating ARM1176 Processor). On this Armulator setup, some nesting/multiple test cases caused zero results. Eventually, by using quite modest test cases, I managed to get somewhat reasonable test results, but closer analysis showed that those results could not hold either. So, in the end I ported my own tests from the Armulator setup to normal Linux environment using gcc-compiler, compiled the given cryptographic library into static library that could be linked on compile time to my tests and run the tests on Linux servers, knowing that other processes running there could mess up with the measurements. On the other hand, running tests on Linux servers is much faster than on Armulator. So, if only the relative times are of interest, the likelihood of some new process messing up the measurements that only last some minutes, is low enough.

By analyzing the algorithms for the Montgomery multiplication and the Montgomery exponentiation, one gets respectively $2k(k+1)$ and (worst-case) $4k(k+1)(j+1)+k(k+1)$ single precision multiplications, where k is length of the b-base representation of the used modulus and j is the length of the binary representation of the used modulus. To test out these limits I made a couple of new test cases to our teams existing testing environment, ported them to gcc-environment and adopted the usage of standard library (time.h) function *clock()* to measure the time consumption.

As generally happens in testing, the hardest part is deciding what should be tested and finding suitable test vectors for it. Ideal testing in this case would probably be generating a statistically significant amount of random test vectors of suitable

length and then taking the average timings. As running tests on Armulator is reasonably slow, and extracting random vector generation times from the times I want to measure might be problematic, I decided to start with repeated operations with constant test vectors, knowing that this is not statistically valid method. As I had built this setup already, I continued with this approach even though running tests on Linux server would give an opportunity to run more comprehensive test sets.

For test moduli I decided to use primes that can be found from RFCs. RFC 2409 (http://www.ietf.org/rfc/rfc2409.txt, section 6.2, link tested 01-Dec-2011) gives 1024-bit prime M_1, which has hexadecimal value

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE65381
FFFFFFFF FFFFFFFF
```

and RFC 3526 (http://www.ietf.org/rfc/rfc3526.txt, section 3, link tested 01-Dec-2011) gives 2048-bit prime M_2, which has hexadecimal value

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B
E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510
15728E5A 8AACAA68 FFFFFFFF FFFFFFFF.
```

The Montgomery multiplication requires that X and Y are smaller than the modulus used. Hence, a quite natural selection for values to be multiplied was $X_1 = M_1 - 1, Y_1 = M_1 - 2$ in 1024-bit case and $X_2 = M_2 - 1, Y_2 = M_2 - 2$ in 2048-bit case. Numbers in the given cryptographic library are presented in $b = 2^{16}$-base. This way the length of the b-base representation of a 1024-bit modulus is 64 and the length of the b-base representation of a 2048-bit modulus is 128. The expected running time of the Montgomery multiplication for a 1024-bit modulus would be something around $2k(k+1) = 2 * 64(64+1) = 8320$ single precision multiplications and for a 2048-bit modulus around $2*128(128+1) = 33024$ single precision multiplications.

To start with, I wrote a simple loop to measure how long $PC*T_k = PC*2*k(k+1)$ single precision multiplications take, where PC is the performance constant used to get non-zero times. The faster the environment is, the bigger this constant needs to be. For test values I chose the maximal unsigned 16-bit value, that is, I multiplied $(2^{16} - 1)$ with itself using the following code.

```
uint32 i, j;
clock_t t;
uint32 res;
uint16 x=0xffff,y=0xffff;
double time;

t = clock();

for (j = 0; j < PERF_CONST; j++)
{
    for (i = 0; i<Tk; i++)
      res = x*y;
}

time = ((double)(clock() - t))/CLOCKS_PER_SEC;
```

```
<print out time>
```

Similarly, I measured how much CPU time $M_{\text{mult}}(X_i, Y_i, M_i, M_i')$, where $i \in \{1, 2\}$, consumed, when performed PC times. For testing the Montgomery exponentiation I reused the same values and I measured CPU time for PC2 exponentiations $M_{\text{exp}}(X_i, Y_i, M_i) = X_i^{Y_i} \bmod M_i$. As a result I got the following table

k	$f(k)$	$F(k)$	$E(k)$
64	$1.39sec$	$3.03sec$	$4.97sec$
128	$5.56sec$	$12.03sec$	$36.96sec$,

where

- $f(k)$ gives the CPU time for performing multiplication $(2^{16} - 1) \times (2^{16} - 1)$ $PC * T_k = PC * 2 * k(k + 1)$ times,

- $F(k)$ gives the CPU time for performing PC Montgomery multiplications $M_{\text{mult}}(X_i, Y_i, M_i, M_i')$, where $i = 1$, if $k = 64$ and $i = 2$, if $k = 128$ and

- $E(k)$ gives the CPU time for performing PC2 Montgomery exponentiations $M_{\text{exp}}(X_i, Y_i, M_i)$, where $i = 1$, if $k = 64$ and $i = 2$, if $k = 128$,

where $PC = 100000$ and $PC2 = PC/1000 = 100$.
Let us now compare the achieved results to the expected running times. We know that $f(k)$ should equal to $F(k)$, as one k-byte Montgomery multiplication is supposed to use $2 * k(k + 1)$ single precision multiplications. In some runs, these columns matched quite nicely, but the latest run doubled the times for the Montgomery multiplication. Common sense indicates that $f(k)$ should be smaller than $F(k)$ as the Montgomery multiplication also requires some other operations than just multiplication, although multiplications are dominating the complexity analysis. And this is the case.
If we look at the worst case analysis of the Montgomery exponentiation $E(k)$ should equal to $4k(k + 1)(j + 1) + k(k + 1)$, where $j = 1024$, when $k = 64$ and $j = 2048$, when $k = 128$. This means that in the worst case scenario, the

Montgomery exponentiation would require about $2j$ Montgomery multiplications, but we see from the columns that this is not the case. Explanation for this is that we should look at the average complexity of exponentiation instead. We note that in the loop of the Algorithm 5, item 3.2 is only executed if $e_i = 1$. On average this happens half of the times. This means that instead of performing $2j$ Montgomery multiplications we actually need to perform only $1.5j$ Montgomery multiplications. So, on average we should have $E(k) = 1.5 * j * F(k)/1000$, which gives 4.65 seconds when $k = 64$ and 36.96 seconds when $k = 128$. Hence we can conclude that implementation of the Montgomery exponentiation in the given cryptographic library performs according to the expectations.

5 Elliptic Curve Diffie-Hellman Key Exchange

Let us assume that the parties have decided on parameters, that is, prime modulus p, coefficients $a, b \in \mathbf{Z}_p$ and generator P of $(E_{a,b}, \oplus)$ or a suitable subgroup of $(E_{a,b}, \oplus)$. Then the Elliptic Curve Diffie-Hellman Key Exchange for one party looks like this:

Algorithm 6

1. Generate a random number $0 < \alpha < n - 1$, where n is the size of the used (sub)group.

2. Compute αP.

3. Compute αP_2, where P_2 was received from the other party.

Note that this is just a very basic approach to elaborate the ideas. More variation and details can be found from the standards ([3], [25]).
As we saw in the previous chapter, the basic idea for exponentiation is to use square-and-multiply algorithm induced by the binary presentation of the exponent. Natural analogue here is to use double-and-add algorithm to compute scalar multiplication needed for Elliptic Curve Diffie-Hellman Key Exchange. By Definition 8, both doubling and addition require (modular) inversion which in practice tends to be significantly more demanding operation than (modular) multiplication. For example comparing our teams implementation of modular inversion and multiplication gives the table

k	$M(k)$	$I(k)$
256	$0.48sec$	$0.38sec$
1024	$6.93sec$	$3.92sec$

where

- $M(k)$ gives the CPU time for performing $PC = 100000$ modular multiplications on k-bit modulus and

- $I(k)$ gives the CPU time for performing $PC_3 = PC/100 = 1000$ modular inversions on k-bit modulus.

This motivates the usage of projective coordinates instead of traditional, so-called affine, coordinates that were introduced in Section 2.2.
In projective system points are represented by triples (X, Y, Z) (over the underlying field, which in our case is $\mathbf{Z}_p$). Projective points are actually equivalence classes

$$(X : Y : Z) = \{(\lambda^c X, \lambda^d Y, \lambda Z) \mid \lambda \in \mathbf{Z}_p^*\},$$

where c and d are positive integers. If $Z \neq 0$, then $(X/Z^c, Y/Z^d, 1)$ is a representative of the projective point $(X : Y : Z)$. Replacing x by X/Z^c and y by Y/Z^d and clearing the denominators gives the projective form for the elliptic curve equation. [14, p. 87]
Variation in weights c and d gives different kinds of projective coordinates. The weights $c = 2$ and $d = 3$ have been chosen to IEEE Std 1363-2000 Standard [16] as they provide (at least for that time being[1]) the fastest arithmetic on elliptic curves [16, p. 121]. This choice of weights gives so called Jacobian coordinates.

Example 4 (Jacobian coordinates) Let $Z \neq 0$. Substituting

$$x = \frac{X}{Z^2}, y = \frac{Y}{Z^3}$$

to elliptic curve equation $y^2 = x^3 + ax + b$ (and clearing the denominators) gives us projective form

$$Y^2 = X^3 + aXZ^4 + bZ^6$$

of elliptic curve equation. The neutral element $\bigcirc$ corresponds to $(1 : 1 : 0)$ and additive inverse of $(X : Y : Z)$ is naturally $(X : -Y : Z)$ (see [14, Example 3.20, p. 88]).

Note that it is wise to use these projective coordinates only internally even though this requires one extra division in the end to convert the coordinates back to

[1] For newer results, see for example [1]

affine coordinates. Firstly, because they require more space, which might be an issue in transmission, but more importantly, because external usage of projective coordinates leaks information [23].

5.1 Algorithms and their complexity

In this section we present algorithms given in [16] for point doubling, point addition and scalar multiplication in Jacobian coordinates. Let us start with the point doubling. The same substitution that was used in Example 4 to point doubling formula, that is in place of x_0, x_1 use X/Z^2 and in place of y_0, y_1 use Y/Z^3 when $P_1 = P_2 \neq -P_1$, in Definition 8 gives us (after some steps of making the formulas look nicer)

$$X_2' = \frac{(3X^2 + aZ^4)^2 - 8XY^2}{(2YZ)^2}$$

and

$$Y_2' = \frac{(3X^2 + aZ^4)(4XY^2 - ((3X^2 + aZ^4)^2 - 8XY^2) - 8Y^4}{(2YZ)^3}.$$

We want the result in Jacobian coordinates, and we know that $X_2' = X_2/Z_2^2$ and $Y_2' = Y_2/Z_2^3$. Hence, by choosing $Z_2 = 2YZ$ (and setting $X_2 = X_2 Z_2^2, Y_2 = Y_2' Z_2^3$), we get $2P = 2(X/Z^2 : Y/Z^3 : 1) = (X_2, Y_2, Z_2)$, where

$$\begin{aligned} X_2 &= (3X^2 + aZ^4)^2 - 8XY^2 \\ Y_2 &= (3X^2 + aZ^4)(4XY^2 - X_2) - 8Y^4 \\ Z_2 &= 2YZ \end{aligned} \tag{5.1}$$

(see [14, Example 3.20, p. 88]).
Equations 5.1 lead us directly to the algorithm for projective elliptic doubling given in Appendix A.10.4. of IEEE Std 1363-2000 [16]. In order to minimize the number of calculations needed, it uses some temporary variables to store values that are used more than once. This gives us Algorithm 7.

Algorithm 7 (Projective Elliptic Point Doubling) Let $E_{a,b}$ be an elliptic curve over $\mathbf{Z}_p$ and let (X_1, Y_1, Z_1) be a representation of a projective point P on $E_{a,b}$.

1. Calculate $M = 3X_1^2 + aZ_1^4$.

2. Calculate $Z_2 = 2Y_1Z_1$.

3. Calculate $T_1 = Y_1^2$ and $S = 4X_1T_1$.

4. Calculate $X_2 = M^2 - 2S$.

5. Calculate $T = 8T_1^2$

6. Calculate $Y_2 = M(S - X_2) - T$.

7. return $2P := (X_2, Y_2, Z_2)$.

[16, Appendix A.10.4, p. 123].

We have already concluded that Algorithm 7 works, so let us check how much time it requires. The step-by-step requirements are:

1. three field squarings $(X_1^2, Z_1^2, (Z_1^2)^2)$, one field multiplication (aZ^4) and four field additions $(X_1^2 + X_1^2 + X_1^2 + aX^4)$,

2. one field multiplication and one field addition,

3. one field squaring, one field multiplication and three field additions,

4. one field squaring and two field additions,

5. one field squaring and seven field additions,

6. one field multiplication and two field additions.

As multiplication is again much more expensive than addition and we do not make a difference between squaring and multiplication, we end up with ten field multiplications [16, p. 125]. If a is small enough, multiplication by it can be

done by repetitive additions, which saves one field multiplication. Furthermore if $a = -3$, then $M = 3X_1^2 + aZ_1^4 = 3X_1^2 - 3Z_1^4 = 3(X_1 - Z_1^2)(X_1 + Z_1^2)$, which saves one more field multiplication leading to only eight field multiplications for all steps.

As field multiplication, that is multiplication modulo p, is the dominant operation here, faster implementation for it leads to faster implementation for point doubling. To accomplish this, Sampo Sovio [34] suggested in his implementation of the cryptographic library an idea to use Montgomery arithmetics also in this context. Quick test run on 256-bit NIST-P256 modulus [11, p. 100] combined with previous results on 1024-bit modulus gives out the table

k	$M(k)$	$F(k)$
256	$0.48sec$	$0.26sec$
1024	$6.93sec$	$3.03sec$

where

- $M(k)$ gives the CPU time for performing $PC = 100000$ modular multiplications on k-bit modulus and

- $F(k)$ gives the CPU time for performing $PC = 100000$ Montgomery multiplications on k-bit modulus.

As we see in the table above, the Montgomery multiplication is faster than normal modular multiplication. This justifies the use of Montgomery arithmetics. It is done simply by converting points and coefficients of the curve to use their Montgomery representations.

- Point conversion: Affine point (x, y) is converted to projective representation (X, Y, Z) by simply setting $X := x, Y = y$ and $Z = 1$. Triple $(x, y, 1)$ is further converted to the used Montgomery representation by replacing each element by its Montgomery representation, which leads to $([x], [y], [1]) = (xR \bmod p, yR \bmod p, R \bmod p)$.

- Curve conversion: The coefficients a and b are replaced by their Montgomery representation[2], which leads to equation $y^2 = x^3 + [a]x + [b]$. Note that it is a good idea also to store some precomputed values (like $-M^{-1} \bmod b$ that is needed in Montgomery calculations) to the structure that is used for storing the elliptic curve in the implementation.

After these conversions we can use exactly the same algorithm as before, but all arithmetic operations can be performed in Montgomery arithmetics instead of the original field arithmetics.
Let us then move on to projective elliptic addition.

Algorithm 8 (Projective Elliptic Point Addition) Let $E_{a,b}$ be an elliptic curve over $\mathbf{Z}_p$ and let $(X_0, Y_0, Z_0), (X_1, Y_1, Z_1)$ be a representations of a projective points $P_0 \neq P_1$ on $E_{a,b}$ such that $Z_0 \neq 0$ and $Z_1 \neq 0$.

1. Calculate $U_0 = X_0 Z_1^2$ and $S_0 = Y_0 Z_1^3$.
2. Calculate $U_1 = X_1 Z_0^2$ and $S_1 = Y_1 Z_0^3$.
3. Calculate $W = U_0 - U_1$ and $R = S_0 - S_1$.
4. Calculate $T = U_0 + U_1$ and $M = S_0 + S_1$.
5. Calculate $Z_2 = Z_0 Z_1 W$.
6. Calculate $T_1 = TW^2$ and $X_2 = R^2 - T_1$.
7. Calculate $V = T_1 - 2X_2$ and $T_2 = VR - MW^3$.
8. Calculate $Y_2 = T_2/2$.
9. return $P_0 \oplus P_1 := (X_2, Y_2, Z_2)$.

[2] There exists also another kind of Montgomery form, see [7, p. 285]

[16, Appendix A.10.5, p. 125].

This algorithm is again based on simple substitutions and then storing the intermediate values for future use. By substitutions $x_i \leftarrow X_i/Z_i^2$ and $y_i \leftarrow Y_i/Z_i^3$, where $i \in \{0, 1\}$ in appropriate formula of Definition 8, we get the following values (after some simplifications):

$$X_2' = \frac{(Y_0Z_1^3 - Y_1Z_0^3)^2 - (X_1Z_2^2 + X_2Z_1^2)(X_0Z_1^2 - X_1Z_0^2)^2}{(Z_0Z_1(X_0Z_1^2 - X_1Z_0^2))^2}$$

and

$$Y_2' = \frac{(Y_0Z_1^3 - Y_1Z_0^3)(X_0Z_1^2(X_0Z_1^2 - X_1Z_0^2)^2 - X_2) - Y_1Z_2^3(X_0Z_1^2 - X_1Z_0^2)^3}{(Z_0Z_1(X_0Z_1^2 - X_1Z_0^2))^3},$$

where $X_2 = (Y_0Z_1^3 - Y_1Z_0^3)^2 - (X_1Z_2^2 + X_2Z_1^2)(X_0Z_1^2 - X_1Z_0^2)^2$. Bearing in mind that we are aiming for Jacobian coordinates ($X_2' = X_2/Z_2^2, Y_2' = Y_2/Z_2^3$), the natural choice for Z_2 is $Z_0Z_1(X_0Z_1^2 - X_1Z_0^2)$. By using the temporary variable from Algorithm 8, we get

$$\begin{aligned} X_2 &= R^2 - TW^2 \\ Y_2 &= R(U_0W^2 - X_2) - S_0W^3 \\ Z_2 &= Z_0Z_1W, \end{aligned}$$

which is almost what we wanted. As we know that $\oplus$ is an Abelian operation, we can switch the places of x_0 and x_1, and the places of y_0 and y_1. This gives us

$$Y_2 = R(U_1W^2 - X_2) - S_1W^3.$$

Adding these two representations together, we get

$$2Y_2 = R((U_1 + U_2)W^2 - 2X_2) - (S_0 + S_1)W^3,$$

which is the formula for $2Y_2$ that was presented in Algorithm 8. Using this latter formula, instead of calculating Y_2 directly, saves one field multiplication as TW^2 was previously computed (unlike U_0W^2). On the other hand, it adds one field addition and one division by two. This difference is not further analyzed, but we choose to concentrate on the IEEE standard presentation of the algorithm.

Let us look at the step-wise complexity of Algorithm 8

1. one field squaring (Z_1^2) and three field multiplications ($X_0Z_1^2$, $Z_1Z_1^2$, $Y_0Z_1^3$),
2. one field squaring and three field multiplications,
3. two field additions,
4. two field additions,
5. two field multiplications,
6. two field squarings, one field multiplication and one field addition,
7. three field multiplications (VR, WW^2, MW^3) and three field additions ($X_2 + X_2$, $T - 2X_2$, $VR - MW^3$),
8. division by 2.

As multiplication is again much more expensive than addition and we do not make difference between squaring and multiplication, we end up with 16 field multiplications [16, p. 126]. Note that if $Z_1 = 1$, which often is the case in scalar multiplication, this only requires 11 field multiplications.

Remark 11 In Algorithm 8 we have omitted special cases involving $\bigcirc$, but naturally $P + \bigcirc = \bigcirc + P = P$, for any point P, and $P + (-P) = (-P) + P = \bigcirc$. Full addition FullAdd$[(X_1, Y_1, Z_1)(X_2, Y_2, Z_2)]$ refers to combination of doubling, addition and special case handling, depending on the given inputs. For the scalar multiplication algorithm we define next, we also need subtraction, but that is naturally defined as adding the additive inverse, that is, Subtract$[(X_1, Y_1, Z_1)(X_2, Y_2, Z_2)] = FullAdd[(X_1, Y_1, Z_1), (X_2, -Y_2, Z_2)]$.

Now we have the components to build projective elliptic scalar multiplication. Direct analog of left-to-right binary exponentiation would take the binary presentation of the scalar multiplier, make doubling on every round and addition,

when the i^{th} bit of the multiplier is one. As computing additive inverses is not more expensive than actual addition, this can be made more efficient with the help of signed binary representation of the multiplier.

Definition 16 A *signed binary representation* of integer n is tuple $(n_{l-1}, \ldots, n_0)$ such that

$$c = \Sigma_{i=0}^{l-1} n_i 2^i,$$

where $n_i \in \{-1, 0, 1\}$. Representation $(n_{l-1}, \ldots, n_0)$ is said to be in *non-adjacent form* (NAF), if no two consecutive n_i's are non-zero.

Using a signed binary representation instead of the binary representation gives us *Double-and-Add-or-Subtract* algorithm. Every integer has a unique NAF representation and it is the optimal[3] signed binary representation [4, Lemma IV.1]. There are many ways to compute signed binary representation from a given binary representation, see for example [38, Section 3.1.2] and [18]. NAF can be computed easily, when we notice that

$$2^i + 2^{(i-1)} + \cdots + 2^j = 2^{(i+1)} - 2^j.$$

This means that substring of the form $(0, 1, 1, 1, \ldots, 1)$ can be replaced by substring $(1, 0, 0, \ldots, 0, -1)$. On average, an l-bit binary number contains $l/2$ zeros whereas its NAF representation contains $2l/3$ zeros. Hence, using NAF representation gives roughly 11 % speedup. [35, Section 6.5.5]

Algorithm 9 (Projective Elliptic Scalar Multiplication (PESM)) Let n be a positive integer, $E_{a,b}$ an elliptic curve over $\mathbf{Z}_p$ and (X, Y, Z) a representation of a projective point P on $E_{a,b}$ such that $Z \neq 0$.

1. Set $(X^\star, Y^\star, Z^\star) = (X, Y, Z)$ and $(X_1, Y_1, Z_1) = (X^\star/(Z^\star)^2, Y^\star/(Z^\star)^3, 1)$.

2. Calculate $3n$ and let $h_l h_{l-1} \ldots h_1 h_0$, where $h_l \neq 0$, be the its binary representation.

[3] Optimal, sometimes also called minimum weight, signed binary representation contains as few non-zero elements as possible.

3. Let $k_l k_{l-1} \dots k_1 k_0$ be the binary representation of n.

4. **for** $i := (l-1);\ i \geq 1;\ i--$ **do**

   4.1 Calculate $(X^\star, Y^\star, Z^\star) := 2(X^\star, Y^\star, Z^\star)$.

   4.2 **if** $h_i == 1$ and $k_i == 0$, **then** $(X^\star, Y^\star, Z^\star) :=$ FullAdd$[(X^\star, Y^\star, Z^\star), (X_1, Y_1, Z_1)]$.

   4.3 **if** $h_i == 0$ and $k_i == 1$, **then** $(X^\star, Y^\star, Z^\star) :=$ Subtract$[(X^\star, Y^\star, Z^\star), (X_1, Y_1, Z_1)]$.

5. **return** $nP := (X^\star, Y^\star, Z^\star)$.

[16, Appendix A.10.9, p. 130].

Remark 12 Naturally Algorithm 9 can also be extended for non-positive integers given that

$$0P = \bigcirc$$

and

$$(-n)P = n(-P).$$

The basic idea in Algorithm 9 is that it calculates signed binary representation of the multiplier n on the fly.

Proposition 1 *Let n be positive integer, let $(h_l, h_{l-1}, \dots, h_1 h_0)_2$, where $h_l \neq 0$, be the binary representation of $3n$ and $(k_l, k_{l-1}, \dots, k_1, k_0)_2$ the binary representation of n. Now let $n_{l-2}, \dots, n_1, n_0$ be such that, for $i = l-1, \dots, 1$,*

- *if $h_i == 1$ and $k_i == 0$, $n_{i-1} = 1$;*
- *if $h_i == 0$ and $k_i == 1$, $n_{i-1} = -1$;*
- *otherwise, $n_{i-1} = 0$.*

Now $(1, n_{l-2}, \dots, n_1, n_0)_{\bar{2}}$ is a signed binary representation of n.

Proof. If we do a bit-by-bit subtraction $3n - n$ and move to signed representation at the same time to avoid carries, we get signed binary representation of $2n$. As multiplication by 2 is just a left-shift for a binary number, by ignoring the zero in the end of signed binary representation of $2n$, we get signed binary representation of n. Hence, the part stated in the for-loop is true. □

By Proposition 1 it is clear that Algorithm 9 works in the same way as exponentiation – on every round we double and at the same time we go through the signed binary representation subtracting or adding accordingly.
Now that we know that Algorithm 9 works, we can look at its complexity:

1. One field squaring and one field multiplication if $Z \neq 1$.

2. One field multiplication, the i^{th} bit is calculated on the fly as algorithm proceeds.

3. The i^{th} bit is calculated on the fly as algorithm proceeds.

4. Loop is executed $l - 1$ times.

    4.1 One doubling.

    4.2 **if** $h_i == 1$ and $k_i == 0$, **then** one full addition (might be doubling or normal addition or limit case).

    4.3 **if** $h_i == 1$ and $k_i == 0$, **then** one subtraction.

Note that the performance Algorithm 9 can be improved in several ways, see, for example, [38] or [12]. As field operation only takes negligible time, the total complexity is $(l-1)D+kA$, where D is the time consumed for doubling a point, A is the time consumed for full addition (assuming subtraction takes practically the same time as addition) and k is the Hamming weight[4] of the SDR representation that is calculated while performing the algorithm.

[4] The Hamming weight is the number of non-zero symbols.

Note that similarly to Double-and-Add-algorithm (see SPA attack for Double-and-Add for example in [37]), implementation of Double-and-Add-or-Subtract algorithm as such is vulnerable to power analysis attacks. From power traces it is possible to see on which rounds only doubling was performed. This means that the zero bits of the multiplier are revealed. One possible way to overcome this is to use the Montgomery Power Ladder [22, p. 261] that performs the same operations on every round.

Algorithm 10 (PESM with Montgomery Power Ladder) Let n be positive integer, $E_{a,b}$ be an elliptic curve over $\mathbf{Z}_p$ and let P be a (projective) point on $E_{a,b}$.

1. Let $k_l k_{l-1} \dots k_1 k_0$ be the binary representation of n, where $k_l \neq 1$.

2. Set $X = \bigcirc$ and $Y = P$.

3. **for** $i := l;\ i \geq 0;\ i--$ **do**

   3.1 **if** $k_i == 0$, **then** calculate $Y := \text{FullAdd}[X, Y]$ and $X := 2X$.

   3.2 **if** $k_i == 1$, **then** calculate $X := \text{FullAdd}[X, Y]$ and $Y := 2Y$.

4. **return** $nP := X$.

[38, Algorithm, p. 34].

To see that Algorithm 10 works, it is enough to notice the invariant $Y - X = P$. Now it is easy to see that if we concentrate on X, the algorithm performs exactly like the classical Double-and-Add algorithm for scalar multiplication. As Double-and-Add algorithm is analogous to Square-and-Multiply, which we already dealt with in context of exponentiation, we can skip more detailed proofs.
It is easy to see that the complexity of elliptic scalar multiplication with the Montgomery Power Ladder is $(l + 1)(D + A)$, where D is the time consumed for doubling a point and A is the time consumed for full addition.

5.2 Testing the performance of one implementation

The cryptographic library[5] (CL), that was used for testing, provides an interface for the scalar multiplication and affine addition (and affine subtraction). Comparing the performance of these algorithms does not make much sense as projective co-ordinates are used in implementation of scalar multiplication. So, for testing I decided to use scalar multiplication provided by the interface and to compare that with the internal implementations of projective addition and doubling. Subtraction takes practically the same time as addition, so it was not included separately.
For test setup I decided to use NIST-P-256 and NIST-P-192 curves given in Digital Signature Standard [11, Appendix D.1.2] and supported by the given cryptographic library. For the actual test cases points on the curve are needed. Using random numbers and trying to convert these into points of the curves is one option, but it is easier to start with the generators. We call the generator of NIST P-192 G_{p192}, which has a hexadecimal value

```
( 188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd 82ff1012,
  07192b95 ffc8da78 631011ed 6b24cdd5 73f977a1 1e794811 )
```

and the generator of NIST P-256 G_{p256}, which has a hexadecimal value

```
( 6b17d1f2 e12c4247 f8bce6e5 63a440f2
  77037d81 2deb33a0 f4a13945 d898c296,
  4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16
  2bce3357 6b315ece cbb64068 37bf51f5 ).
```

To test the performance of projective Doubling and Addition, I decided to use random points by choosing random 32-bit multipliers and multiplying the generators of the curves with these values. I repeated this 50 times and calculated the average. In pseudo code this would give:

[5] Implementation of the cryptographic library is done by S. Sovio [34]

```
#include <time.h>

clock_t t;
double time;
double sum = 0;
int i;

for (i = 0; i < 50; i++)
{
    <generate 32-bit randoms rand1 and rand2>

    <calculate P1 = rand1*g and P2 = rand2*g>

    <convert P1 and P2 to projective points>

    t = clock();

    <operation>

    time = ((double)(clock() - t))/CLOCKS_PER_SEC;

    sum += time;
}

<print average time: sum/j>
```

where the operation is either projective doubling $2P_1$ or projective addition $P_1 + P_2$. As a result I got the table

$curve$	$Double(P1)$	$Add(P1, P2)$
$NIST - P192$	$0.1328sec$	$0.2164sec$
$NIST - P256$	$0.2154sec$	$0.4002sec$,

where

- Double(P1) gives the average CPU time (of 50 trials) for performing PC projective doublings of P1, and P1 is achieved on each trial round from generator G by multiplying it with 32-bit random,

- Add(P1,P2) gives the average CPU time (of 50 trials) for performing PC projective additions of P1 and P2, and P1 and P2 are achieved on each trial round from generator G by multiplying them with 32-bit randoms,

where $PC = 10000$.
As we concluded earlier the expected performance of projective doubling is relative to 8 field multiplications (8 M) and for projective addition to 16 field multiplication (16 M). As we see from the table above, the relation between doublings and additions in the tested cryptographic library is about 1:2 as it should be. We can also compare the times for NIST-P256 curve to the times that were achieved for modular and the Montgomery multiplication. According to the previous tests PC*8*M for 256-bit modulus should take about $8 * F(256)/10 = 0.208$ seconds if we count with Montgomery multiplication and $8 * M(256)/10 = 0.384$ seconds if we use ordinary modular multiplication. This is well in line with the result above, as we need to use some time for conversions too, in order to use the Montgomery multiplication which is what we do.
To test the performance of scalar multiplication, I decided to try with k-bit random multipliers (average test), where k is 192 or 256. As a result I got the table

k	$curve$	$rand_k * G$ with conv.	$rand_k * G$ w/o conv.
192	$NIST - P192$	$0.7708sec$	$0.4522sec$
256	$NIST - P256$	$1.334sec$	$0.946sec$,

where

- $rand_k * G$ with conv. gives the average CPU time (of 50 trials) for performing 100 scalar multiplications with 192- or 256-bit random multiplier and IEEE algorithm with the interface function that takes in affine coordinates and

- $rand_k{*}G$ w/o conv. gives the average CPU time (of 50 trials) for performing 100 scalar multiplications with 192- or 256-bit random multiplier and IEEE algorithm with the primitive function that takes in coordinates in projective Montgomery form.

The same tests were run for scalar multiplication using the Montgomery Power Ladder and gave the following results:

k	$curve$	$rand_k * G$ with conv.	$rand_k * G$ w/o conv.
192	$NIST - P192$	$1.1438sec$	$0.8112sec$
256	$NIST - P256$	$2.096sec$	$1.6952sec$,

where

- $rand_k \times G$ with conv. gives the average CPU time (of 50 trials) for performing 100 scalar multiplications with 192- or 256-bit random multiplier and using the Montgomery Power Ladder with the interface function that takes in affine coordinates and

- $rand_k \times G$ w/o conv. gives the average CPU time (of 50 trials) for performing 100 scalar multiplications with 192- or 256-bit random multiplier and using the Montgomery Power Ladder with the primitive function that takes in coordinates in projective Montgomery form.

As we saw in the previous section, the expected time for scalar multiplication using IEEE algorithm is $(l-1)D + h_w A$, where D is time consumed for doubling a point, A is the time consumed for full addition (assuming subtraction takes practically the same time as addition) and h_w is the Hamming weight of the SDR representation of the multiplier that is calculated while performing the algorithm. Supposing that IEEE uses minimum weight, this gives on average

$curve$	$k * (D + A/3)$
$NIST - P192$	$192 * (0.1328 + 0.2164/3)/100 = 0.39sec$
$NIST - P256$	$256 * (0.2154 + 0.4002/3)/100 = 0.89sec.$

With the Montgomery Power Ladder both Doubling and Addition are performed on every round. So the expected values would be

$curve$	$k * (D + A)$
$NIST - P192$	$192 * (0.1328 + 0.2164)/100 = 0.67sec$
$NIST - P256$	$256 * (0.2154 + 0.4002)/100 = 1.576sec.$

Both with the IEEE case and the Montgomery Power Ladder, we notice that average expectations match quite well to the received results, when we look at the scalar multiplication primitives that assume the conversions have already been made. Of course in the real world numbers hardly hang around in projective Montgomery form, so there we should expect somewhat bigger overhead from the conversions.

6 Conclusions

In this work I have given an introduction to some basic implementation issues in Public Key Cryptography. I did this by giving an introduction to the needed mathematical background and then by studying the IEEE standard algorithms that are needed in implementing (non-authenticated) Diffie-Hellman Key Exchange both in traditional and in Elliptic Curve setting. The main algorithms are exponentiation in traditional setting and scalar multiplication in elliptic curve setting. These algorithms further need some building blocks like addition and multiplication on big numbers.

We have an implementation of these algorithms in our teams cryptographic library, so I created a small test setup on top of the existing implementation to compare expected running times of the introduced algorithms to actual running times of the functions that are present in our teams cryptographic library. Our algorithms performed somewhat according to the expectations, but in affine scalar multiplication conversions introduced a somewhat surprising overhead. This is one thing that should be kept in mind when implementing algorithms – if we need to make conversions on each step, the final result might come out much slower than we would expect.

Nevertheless, if we compare scalar multiplication to exponentiation (bearing in mind the safety level correspondences), scalar multiplication performs much better on our teams software implementation. According to NIST (see recommendations at http://www.keylength.com/en/4/, link tested 01-Dec-2011), 256-bit elliptic curves correspond to 3072-bit groups, and already with 2048-bit numbers exponentiation is much slower than scalar multiplication on a 256-bit curve (36.92 seconds versus 2.098 seconds) even if we use the Montgomery Power Ladder that is considerably slower than the IEEE algorithm but presumably safer against side-channel attacks. Of course the situation changes dramatically when hardware support is available for one algorithm but not for the other.

I hope I managed to introduce the needed background and the algorithms so that it helps to understand what is needed in practice when one wants to implement Public Key Cryptosystem starting from scratch. I tried to write down everything

in such a way that when I later need to understand something again, I can recall it by checking my notes. Sometimes this has been the case, other times I could have put in more details. There is also a lot that I did not cover here. The world is full of research on how to make all this faster, then on how to find security holes, for example, through side channels, and then again on how to protect against those and then how to break the protections by introducing faults etc. In security, the bad guys need only one tiny whole in your system. And all you need to do, is to make sure there are none. Good luck with it!

Bibliography

[1] T.F Al-Somani: *Performance Evaluation of Elliptic Curve Projective Coordinates with Parallel GF(p) Field Operations and Side-Channel Atomicity*. Journal of Computers, 5, 1, January 2010. Available at http://www.academypublisher.com/ojs/index.php/jcp/article/viewFile/050199109/1350 (link tested 01-Dec-2011).

[2] ANS X9.42-2001: *Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography*.

[3] ANS X9.63-2001: *Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography*.

[4] I. F. Blake, G. Seroussi and N. P. Smart: *Elliptic Curves in Cryptography*, London Mathematical Society Lecture Note Series. 265. Cambrigde University Press, 1999.

[5] J. W. Bos and M. E. Kaihara: *PlayStation 3 computing breaks 2ĥ0 barrier: 112-bit prime ECDLP solved*, EPFL Laboratory for cryptologic algorithms - LACAL, 2009. Available at http://lacal.epfl.ch/112bit_prime (link tested 01-Dec-2011).

[6] A. Bosselaers, R. Govaerts and J. Vandewalle: *Comparison of three modular reduction functions*. Appeared in Advances in Cryptology - CRYPTO 1993, Lecture Notes in Computer Science 773, D. R. Stinson (ed.), Springer-Verlag, 175–186, 1993.

[7] H. Cohen and G. Frey (Sc. Eds.): *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2006.

[8] *Discrete Logarithms*, ECRYPT II wikipage. Available at http://www.ecrypt.eu.org/wiki/index.php/Discrete_Logarithms (link tested 01-Dec-2011).

[9] *Discrete logarithm records* wikipedia page. Available at http://en.wikipedia.org/wiki/Discrete_logarithm_records (link tested 01-Dec-2011).

[10] N. Ferguson and B. Schneier: *Practical Cryptography*. Wiley, 2003.

[11] FIPS PUB 186-3: *Digital Signature Standard (DSS)*. National Institute of Standards and Technology, 2009. Available at http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf (link tested 01-Dec-2011).

[12] D. Gordon: *A survey of fast exponentiation methods*. Center for Communications Research, San Diego, December, 1997. Available at http://www.ccrwest.org/gordon/jalg.pdf (link tested 01-Dec-2011).

[13] P. Haukkanen: Algebra I, luentomateriaali, 2004. Saatavilla osoitteessa http://mtl.uta.fi/Opetus/Algebra/algI04.pdf (linkki tarkastettu 01-Dec-2011).

[14] D. Hankerson, A. Menezes and S. Vanstone: *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[15] E. Hyry: *Kryptografian algebralliset menetelmät*, luentomateriaali, Tampereen yliopisto, syyslukukausi 2008.

[16] IEEE Std 1363-2000: *IEEE Standard Specification for Public-Key Cryptography*, 2000.

[17] D. Johnson, A. Menezes and S. Vanstone: *The Elliptic Curve Digital signature Algorithm (ECDSA)*. Certicom Corporation, 2001. Available at http://tlapixqui.izt.uam.mx/sem_cripto/PGP/ecdsa.pdf (link tested 01-Dec-2011).

[18] M. Joye and S-M Yen: *Optimal Left-to-Right Binary Signed-Digit Recording*, IEEE Transactions on Computers, 49, 7, 740–78, July 2000.

[19] A. Menezes, T. Okamoto and S. Vanstone: *Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field*, IEEE Transactions on Information Theory, 39, 5, 1639-1646, 1993.

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

[21] P. Montgomery: *Modular Multiplication Without Trial Division*. Math. Computation, 44, 519-521, 1985.

[22] P. Montgomery: *Speeding the Pollard and Elliptic Curve Methods of Factorization*, Mathematics of Computation, 48, 177, 243–264, 1987. Available at (link tested 27-Sep-2011).

[23] D. Naccache, N. Smart and J. Stern: *Projective Coordinates Leak*. Presentation at EuroCrypt 2004. Available at http://www.zurich.ibm.com/eurocrypt2004/slides/session8talk2.pdf (link tested 01-Dec-2011).

[24] P. Q. Nguyen: *Can We Trust Cryptographic Software? Cryptographic Flaws in GNU Privacy Guard v1.2.3*. Appeared in C. Cachin and J. Camenisch (Eds.): EUROCRYPT 2004, Lecture Notes in Computer Science 3027, 555-570, Springer, 2004.

[25] NIST SP 800-56A: *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*. March, 2007

[26] J.M. Pollard: *Monte Carlo methods for index computation (mod p)*. Mathematics of Computation 32 (143): 918–924, 1978.

[27] RFC 2246: *The TLS Protocol Version 1.0*. January 1999. Available at http://tools.ietf.org/html/rfc2246 (link tested 01-Dec-2011).

[28] RFC 4492: *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS).* May 2006. Available at http://www.faqs.org/rfcs/rfc4492.html (link tested 01-Dec-2011).

[29] K. Ruohonen: Mathematical Cryptology, lecturing material, 2008. Available at http://math.tut.fi/ ruohonen/MC.pdf (link tested 01-Dec-2011).

[30] R. Schoof: *Elliptic Curves over Finite Fields and the Computation of Square Roots mod p.* Math. Comp., 44(170): 483–494, 1985. Available at http://www.mat.uniroma2.it/ schoof/ctpts.pdf (link tested 01-Dec-2011).

[31] D. Shanks: *Class number, a theory of factorization and genera*. Proc. Symp. Pure Math. 20, pages 415–440. AMS, Providence, R.I., 1971.

[32] V. Shourp: *Lower bounds for discrete logarithms and related problems*, in Proc. Eurocrypt '97, 256–266, 1997. Revised version available at http://www.shoup.net/papers/dlbounds1.pdf (link tested 01-Dec-2011).

[33] N.P. Smart: *The discrete logarithm problem on elliptic curves of trace one* in Journal of Cryptology 12 (3): 193–196, 1999.

[34] S. Sovio, personal communication, 2010.

[35] D. R. Stinson: *Cryptography - Theory and Practice*, 3rd edition, Chapman & Hall/CRC, 2006.

[36] C. Studholme: *The Discrete Log Problem*, June 21, 2002. Available at http://www.cs.toronto.edu/ cvs/dlog/research_paper.pdf (link tested 01-Dec-2011).

[37] K. Wu, H. Li, T. Chen and F. Yu: *Simple Power Analysis on Elliptic Curve Cryptosystem and Countermeasures: Practical Work*, isecs, 1, 21–24, Second International Symposium on Electronic Commerce and Security, 2009.

[38] O. Yayla: *Scalar Multiplication on Elliptic Curves*, Master's Thesis, Department of Cryptography, Middle East Technical University, August 2006. Available at http://www3.iam.metu.edu.tr/iam/images/3/3e/O%C4%9Fuzyaylathesis.pdf (link tested 01-Dec-2011).