# Elliptic Curves Cryptography and Applications

May 19, 2014

**Abstract**

Elliptic curves are mathematical objects that happen to be suitable for cryptography of the Diffie Hellman kind, which is based on the discrete logarithms problem DLP. This introduction was written as part of a self learning process and takes an software engenieering perspective. The mathematical notation used is a mixture of different things I found in the internet, mainly Wikipedia and the many other sources who published their insights on the matter.

## 1 Trapdoors

To understand why modern asymmetric cryptographic algorithms are considered secure, it is worth to look at the mathematics behind it from a basic perspective. Think of a number $m = p \cdot q$ where $p$ and $q$ are integers and primes. According to an important theorem, there are no integers except $p$ and $q$ that satisfy the equation. We say that $p$ and $q$ are the prime factors of $m$ and the only numbers grater than one that divide $m$.

Now given some $m$ known to have two different prime factors, how difficult is it to calculate their actual values for that $m$? We are looking for integers $p, q$ so that $q = m/p$, with no remainder. This is one equation with two unknowns. If we look at the curve $C$ of function $q = f(p) = m/p$ in the $p, q$ plane we find a hyperbola. The solutions of the equation are points on C where both coordinates have integral value. Call the set of points with integral coordinates a lattice $L$, then the intersection $C \cap L$ contains all the solutions.

While this looks simple, it turns out to be pretty difficult to calculate the intersection of a curve and a lattice, if the curve is suitable. The number $m$ should be fairly large to prevent exhaustive search by simply trying all possible values or by colliding available data.

The RSA cryptosystem is based on the fact that it is possible to calculate $m = pq$ easily, while the another operation, the factoring, turns out to be practically infeasible, if $p$ and $q$ are large enough. This fact is used to construct a *trapdoor function* with has the property that it can be calculated in one way by everybody knowing $m$, but only if one knows $p$ and $q$ he can calculate the inverse.

The particular factoring problem presented here is called the *RSA problem*. Who can solve it will be able to break all RSA keys. Currently no algorithm is known to solve the problem in what is called *polynominal time*. Practically this means, that even if computers get faster, the cryptosystem can adapt by using larger numbers.

## 2 Groups and Fields

These very important concepts from algebra are worth studying, but I must confess my understanding is too limited to give any guidance. From the software perspective, all is very natural, even more natural than the natural numbers, indeed. This is a hands-on approach using the Wikipedia.

So how do we cook a group? Just google it: first, we need a set. A set of what? Well, say some numbers: 1, 2, 3 and 4. Then a binary operation, the group law: say addition to start with. Then check the

axioms. $1 + 4 = 5$ is not in the set. Obviously, we need a tweak to avoid 5 and larger numbers. So, if the sum $s$ of two numbers from our group is greater than 5, we replace it with the remainder $r = s \mod 5$. Now $1 + 4 \mod 5 = 0$ is again not in our set. Lets add it, and voila: the group axioms hold for numbers $\{0, 1, 2, 3, 4\}$ with addition defined as $c = a + b \mod 5$.

This is called *modular arithmetics* and is quite suited for computers, as it operates with finite sets which are more easy to represent than other, possibly infinite, sets. The example above is called $\mathbb{Z}_5$. On computers,

Now groups are very fundamental and sets with a finite number of elements can be represented by a number of prototypes or combinations of these. One of these prototypes is called the *cyclic group* and has the property that every element of the group is generated by repeated addition of a special element, called the *generator* of the group.

Lets check if group $G(\mathbb{Z}_5, +)$ is cyclic. We choose 2 as the generator. We add $2 + 2 = 4 \mod 5$ and get another element. Now we add $4 + 2 = 1 \mod 5$, $1 + 2 = 3 \mod 5$, $3 + 2 = 0 \mod 5$ and $0 + 2 = 2 \mod 5$ where the story begins to repeat. As demanded, the calculation of $2n \mod 5$ for $n = 1, 2, ...$ yields the elements of $\mathbb{Z}_5$ ordered as $\{2, 4, 1, 3, 0\}$. We can choose any element except 0 as a generator.

Since we've been talking about addition all the time, these groups are called *additive groups*. Multiplicative groups are based on multiplication instead of addition. Lets see what this does with $\mathbb{Z}_n$: for any two numbers $a$ and $b$, the product $c = a \cdot b \mod n$ is also an element. But the inverse axiom does not hold: it is required that $a \cdot a^{-1} = 1 \mod n$, which does not work for 0. If we leave 0 out, it works and we get the *multiplicative group of integers modulo n*, if $n$ is a prime number. Why prime?

First figure out the case for 5, the set $\{1, 2, 3, 4\}$ being called $\mathbb{Z}_5^*$. Element 1 is obviously not a candiate for a generator. We pick 2: $2 \cdot 2 = 4 \mod 5$, $4 \cdot 2 = 3 \mod 5$, $3 \cdot 2 = 1 \mod 5$ and $1 \cdot 2 = 2 \mod 5$ closes the circle. We also find that 4 is not a generator. As a practical result $n$ must be prime because otherwise no generator exists. The existance of a generator however is a precondition for the group to be cyclic.

We finish with this: a field $\mathbb{F}$ is a set together with two binary operations, addition and multiplication, so that the field axioms are fulfilled. There is some subtle handling of the 0 element which is not required to have an inverse. For any $p$ prime, a finite field $\mathbb{F}_p$ exists with all operations as usual, modulo $p$.

# 3 Diffie Hellman and the DLP

Some what does modular arithmetic have to do with the lattice problem? Given field $\mathbb{F}_p$ repeated multiplication of element $g$ can be written as $a = g \cdot g \cdot ... = g^n \mod p$, the n-th power of $g$ in the finite field, which is also a multiplicative group. Now given fairly large prime number $p$ and some suitable generator $g$, we can describe the Diffie Hellman key exchange as follows:

- Let Alice choose a secret number $a$ and calculate $A = g^a$ which she sends to Bob.
- Bob also chooses a secret number $b$ and sends $B = g^b$ to Alice.
- Now alice calculates $B^a = (g^b)^a = g^{ab} = s$.
- Bob calculates $A^b = (g^a)^b = g^{ba} = g^{ab} = s$.

It's claimed that $s$ is now a secret shared between Alice and Bob and that there is not way to calculate $a$, $b$ or $s$ from $A$ and $B$, the messages transmitted.

Let's attack message $A$. We presumably know $g$ as well, so we need to find $a$ so that $A = g^a$ or $a = log_g A$. Enumeration won't work, if $p$ is big enough. It turns out to be difficult and that's why it's called the *disrete logarithm problem* or DLP.

To understand why it is difficult, consider the curve $C : y = 2^x,\ x, y \in \mathbb{R}$. Obviously, the points with integral coordinates $C \cap L$ are all pairs $(x, 2^x)$. Now given some value $v = 2^u$ find u. In the world of real numbers $\mathbb{R}$, this can be approximated in polynominal time. In $\mathbb{F}_p$ things look a bit different however. Remember that here $\tilde{v} = 2^u \mod p$. So if $u$ is a $k$-digit number, in $\mathbb{N}$, $v$ will be a $2^k$-digit number.

Assuming $k = 512$ this gives use the quite fantastic number of roughy $10^{154}$ bits needed to store the number. In $\tilde{v}$ this is compressed by $p$ to $k$ digits again, but at the same time the information needed by the real number algorithm is lost.

From the geometrical interpretation of the problem, we see that the discrete logarithm is hard because we are (again) looking for curve lattice points and all we've got is a residue $\tilde{v}$ of the real value modulo $p$. The residue $\tilde{v}$ represents all numbers $v = \tilde{v} + kp,\ k \in \mathbb{Z}$, the majority of bits of $v$ unknown, so one must enumerate a large region of $k$ numbers to find it.

If we compare to the RSA problem, the burden seems even higher. In RSA, the curve is choosen to have two secret lattice points at numbers in the range of $k/2$ digits, if $k$ is the bitlength of the modulus $m$. With DLP, the curve is generic but the number of digits is tremendously bigger.

# 4 Building Blocks

While RSA is very popular, it certainly lacks the feature of *perfect forward secrecy* PFS. This not an inherent limitation but a consequence of how RSA is defined and used. They key generation process is difficult and computationally expensive, making it unattractive to create keys frequently.

DLP has a number of advantages, including PFS and cheap key generation, that can be exploited to realise more sophisticated security protocols. Lets first have a look at DLP based signature scheme, DSA proposed by NIST. In this scheme, each user has a key pair $(x, g^x)$ where $x$ is the private key and $X = g^x \mod p$ is the private key for some agreed $\mathbb{F}_p$.

It has some significance for the discussion to define the multiplicative order $q$ of element $g$ in $\mathbb{F}_p$. Since the multiplicative part of $\mathbb{F}_p$ has $p - 1$ elements (zero is excluded), the order of $g$ must be a divisor of $p - 1$. Furthermore $q$ must be a prime and $g^q = 1 \mod p$, the identity element of the multiplicative group.

## 4.1 Digital Signature

Here is how the DSA signature is calculated:

- Derive a suitable value $z$ from the content to be signed.
- Generate a random value $k$ so that $0 < k < q$.
- Calculate $r = g^k \mod p \mod q$
- Calculate $s = k^{-1}(z + xr) \mod q$.

The signature $(r, s)$ can be verified as follows:

- $u_1 = zs^{-1} \mod q$.
- $u_2 = rs^{-1} \mod q$.
- Calculate $v = r = g^{u_1} y^{u_2} \mod p \mod q$.

To get the general concept here, we look why this is so. We simply insert everyting into the last formula:

$$g^{u_1} X^{u_2} \mod p \mod q = g^k \mod p \mod q$$

For brevity, the mod stuff is not shown in the following:

$$g^{u_1} X^{u_2} = g^{u_1} g^{xu_2} = g^{u_1 + xu_2} = g^{\frac{z+rx}{s}} = g^{\frac{(z+rx)k}{z+rx}} = g^k$$

Let me quickly deviate and present the additive notation of a group. This can be done depsite that fact that the group is multiplicative. Write $x \oplus g$ for $g^x$. Then above can written as

$$u_1 \oplus g + u_2 \oplus X = (u_1 + xu_2) \oplus g = s^{-1}(z + rx) \oplus g = k(z + xr)^{-1}(z + xr) \oplus g = k \oplus g$$

or even more compressed:

$$u_1 G + u_2 X = (u_1 + xu_2)G = s^{-1}(z + rx)G = k(z + xr)^{-1}(z + xr)G = kG$$

with the modification that capital letters are used for the generator and any products (powers) of G.

So where come the safety here from, exactly? Only signature parameter $s$ depends on the private key $x$, as $s = (z + xr)/k \bmod q$. Obviously, $x$ can be calculated as

$$x = \frac{sk - z}{r} \bmod q$$

if we know $k$. Or in other words: don't disclose $k$ but destroy it a soon as used. Also use good random numbers. Also never reuse any value of $k$, the effect is catastrophic: assume the same $k$ had been used to generate signatures $(r, s_1)$ and $(r, s_2)$, then

$$s_i = \frac{z_i + xr}{k},\ i = 1, 2$$

$$k = \frac{z_i + xr}{s_i},\ x = \frac{ks_i - z_i}{r}$$

Now we insert one equation into the other, for different $i$ and $j$:

$$k = \frac{z_i + ks_j - z_j}{s_i} = \frac{z_i - z_j}{s_i} + k\frac{s_j}{s_i}$$

$$k(1 - \frac{s_j}{s_i}) = k(\frac{s_i - s_j}{s_i}) = \frac{z_i - z_j}{s_i}$$

$$k = \frac{z_i - z_j}{s_i - s_j}$$

Disclosure of $k$ reveals the private key $x$ which can be readily computed. Conclusion is that $k$ is needed to make the scheme secure.

Lets recapitulate why the signature is secure: if we assume the DLP is not solvable and the attacker does not know $x$ or $k$, the problem comes down to solving $k = log_g(r)$.

## 4.2 Encryption

Encryption schemes based on DLP follow the Diffie Hellman key exchange to create a shared secret which is used to derive conventional keys for symmetric encryption and message authentication.

But the shared secret can also be use to encrypt values in $Z_q$ directly. Take $z$ to represent a message to be encrypted by Alice so that only Bob can read it:

- Alice chooses random secret $k$ and $c_1 = kG$.
- Alice calculates $s = k \cdot bG$ and $c_2 = z \cdot s$.
- Alice sends $(c_1, c_2)$ to Bob.
- Bob calculates $s = bc_1$.

- Bob calculates $z = s^{-1}c_2$.

The procedure constructs a shared secret based on $b$ and the ephemeral key $k$. The private key of Alice is not involved in the calculation and Alice remains anonymous, meaning that the message is not authenticated.

We see that $(c_1, c_2) = (kG, zkbG)$ is just one of many options how to embed the message $z$ into the calculation. It could as well be $(kG, z + kbG)$, where Bob would calculate $z = c2 - s$.

## 4.3 Certificates

You may know X.509 certificates used in conjunction with HTTPS in the browser. That kind of certificate has a lot of structure, which comes down to a *distinguished name*, a *public key* and some other data, such as constraints, wrapped with a signature on the serialized content of all that data, called the *to-be-signed*.

There is a interesting alternative called *implicit certificates* based on the DLP. Assume we have a *certification authority* CA with key $(c, cG)$. The following procedure associates a key pair with a user identifier $u$.

- To initiate the process, Alice creates a secret, random $\alpha$ in the range $0 < \alpha < q$ and sends $\alpha G$ to the CA.
- The CA chooses $k$ randomly and calculates $\gamma = \alpha G + kG$.
- CA calculates a hash based value $e = h(\gamma, u, ...)$.
- CA calculates $s = ek + c$.
- Alice calculates $a = e\alpha + s$ to get key pair $(a, aG)$.
- Everybody can calculate $aG = e\gamma + cG$, the public key.

The explanation is simple:

$$aG = e\gamma + cG = (e\alpha + ek + c)G = (e\alpha + s)G$$

Obviously the process is different from a flow where a secret key is created first and a certificate is requested later. Here Alice can only calculate the secret key $a$ once she receives confirmation $s$ from the CA.

## 4.4 Authentication

I want to finish this section with a *proof of knowledge* protocol. Assume Alice, having key pair $(a, aG)$ wants to prove to Bob that she posses key $a$.

- First, Alice, the *prover*, generates a random value $k$ and calculates $kG$ which she sends to Bob.
- Bob, the *verifier*, remembers $kG$ and generates a random challenge $c$.
- Alice now calculates $s = k + ca$ and sends it to Bob.
- Bob checks that $kG + cA = (k + ca)G = sG$.

This proves to Bob that Alice knew $k$ and $a$ at the time she calculated $s = k + ca$, which entangles $c$, the challenge generated just before with $a$ and $k$, the values Alice claims to posses. Because we assume that DLP ensures these values can not be calculated from $s$ or any of the other shared values, the procedure is safe.

# 5 Elliptic

So far we haven't met any elliptic curves, as promised in the title. Well, lets get to that by stating the equation for an elliptic curve, following common internet resources:

$$y^2 = x^3 + ax + b$$

The solution of this equation is a subset of the real plane $\mathbb{R}^2$. The resulting curve has some remarkable properties that allow us to construct a specific group suited for the DLP but with less bits required to be secure. The remarkable property, that I will only sketch as material can easily be found, is that points on that curve can be added in a way which ensures that the result will also be a point of the curve, as required for a group. The group operation is defined so that every $C = A \oplus B$ is either a point on the curve or an additional point at infinity, which serves the role of the group identity element.

The beautyfull thing with mathematics is now that we can readily apply the results of the previous discussion if we choose $x$ and $y$ from a field $\mathbb{F}_p$, giving us:

$$y^2 = x^3 + ax + b \bmod p$$

The solution of this equation is a set of points $P_i = (x, y) \in \mathbb{F}_p^2$, which together with the point at infinity, build a cyclic group, sometimes. For cryptography, they must and we can find a generator point $G$ of order $q$ so that $qG = 0$. Typically the numbers are given by some standard.

Technically we have now two different types of objects to cope with, namely points and factors, that's why the additive notation is somewhat easier, if we choose to use capital letters for points. We can restate the Diffie Hellman key exchange

$$s = abG = a(bG) = b(aG) = bA = aB$$

and the signature for message $z$:

$$r = R_x \text{ where } R = kG$$
$$s = k^{-1}(z + ar) = k^{-1}\omega \bmod q$$

By $R_x \in \mathbb{F}_p$ we denote the x-coordinate of point $R$, deemed to be sufficient for the task. To verify the signature $(r, s)$, calculate

$$R = zs^{-1}G + rs^{-1}A = (zk\omega^{-1} + ark\omega^{-1})G = k\omega^{-1}\omega G = kG$$

# 6 Random Numbers

We have seen that random numbers, often termed $k$, play an important role in the security of DLP based cryptography. The role of the *random number generator* cannot be underestimated. I want to add some rather speculative considerations here, due to lack of theoretical background.

First consider a very simple computer system, offline, that has everything but a clock. Every time it starts up it finds itself in an exactly defined starting state. How could such a system generate random numbers? Obviously, without help from outside it can't. Any output produced by a Turing machine with a finite tape, which such a computer is equivalent to, is completely defined and so this is a *deterministic* system.

The DLP algorithms have an interesting property which RSA does not share. To prevent a computer using DLP from creating bad random numbers, namely such that could be predictable in some way, entropy must be added to the system. Every time a random bit is created, the entropy decreases, signature

and authentication procedures require random values and consequently consume entropy when being used. The RSA algorithm does not require this, random number are only used during key generation.

To produce a different output on every run, assume the program on the deterministic computer could access an entropy source to ask for a true random bit at every startup. Would it be less deterministic now? Indeed, we find the program could now behave in two different ways, depending on the value of the random bit. But not more. The number of possible behaviours depends on the number of bits requested as $n = 2^k$, where $k$ is the number of random bits requested. There seems to be some entropy economy here: we simply don't get more entropy out than goes in, in case of the deterministic computer attached to a random source.

Practially, it seems to be a good advice to carefully assess and measure entropy available in a RNG as well as to use different, independent sources to get the entropy from. Assuming total traffic control, it might be reasonable to encrypt random data while in flight.

Good random is about entropy management. Once entropy is available it can readily be consumed by standard symmetric ciphers to produce random bits. It should be clear that if an attacker has full control of your entropy sources, you might loose all security.

I want to close with an important fact: random sources are not computers, they are rather complementary. While a computer is a deterministic system, the random source is the very opposite. Its output is expected to be completely *nondeterministic*, which means it's not forseeable and there is no program or simulation that could reveal what the future output could be. Any true random source must be based on overservation of some physical phenomena in a stochastic system, such as radioactive decay, noise or human interactions.

# 7 Advanced Applications

Users of X.509 and CMS libraries might be disappointed by the straight forward approach used there. In fact, those libraries reduce cryptography to some very basic objects like certificates, signed and encrypted messages. With Diffie Hellman we can do much more, but it's not simple. An obvious way to share a secret between a group of people with key pairs $(a_i, A_i)$ is:

$$s = G \cdot \prod a_i = G \cdot a_j \cdot \prod^{k \neq j} a_k$$

The first user sends $a_1 G$ to the second user, who calculates $a_2 a_1 G$ and sends that to the third user who calculates $a_3 a_2 a_1 G$ and so on until we get the shared secret $s$, analogous to the case with two users. Oh, before I forget, check $a_2 a_1 G$ being sent from the second to the third user is in fact the shared secret of the first and the second user!

So uups, what does this tell us? We must be very carefull about how things work to avoid producing leaking systems. This is not the case when we just define the intermediate results worthless and public, but there is some potential for confusion, especially when somebody is used to RSA. If the same key pair would be used for different groups, this simple shared secret protocol would spill a lot of secrets.

Let's look at a method to create a signature with multiple keys. It is not a multi-signer scheme, only a simple example! But it could be useful in situations where a static key is used in combination with a temporary key to ensure that both are related in a session. Given a set of key pairs, we choose a $k$ and calculate:

$$r = kG,\ s = k^{-1}(z + a_1 r + a_2 r + ...)$$

The signature is verified by

$$r = s^{-1}(zG + rA_1 + rA_2 + ...) = s^{-1}(z + a_1 r + a_2 r + ..)G = kG$$

Lets revisit the discussion of $k$ reuse, which is fatal. Is it a problem when z is predictable or reused? It hasn't been mentioned anywhere, so, we assume for brevity $z = 0$. Then we have:

$$s = xr/k \Rightarrow x = sk/r = skr^{-1}$$

Since we don't know either $k$ or $r$ finding $kr^{-1}$ is impossible, the key is not exposed. We can subsequently ignore z in our considerations, since all must hold for $z = 0$ too.

Obviously, operations involving private keys and random values must be constructed so that there is some mathematical protection of these values.

What happens, for example, if we take a simpler formula, such as $s = zk + a$, the one from the implicit CA? I don't see a way to do a construction similar to the one above. Ok. What about this: $s = k^{-1}a$? When we put this into the verification formula,

$$r = s^{-1}A = ka^{-1}aG = kG$$

it obviously holds, $a = sk$ clearly shows what we already know: exposing $k$ is fatal. But if $k$ is unknown, the key is still protected. The crucial part seems to be the inclusion of $k^{-1}$ into $s$. If we define, more generally

$$s = k^{-1}f(z, r, a)$$

and try to insert something suitable into the verifier, based on two functions of $z$ and $r$, the values available.

$$r = s^{-1}(\tilde{f}_0(z, r) + a\tilde{f}_1(z, r))G = kG$$

we require that

$$\tilde{f}_0(z, r) + a\tilde{f}_1(z, r) = f(z, r, a)$$

for the verification condition to hold. This covers the DSA case with

$$\tilde{f}_0(z, r) = z,\ \tilde{f}_1(z, r) = r$$

It seems better to connect the random component $r$ with the private key via $\tilde{f}_2$ than $z$, which may be influenced from outside. We can safely tweak to $s$ value by including a term, such as $s = z + \tau + ar$. If the tweak is a shared secret, the signature can only be validated by the those who share it. There is also no theoretical reason for using $r = kG$ instead of any other random value, it's merely an optimization to require less random and less I/O.

For the simple multikey signature sample using a shared random $z$ we can formulate:

$$s = k^{-1}f(z, r, a_1, a_2, ...) = k^{-1}(\tilde{f}_0(z, r) + a_1\tilde{f}_1(z, r) + a_2\tilde{f}_2(z, r) + ...)$$

If we assume that all $\tilde{f}_i,\ i > 0$ are the same function, does this affect security somehow? From above considerations I assume that the $r$ factor is not relevant and $\tilde{f}_{i>0} = 1$ should do as well. Determining the terms of a sum gives infinitely many solutions, so it will not make things worse to add multiple private keys together.

# 8 Dependencies

Lets have a deeper look at the multikey signature scheme by asking what we can say about the time a particular datum was created. The *proof of knowledge* algorithm used for authentication above is a method to ensure that Alice knows a particular secret, without revealing it. By choosing a random challenge, Bob can ensure that the response sent by Alice is not a replay. The actual process of authentication, the proof, is physically acomplished by entangling the two secret values in a irreversible calculation.

Given private key $x$ the public key is $X = xG$, what is $x \cdot X = x^2G$? And what about $(x + \tau) \cdot X = (x^2 + \tau x)G$? Lets consider the order of operations in the calculation: we can assume $x$ and $X$ to already exist, obviously also $x^2G$. Lets call $\tau$ fresh and an expression that involves a fresh term or factor also fresh. So in the second expression we could say $x + \tau$ is fresh. The idea behind freshness is that $\tau$ is possibly a challenge value coming from another party, so a proof containing $\tau$ and $x$ bound together ensures that $x$ was known after $\tau$, so it's indeed fresh itself. Obviously the expression $(x + \tau)G$ is also $\tau$-fresh, there's no point in squaring the secret value.

What's the point in freshness? When designing a system, consider attacks. Things may be *foul*: random values less random than they look or choosen intentionally to weaken the system, compromised keys etc. It is certainly necessary to consider the behaviour of the system in scenarios like partical key compromise to understand how much other information is possibly leaked. Could the leaking of $k$ of a multikey signature reveal all the private keys? Can this be prevented by suitable choice of calculations?

Besides that, the freshness concept can help understand the effect of replay attacks in protocols. Generally, every party must ensure freshness of external contributions to a procedure by injecting it's own $\tau$ into the process, such as an emphemeral shared key.

# 9 The Trap

We introduce a new notation $\lambda(qG) = \lambda(Q) = \lambda Q$ to denote the conversion of some point on the elliptic curve in to an element of $\mathbb{Z}_q$. This is a mapping $\mathbb{F}_p^2 \Rightarrow \mathbb{Z}_q$, but the y coordinate is typically left out. From the curve equation it follows that there are at most two $y$ for each $x$, so this should not matter to much. The signature function $f : s = f(z, k, x)$ involves the mapped values $(\lambda kG, \lambda xG)$ to ensure that the signature is bound to the corresponding private knowledge. Using the notation

$$r = \lambda kG,\ s = z + ar = z + a \cdot \lambda kG$$

the relations between the secret values (small letters) and derived public values (the $\lambda$'s) gets more visible.

Lets formulate another generalization of the joint signature operation of message $z$ using variables $c_0, c_1, c_2, ...$ where $c_0 = z$, $c_1 = k$ and all other $c_j = a_j$, $j > 1$ are private keys. The general signature function is $f(c_0, c_1, c_2, ...)$ with an additional requirement that it must have the form

$$f(c_0, c_1, ...) = \bar{f}(c_0, c_1, ..., \lambda c_0 G, \lambda c_1 G, ...)$$

Since $z = c_1$ is an external value and not a curve point, we assume $\lambda z = z$. This could be written in matrix form as

$$f(\mathbf{c}) = \bar{f}(\mathbf{c}, \lambda \mathbf{c} G)$$

Combining the $\lambda$'s of the public values into the result of the $f$ binds the signature to the private keys, this is the essential ingredient that makes the security. Remember that we need a function that gives us the factor for every public key used in the verification process without knowing the private values

$$s = k^{-1} f(c_0, c_1, ...) = k^{-1} \sum \tilde{f}_i(\lambda c_0 G, \lambda c_1 G, ...) c_i = k^{-1} \sum \tilde{f}_i(\lambda \mathbf{c} G) c_i$$

If we assume the $\tilde{f}_i$ are linear combinations of the inputs, we can write the coefficients as a matrix $\tilde{\mathbf{F}}$ with elements $f_{ij}$ acting on vector $\lambda \mathbf{c} G$.

$$\tilde{f}_i(\lambda \mathbf{c} G) = f_{i,0} \cdot \lambda c_0 G + f_{i,1} \cdot \lambda c_1 G + ...$$

To calculate the signature $f$, we use the same coefficients:

$$f(c_0, c_1, ...) = \sum_i \tilde{f}_i(\lambda \mathbf{c} G) c_i = \sum_i \sum_j f_{ij} \cdot \lambda c_j G \cdot c_i$$

Now how does this look for the DSA signature? If we transfer $f = z + ar$ into the new notation we get:

$$\begin{bmatrix} 1 & \lambda k G & \lambda a G \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{c} = \begin{bmatrix} 1 & 0 & \lambda k G \end{bmatrix} \cdot \mathbf{c} = \begin{bmatrix} 1 & 0 & \lambda k G \end{bmatrix} \cdot \begin{bmatrix} z \\ k \\ a \end{bmatrix} = z + a \cdot \lambda k G = z + ar$$

Written more formally it looks like

$$f(\mathbf{c}) = \lambda (\mathbf{c} G)^T \cdot \mathbf{F} \cdot \mathbf{c}$$

The choice of coefficients determines how much information gets mixed up. By choosing a suitable $\mathbf{F}$ it is possible to calculate the verification value so that it depends on all keys in a way that proves that all shared secrets where known at the same time. Since we have $k^{-1}$ as a factor of $s$, it is ensured that the secret was calculated after $k$ was generated. The question is what can we say about the components of $\tilde{f}$? If we choose some coefficient zero? Then there is no relation between one of the public keys and one of the private keys used to sign the message. This means theoretically that value could have been precalculated.

# 10 Authentication

The most prominent reason for having cryptography is authentication. No authentication works without crypto today, but there are different levels, ranging from TLS encrypted plaintext password to elaborate key distribution architectures.

The purpose of authentication can easily be seen from the Diffie Hellman key exchange. There is no protection from the *man in the middle* attack. Some additional mechanism is required. The purpose of authentication is to ensure that communicating parties are talking to who they are expecting to talk. Typically, a central entity controls the relation between keys, identifiers and credentials.

Lets look at a simple authentication protocol. We assume that all parties have key pairs and corresponding certificate $\mathbf{C}_q$ of whatever format stating the validity of some $Q = qG$. Further we imply that $\mathbf{C}_q$ contains $Q$, so we can write it instead of a public key or omit the public key in messages where the certificate appears.

1. Alice invents $r_1$, calculates $R_1 = r_1 G$ and sends that to Bob.
2. Bob invents $R_2 = r_2 G$.
3. Bob calculates $s_2 = h(\mathbf{C}_b) + \lambda R_1 + (b + r_2)\lambda R_2$.
4. Bob sends his public key $B$ together with $R_2$, $s_2$ and the certificate $\mathbf{C}_b$.
5. Bob derives the shared secret $s_{eph} = r_2 R_1$.
6. Alice validates $[h(\mathbf{C}_b) + \lambda R_1]G + \lambda R_2 B + \lambda R_2 R_2 = s_2 G$.
7. Alice validates the certificate $\mathbf{C}_b$.
8. Alice calculates $s_3 = s_2 + h(\mathbf{C}_a) + \lambda R_2 + (a + r_1)\lambda R_1$.
9. Alice derives the emphemeral shared secret $s_{eph} = r_1 R_2$.
10. Alice sends her public key $A$, $s_3$ and her certificate $\mathbf{C}_a$ as payload.
11. Bob validates $[s_2 + h(\mathbf{C}_a) + \lambda R_2]G + \lambda R_1 A + \lambda R_1 R_1 = s_3 G$.
12. Bob validates the certificate $\mathbf{C}_a$.

## 10.1 Safety

So now how could we judge that? Lets draw it more nicely in terms of messages transmitted between Alice and Bob.

$m_1 = R_1$
$m_2 = R_2, s_2, \mathbf{C}_b$
$m_3 = s_3, \mathbf{C}_a$

We notice that variable $r_1$ and $r_2$ are ephemeral and so is the resulting secret $s_{eph}$. In the commitment message $m_1$ Alice does not reveal any information about herself. In this protocol, it's the server to expose his identity first, by supplying a certificate $\mathbf{C}_b$ for this public key $bG$.

Messages $m_2$ and $m_3$ are signed, $m_2$ is signed by $b$ and $r_2$, while $m_3$ is signed by $a$ and $r_1$. Both signatures require the knowledge of two keys to be generated. One of these keys is static while the other is emphemeral. So, unless data is intentionally captured, the signature will loose its value once the emphemeral key expires.

The certificates mentioned in this protocol can be of common X.509 type or be based on an implict scheme, as described above. If we assume certificates are safe, then the protocol is safe against MITM attacks.

### 10.2 Attack

Lets see what happens if adversary Eve tries to interfer: she catches message $m_1$ to replace $R_1$ with $\tilde{R}_1 = \tilde{r}_1 G$ and forwards it to Bob. Bob then calculates $s_2 = \lambda\tilde{R}_1 + b\lambda R_2 + r_2\lambda R_2$ using the forged value. Can Eve possibly use $s_2$ to calculate some other signature? Lets put $\tilde{s}_2 = s_2 + \alpha$ for some chosen $\alpha$. We want to exploit the signature to replace the emphemeral key established with Alice and Bob. Now Alice sees a valid certificate for $bG$ and modified $\tilde{R}_2 = \tilde{r_2}G$ and signature $\tilde{s}_2$. Does the signature match? Can Eve make it match by choosing a suitable $\alpha$? Alice receives from Eve:

$$\begin{aligned}
\tilde{s}_2 G = s_2 G + \alpha G &= \lambda R_1 G + \lambda\tilde{R}_2 B + \lambda\tilde{R}_2\tilde{R}_2 \\
\lambda\tilde{R}_1 + (b + r_2)\lambda R_2 + \alpha &= \lambda R_1 + (b + \tilde{r}_2)\lambda\tilde{R}_2 \\
\alpha &= \lambda R_1 - \lambda\tilde{R}_1 + (b + \tilde{r}_2)\lambda\tilde{R}_2 - (b + r_2)\lambda R_2 \\
\alpha &= [\lambda R_1 - \lambda\tilde{R}_1] + b[\lambda\tilde{R}_2 - \lambda R_2] + \tilde{r}_2\lambda\tilde{R}_2 - r_2\lambda R_2
\end{aligned}$$

This seems not easy to solve for unknown $b$ and $r_2$, which Eve does not have. This is no prove or something, just one possible attack that does not seem to work.

Take a closer look at the expression $s_2 = \lambda R_1 + b\lambda R_2 + r_2\lambda R_2$. We know that $\lambda qG$ is $q$-fresh, so $s_2$ is $(r_1, r_2, b)$-fresh. For $\bar{s}_3 = h(\mathbf{C}_a) + \lambda R_2 + (a + r_1)\lambda R_1$ we conclude that it is $(r_1, r_2, a)$-fresh with no contribution from $b$, that's why in the protocol $s_2$ is added, making the expression $(r_1, r_2, a, b)$-fresh. Without $b$-freshness, knowing $r_2$ might be helpful for Eve's activites, even if she does not know $b$. If $r_2$ is an emphemeral key, it might well be generated under different circumstances than static key $b$, so this could matter.

We close with the conclusion that an authentication protocol is well feasible under DLP. It leaves us with an ephemeral key authenticated by a trusted certificate and the emphemeral key itself. It requires no more than two random values.

## 11 Signatures

Almost any reasonable application of digital signatures implies that signed messages are signed again, including the original signature. Is there any way this can be accomplished without just wrapping one thing into another? Consider implicit certificates mentioned earlier: we expect a chain of certificates, starting with a root, followed by intermediaries and the subject certificate at the top (or bottom). How does this match with implicit certificates?

### 11.1 Certificate Chains

I'm sorry to say that here is a jump to something new, called ASN.1, which I'm not going to explain. The reason I use it is that X.509 uses it too. Together with DER encoding, specified in X.690, it is a compact and robust serialization format suited for cryptography. So let, in ASN.1, certificates look like this, for example:

```
Certificate ::= SEQUENCE OF SEQUENCE {
       name UTF8String,
       p ECPoint,
       constraints OPTIONAL SEQUENCE OF Constraint,
       accept OPTIONAL Signature
}
```

Please note that the sequence of a sequence is an ordered list, or simply an array. Assume that we have a CA with key pair $(c, cG)$ then in the first entry `p[0]` $= c_0 G$, whatever `name` and optionals omitted. In the second entry let `p[1]` $= \gamma = (\alpha_1 + k_1)G$, the implicit certificate generated by the CA for `name[1]`.

| level | name | p | public key | private key |
|---|---|---|---|---|
| 0 | root.acme.org | $c_0G$ | $c_0G$ | $c_0$ |
| 1 | ca.acme.org | $\gamma_1 = (\alpha_1 + k_1)G$ | $C_1 = e\gamma_1 + c_0G$ | $c_1 = e\alpha_1 + s_1$ |
| 2 | user.acme.org | $\gamma_2 = (\alpha_2 + k_2)G$ | $C_2 = e\gamma_2 + c_1C_1$ | $c_2 = e\alpha_2 + s_2$ |

In the table it's shown how the public key for the leaf is calculated from the data in the certificate. We remember that $s_i = e_i k_i + c_{i-1},\ i > 0$, where $e_i = h(\gamma, \texttt{name}, \texttt{constraints}, ...)$, a hash function over some serialization of the certificate elements.

So now, how safe is this? The certificate requestor can certainly validate $C_i = c_iG$, but what about the public key calculate by other users? If we'd modify $\gamma_2$, for example, how could this be detected? That's where the `accept` fields comes into play. The accept value is a signature calculated with the private key $c_j$ of the certificate subject, based on the same $e_i$ as was used for the $\gamma_i$. Obviously the accept is required only on leaf level, but may appear on intermediate levels as well.

It is interesting to note, that in contrast to X.509 based certificate issuing, there is an additional step. The user receiving the certificate must create the accept signature before the certificate becomes valid. This gives the scheme an advantage over the X.509 hierarchy: The CA cannot issue certificates without the users consent. Besides that certificates are significantly smaller.

My main reason for using the ASN.1 language to describe data structures is that it can readily be applied to code. If we define

```
ECPoint ::= CHOICE {
       compressedPoint OCTET STRING,
       point SEQUENCE { x INTEGER, y INTEGER } }

Signature ::= SEQUENCE { r INTEGER, s INTEGER }
```

and some basic constraints

```
ExpirationConstraint ::= [0] UTCTime
NameSuffixConstraint ::= [1] UTF8String
```

we can already reproduce some of the basic features of other certificate schemes. Provided libraries for EC and DER the implementation is straight forward.

### 11.2 Group Signature

A group signature is similar to a voting process. In the first step, the group member send their commitments to a coordinator. The coordinator calculates a group public key and a challenge from the commitments. Each member then sends its proof to the coordinator who sums ...

Let $n$ members $a_1, a_2, ...$ of a group create ephemeral public keys $R_j = r_jG$. A coordinator calculates the shared commitment $R = \sum R_j$, derives a challenge c from it and sends it to each member, who calculates $s_j = c(a_j + r_j)$. The coordinator then collects

$$s = \sum s_j = \sum c(a_j + r_j) = c\sum a_j + r_j$$

$$r = \lambda[c(A + R)]$$

where $A = \sum A_j$ is the common shared public key. The verification then goes like:

$$sG = c\sum(a_j + r_j)G = c\sum A_j + R_j = cA + cR$$

## 12 Password based Cryptography

One might think that passwords are somehow outdated. But in fact we don't have many replacements for them yet for general use. So it's likely passwords will continue to play a role. How unsafe are passwords?

Assume the device capturing the password is not compromised by troyans or keyloggers to be optimistic. If we send a password over TLS to a server in plaintext, it's certainly unclear what it will do with it and we must consider the security of that password undefined, unless we examinate the server. But if we keep passwords in the context of the device, it might still be pretty safe, because the device can launch attack countermeasures like deleteing the keys after a number of failed password entry attempts, for example.

In the following discussion, we assume that the the password entry device PED is not cheating. The focus lies on how keys can safely be derived from passwords. The first step is to use an good keyword derivation function, such as SCRYPT. For our purposes, a key derivation function KDF shall be defined as a function $\Psi(p,t)$ of the password $p$ and a tweak factor $t$. To keep things simple, assume all values, except the password, have the same bit-length like the underlying DLP structure.

To protect private key $x$ with a password $p$, we can calculate $\tilde{x} = x \otimes \Psi(p, h(x))$ where $h$ is a hash function and $\otimes$ is the bitwise XOR operation. Now we can store $(h(x), \tilde{x})$ on the device in an unprotected location, such as the file system and reproduce $x = \tilde{x} \otimes \Psi(p, h(x))$ whenever needed. The input to the hash function can include additional data, such as user identification $id_U$.

Another possibility is to derive key values directly from the output as $x = \Psi(p, h(id_U, ...))$. From the entropy standpoint this is surely worse regarding the quality of keys generated. A key generated in this way may contain significantly less then required entropy (even if hashed). My opinion is that key values should always be generated from true random. The safety of a key stored under a password depends on the strength of the password in first line and there is no need to compromise the key with the password as long as the device has the ability to store $\tilde{x}$.

Using the first method the device to drop information occasionally, after a number of wrong password inputs, for example, and it's compatible with the implicit certificate scheme.

## 12.1 Activation Procedure

The purpose of the activation procedure is to establish a secret key associated with a user identifier on a device. The key shall be protected by a password, as described above. Once activated, the device can use the key to authenticate against other devices or services that support a common CA. The input for the activation procedure is a user identifier and the network address of the common CA.

1. Alice, the subscriber generates $r_1$ and sends $R_1 = r_1 G = \alpha G$ to the CA.
2. The CA generates $R_2 = r_2 G$.
3. The CA calculates the shared secret $s_{eph} = r_2 R_1$.
4. The CA calculates $s_2 = h(\mathbf{C}_c) + \lambda R_1 + (c + r_2)\lambda R_2$.
5. The CA sends $R_2$, $s_2$ and her certificate $\mathbf{C}_c$ to Alice.
6. Alice validates $s_2 G = [h(\mathbf{C}_c) + \lambda R_1]G + \lambda R_2 R_2 + \lambda R_2 C$.
7. Alice validates the certificate $\mathbf{C}_c$.
8. Alice calculates shared secret $s_{eph} = r_1 R_R$ and derives a symmetric encryption key $\omega$.

   At this point, Alice has authenticated, fresh information about who the CA is but has not revealed anything about herself yet. Now Alice shall request activation of by stating a user identifier $id_A$.
9. Alice generates random $R_3 = r_3 G$.
10. Alice encrypts $e_3 = \mathbf{ENC}_\omega(id_A)$.
11. Alice calculates $s_3 = h(id_A) + e_3 \lambda R_1 + r_1 \lambda R_1 + \lambda R_2 + r_3 \lambda R_3$.
12. Alice sends $(R_3, e_3, s_3)$ to the CA.
13. CA validates $s_3$.
14. CA decrypts $id_A = \mathbf{ENC}_\omega^{-1}(e_3)$.

15. CA generates a random activation code $\sigma$ of desired bit length.
16. CA calculates $V_1 = (s_{eph} + h(\sigma))$ G.
17. CA sends $V_1$ to Alice.
18. CA sends $\sigma$ to Alice on an alternative route (such as email or SMS).

    While the activation code $\sigma$ is in flight, it might be captured, it's not a secure channel. Often $\sigma$ should not contain too many digits to remain practical.
19. Alice receives $\sigma$ and validates $[s_{eph} + h(\sigma)]G = V_1$.
20. Alice invents $R_5 = r_5 G$.
21. Alice calculates $s_5 = r_1 + r_5 + s_{eph} + h(\sigma)$
22. Alice sends $R_5, s_5$.
23. CA validates $s_5 G = R_1 + R_5 + (s_{eph} + h(\sigma))G$.
24. CA invents $R_6 = r_6 G$.
25. CA calculates the implicit certificate $\gamma = R_5 + R_6$.
26. CA calculates $e_6 = h(\gamma, id_A, ...)$.
27. CA calculates $s_6 = e_6 r_6 + c$.
28. CA prepares the certificate $\mathbf{C}_a$, the public key is $A = e_6\gamma + C$.
29. Alice calculates $a = e_6 r_5 + s_6$ to get $A = aG$.
30. Alice invents $R_7 = r_7 G$.
31. Alice creates signature $s_7 = h(\mathbf{C}_a) + a\lambda R_7$ and fills the accept field in the certificate.
32. Alice sends the completed certificate to CA.
33. CA registers the certificate, the process is completed.