Diffie-Hellman Key Agreement Method Summary peter-thoemmes.org research

© Peter Thoemmes
Weinbergstrasse 3a
D-54441 Ockfen, Germany

December 31, 2011

Abstract

This paper is a summary of the brilliant algorithm, based on the ideas of a research project proposal CS244 of **Ralph C. Merkle** (Puzzles) from 1974 and the work of **Whitfield Diffie** and **Martin E. Hellman**. In 1976 the Diffie-Hellmann Key Agreement Method was published and in June 1999 it became RFC 2631. This paper is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY, without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

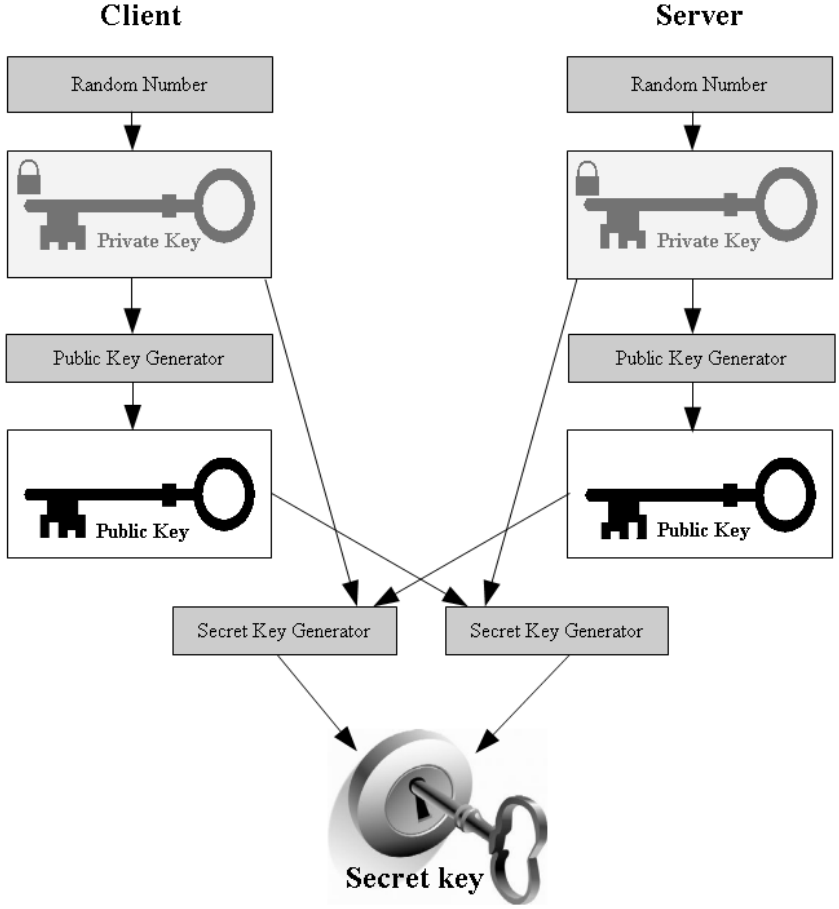
Contents

1 General Stuff

In the 1970's **Whitfield Diffie**, **Martin E. Hellman** and Hellman's Student **Ralph C. Merkle** together worked on public key cryptography. Starting from an idea of Hellman's Student **Ralph C. Merkle**, known as Merkle's Puzzles (Fall, 1974), they developed the Diffie-Hellmann Key Agreement Method, published in 1976. Finally, in June 1999, the results were published in RFC 2631. Based on that publication, Ron Rivest, Adi Shamir und Leonard Adleman published the **RSA** algorithm in August 1977. RSA stands for the names Rivest, Shamir und Adleman.

Using this method makes it possible, that never the actual (secret) encryption/decryption key is transmitted over the insecure channel between two users who want to exchange secret data. But still, after an initial 'handshake', both users know the secret key to use for encryption and decryption. To do so, both users do have to create a private key before, which actually are random numbers. The server defines the domain parameters and provides a public key calculated from his private key using that parameters. The client, after receiving the domain parameters and the server's public key, calculates the secret key from his private key and the server's public key. Then the client calculates a public key from his private key using the domain parameters and sends it back to the server. This enables the server to also calculate the secret key.

Figure 1: Diffie-Hellmann Key Agreement Method



2 DH: Using an Exponential Function

Following equation plays a central role in the Diffie-Hellman Key Agreement Method:

$$y = g^m \bmod n = (g \bmod n)^m \bmod n$$

$$\text{with } g, m, n, y \in \mathbb{N}^+$$

This operation is discrete ($g, m, n, y \in \mathbb{N}^+$) and cyclic (mod n). It is used in a special way: n is defined as a **prime** p and g must be smaller then p but bigger then 1 (as otherwise any power of g would remain 1):

$$n := p$$

$$g \in [2, p-1]$$

As p is a prime, it can never be a power of g, and so the modulo operation will result in following range of natural numbers (RFC 2631 show how to generate a valid p and a valid g):

$$(g^m \bmod p) \in [2, p-1]$$

Now following definitions are done:

Private key of user A (server, agent):

$$k_{priv,A} := a$$

Private key of user B (client):

$$k_{priv,B} := b$$

Actual **secret encryption/decryption key** for both, user A and user B:

$$k := g^{ab} \bmod p$$

That means that we have following situation:

User A (server, agent) defines:	g, p and a
User B (client) defines:	b

So nobody knows the actual encryption/decryption key at this moment, as it consists of g, p and both private keys a and b.

The challenge now is how to make the encryption key known to both without sending it and also without sending the private keys. The solution is given by the equation mentioned above:

$$k = g^{ab} \bmod p = (g^a \bmod p)^b \bmod p = (g^b \bmod p)^a \bmod p$$

Based on that following procedure is implemented:

User A generates a public key:

$$k_{pub,A} := g^a \bmod p$$

The User A sends g, p and $k_{pub,A}$ to user B. User B now has enough information to generate the actual symmetric encryption/decryption key:

$$k = (g^a \bmod p)^b \bmod p = (k_{pub,A})^b \bmod p$$

User B also generates a public key:

$$k_{pub,B} := g^b \bmod p$$

Now user B sends back his public key $k_{pub,B}$ and so enables user A to also calculate the actual symmetric encryption/decryption key:

$$k = (g^b \bmod p)^a \bmod p = (k_{pub,B})^a \bmod p$$

Done! Now encrypted messages can be exchanged between user A and user B.

It depends on the prime p how many possible values the encryption key can have, as the range of the key values is $[2, p-1]$:

$$k = g^{ab} \bmod p \in [2, p-1]$$

An example for a prime p is following 1024 bit length value (RFC 2539):

$$p = 2^{1024} - 2^{960} - 1 + 2^{64} * ((2^{894} * pi) + 129093)$$

This results in a 309 digit length decimal number:

```
179769313486231590770839156793787453197860296048756011706444
423684197180216158519368947833795864925541502180565485980503
646440548199239100050792877003355816639229553136239076508735
759914822574862575007425302077447712589550957937778424442426
617334727629299387668709205606050270810842907692932019128194
467627007
```

Now we should have a look at an **example**. Imagine we would define following domain parameters p and g and the 2 private keys a and b:

p	23
g	5
a	6
b	15

User A (server, agent) would compute following public key:

$$k_{pub,A} := 5^6 \bmod 23 = 15625 \bmod 23 = 8$$

User B (client), after receiving $k_{pub,A}$, would compute the secret key:

$$k = 8^{15} \bmod 23 = 35184372088832 \bmod 23 = 2$$

User B would also compute his public key:

$$k_{pub,B} := 5^{15} \bmod 23 = 30517578125 \bmod 23 = 19$$

User A, after receiving user B's public key, would compute the secret key:

$$k = 19^6 \bmod 23 = 47045881 \bmod 23 = 2$$

So if an **attacker** tries to get the private key from user A's published unencrypted information g, p and $k_{pub,A}$, he has to solve following problem: The private key of user A is the logarithm to the base value $g = 5$ resulting in anything that has a rest of $k_{pub,A} = 8$ when dividing by $p = 23$. So possible values for $y = g^x$ are

$$y_i = i \cdot 23 + 8 \text{ with } i \in \{0, 1, 2, ...\}$$

That means the attacker has to test 8, 31, 54, 77, ..., and so on. Getting the logarithm to a base other then the e (ln) is possible by following equation:

$$x = log_g(y) = \frac{ln(y)}{ln(g)}$$

So here the possible endless many keys:

$log_5(8)$	1.29
$log_5(31)$	2.13
$log_5(77)$	2.69
...	...
$log_5(15625)$	6
...	...

For every value the attacker needs to build the secret key by help of the server's public key and he needs an encrypted probe of the communication to test if decryption using that secret key results in something meaningful.

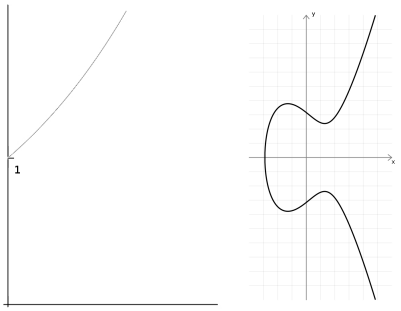
3 ECDH: Using an Elliptic Curve

Instead using the discrete exponential function $y = g^x \bmod p$, it is more secure to use an elliptic curve (first proposed in 1986 by **V. Miller** and independently also by **N. Koblitz**), which can be described by the **Weierstrass equation** in normal form:

$$y^2 = x^3 + cx + d$$

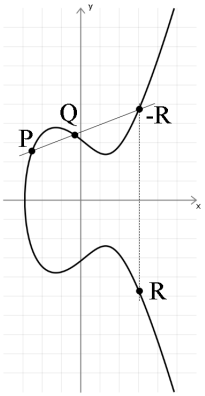
with $c, d \in \mathbb{N}^+$

Figure 2: Exponential Function and Elliptic Curve



The **Group Law** for the elliptic curve says, that if $\vec{P}$ and $\vec{Q}$ are 2 points on the curve, then following operation $\oplus$ (**point addition**) is defined:

Figure 3: The Elliptic Curve Point Addition

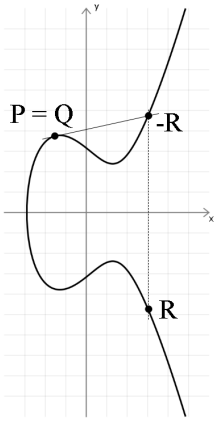


$$\vec{R} := \vec{P} \oplus \vec{Q}$$

That means a third point $\vec{R}$ on the curve is defined by this operation. It can be found by the intersection of the curve with the line through $\vec{P}$ and $\vec{Q}$ (which actually is $-\vec{R}$).

In case $\vec{P} = \vec{Q}$ we will get to point doubling:

Figure 4: The Elliptic Curve Point Doubling



$$\vec{R} = \vec{P} \oplus \vec{P} := 2\vec{P}$$

If the slope s_P of the crossing line (point addition) or tangent (point doubling) $\vec{P}$ moves to infinity (turns vertical), we would run into a singularity, meaning into a point where the operation is not defined. To get out of this, we define the **point at infinity**:

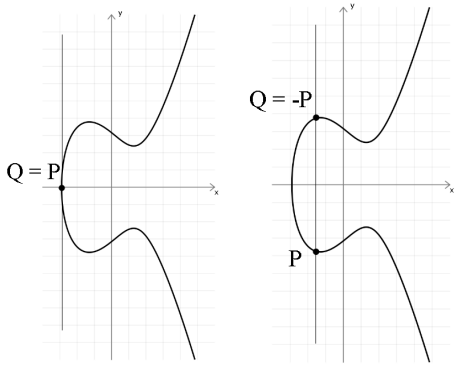
$$\lim_{s_P \to \infty} \vec{R} := \vec{P}_\infty$$

Then we define for point addition and point doubling the operation at that point at infinity such, that the point is **neutral to the operation**. To make all points on the curve being an **abelian group** (German: Abelsche Gruppe) concerning the $\oplus$ operation (meaning the operation is commutative), we define commutativity at the same moment:

$$\vec{Q} = -\vec{P}\text{:} \quad \boxed{\vec{P} \oplus -\vec{P} := -\vec{P} \oplus \vec{P} := \vec{P}_\infty} \tag{1}$$

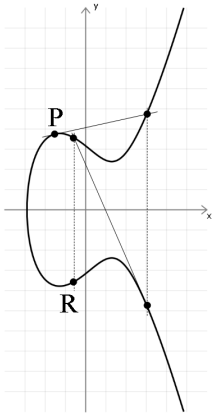
$$\vec{Q} = \vec{P}\text{:} \quad \boxed{\vec{P} \oplus \vec{P} := \vec{P}_\infty} \tag{2}$$

Figure 5: Point at Infinity



To get to the **3 times point addition** we simply repeat the operation:

Figure 6: The Elliptic Curve 3 Times Point Addition



$$\vec{R} = 3\vec{P} = \vec{P} \oplus \vec{P} \oplus \vec{P}$$

In general the definition of **n times point addition** is as follows:

$$\vec{R} = n\vec{P} := \underbrace{\vec{P} \oplus \vec{P} \oplus ... \oplus \vec{P}}_{n}$$

This enables us to fully geometrically describe the $\oplus$ operation on the elliptic curve. To add a point to itself n times we perform **one time point doubling** and **(n - 2) times point addition**:

$$\vec{R}_2 = 2\vec{P} = \vec{P} \oplus \vec{P}$$

$$\vec{R}_3 = 3\vec{P} = \vec{R}_2 \oplus \vec{P}$$

$$...$$

$$\vec{R}_n = n\vec{P} = \vec{R}_{n-1} \oplus \vec{P}$$

To compute this operation we first determine the **slope** of the straight line. We have 2 different cases here: First we have **point doubling**, where the straight is the **tangent** of the elliptic curve and so the slope the **first derivation** of the curve. This is always our starting operation. Then we have a series of **point additions**, where the straight line scrosses both points.

For the starting operation (**point doubling**) we get the slope from the **first derivation** of the curve:

$$s = \frac{dy}{dx} = \frac{d}{dx}(x^3 + cx + d)^{\frac{1}{2}}$$

$$\rightarrow s = \frac{1}{2} \cdot (x^3 + cx + d)^{\frac{-1}{2}} \cdot (3x^2 + c) = \frac{3x^2+c}{2 \cdot \sqrt{x^3+cx+d}} = \frac{3x^2+c}{2 \cdot y(x)}$$

$$\rightarrow s_P = \frac{3x_P^2+c}{2 \cdot y_P}$$

For **point addition** we get the slope from the 2 intersection points $\vec{P}$ and $\vec{Q}$:

$$s_P = \frac{\Delta y}{\Delta x} = \frac{y_Q - y_P}{x_Q - x_P}$$

With that results and the base point $\vec{P} = (x_P, y_P)$ we get following y value for the **straight line**:

$$-y = s_P \cdot (x - x_P) + y_P$$

$$\boxed{y = s_P(x_P - x) - y_P} \tag{3}$$

$$\text{with } s_P = \begin{cases} \frac{3x_P^2+c}{2 \cdot y_P}, & \text{if } \vec{P} = \vec{Q} \\ \frac{y_Q - y_P}{x_Q - x_P}, & \text{if } \vec{P} \neq \vec{Q} \end{cases} \tag{4}$$

Now we bring that equation into normal form:

$$y = (-s_P)x + (s_P x_P - y_P)$$

$$\rightarrow y^2 = (s_P^2)x^2 + (2s_P(y_P - s_P x_P))x + (y_P - s_P x_P)^2$$

$$\boxed{y^2 = (s_P^2)x^2 + (2s_P(y_P - s_P x_P))x + (y_P - s_P x_P)^2} \tag{5}$$

At the intersection points the result is equal to the result of the Weierstrass equation of the **elliptic curve**:

$$\boxed{y^2 = x^3 + cx + d} \tag{6}$$

So subtracting equation (5) from equation (6) must **result in 0 at the points of intersection of the straight line and the elliptic curve**:

$$(5) - (6) = 0$$

$$\boxed{x^3 + (-s_P^2)x^2 + (c - 2s_P(y_P - s_P x_P))x + (d - (y_P - s_P x_P)^2) = 0} \tag{7}$$

This is a cubic equation

$$\boxed{x^3 + a_2 x^2 + a_1 x + a_0 = 0} \tag{8}$$

with the coefficients:

$$a_2 = -s_P^2$$

$$a_1 = c - 2s_P(y_P - s_P x_P)$$

$$a_0 = d - (y_P - s_P x_P)^2$$

It can be solved (first published by Cardano in 1545) by following substitution:

$$\boxed{x := z - \frac{a_2}{3}} \tag{9}$$

$$\rightarrow z^3 + pz + q$$

with the coefficients:

$$p = \tfrac{3a_1 - a_2^2}{3}$$

$$q = \tfrac{2a_2^3}{27} - \tfrac{a_2 a_1}{3} + a_0$$

Cardano found, that the discriminant D determines of which type the 3 results of that equation are:

$$D := \tfrac{q}{2}^2 + \tfrac{p}{3}^3$$

$D > 0$	1 real and 2 complex conjugate results
$D = 0$	3 real results, min. 2 are equal (for p = q = 0 all are equal)
$D < 0$	3 different real results

The solutions for $D > 0$ (1 real and 2 complex conjugate results):

$$u := \sqrt[3]{\frac{-q}{2} + \sqrt{D}}$$

$$v := \sqrt[3]{\frac{-q}{2} - \sqrt{D}} = \frac{p}{3u}$$

$$z_1 = u + v$$

$$z_2 = -\frac{u+v}{2} + j\sqrt{3}\frac{u-v}{2}$$

$$z_3 = -\frac{u+v}{2} - j\sqrt{3}\frac{u-v}{2}$$

The solutions for $D = 0$ (3 real results, min. 2 are equal):

$$\text{If } p = q = 0 \text{ then } z_1 = z_2 = z_3 = \frac{-a_2}{3}$$

$$\text{else } z_1 = \sqrt[3]{-4q} \text{ and } z_2 = z_3 = \sqrt[3]{\frac{q}{2}}$$

The solutions for $D < 0$ (3 different real results):

$$z_i = 2\sqrt{\frac{-p}{3}}\cos(\frac{\varphi}{3} + i \cdot \frac{2\pi}{3}) \text{ with } i \in \{1, 2, 3\}$$

$$\varphi = \arccos\frac{-q}{2\sqrt{-(\frac{p}{3})^3}}$$

In every case we have to regard the substitution by equation (9), and so the final results for x_i are those:

$$x_i = z_i - \frac{a_2}{3} \text{ with } i \in \{1, 2, 3\}$$

Remembering where we started, we have following situation. For the starting operation, **point doubling**, **D must be zero**, as our straight line is crossing the elliptic curve in point $\vec{R}$ and it is at the same time a tangent to the elliptic curve in point $\vec{P}$. So the intersection point $\vec{Q}$ moved into $\vec{P}$ ($\vec{Q} = \vec{P}$), which is reflected by the 2 equal real solutions x_2 and x_3. For the follow up **point additions D must be negative** as we have 3 different real solutions. In general we can say that the solutions are:

$$x_1 = x_R$$

$$x_2 = x_Q$$

$$x_3 = x_P$$

This calculation can easily be done numerically, but it is not handy. To find a much better solution to the problem, we have to look into our **mathematical trick box**. If we look carefully at equation (8) we may remember that Franciscus Vieta (alias Francois Viete) in the 16th century found following equation, so-called **Vieta's root theorem** (German: Vietascher Wurzelsatz):

If the solutions to following equation

$$\boxed{x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0}$$

are x_1, x_2, ..., x_n, then following is true:

$$\boxed{\sum_{i=1}^{n} x_i = -a_{n-1}}$$

Applied to our cubic equation

$$\boxed{x^3 + a_2x^2 + a_1x + a_0 = 0}$$

that means following:

$$\boxed{x_1 + x_2 + x_3 = -a_2}$$

As we found for solutions x_1, x_2 and x_3 the values x_R, x_Q and x_P, we will get following result

$$x_R + x_Q + x_P = s_p^2$$

which leads us to a straight forward solution for x_R:

$$x_R = \begin{cases} x_R = s_p^2 - 2x_P, & \text{if } \vec{P} = \vec{Q} \text{ (point doubling)} \\ x_R = s_p^2 - x_Q - x_P, & \text{if } \vec{P} \neq \vec{Q} \text{ (point addition)} \end{cases} \tag{10}$$

$$\text{with } s_P = \begin{cases} \frac{3x_P^2+c}{2 \cdot y_P}, & \text{if } \vec{P} = \vec{Q} \text{ (point doubling)} \\ \frac{y_Q - y_P}{x_Q - x_P}, & \text{if } \vec{P} \neq \vec{Q} \text{ (point addition)} \end{cases} \tag{11}$$

$$\boxed{y_R = s_P(x_P - x_R) - y_P} \tag{12}$$

Having a closer look at that result, which is valid for **real numbers** $\mathbb{R}$, shows that **starting with a rational point** $\vec{P} = (x_P, y_P)$ with $x_P, y_P \in \mathbb{Q}$, would result in following: The slope s_P would be rational as well, and so would be x_R and y_R. That means that the operation is fully valid for the subgroup of **rational numbers** $\mathbb{Q}$ and so the rational points on the curve E build an **abelian group** $E(\mathbb{Q})$.

If we start at an **integer point** $\vec{P}$ with $(x_P, y_P) \in (\mathbb{Z})$, then **point doubling** would result again in an integer point if following is true:

$$\boxed{3x_P^2 + c = n(2 \cdot y_P) \text{ with } n \in \mathbb{N}^+} \tag{13}$$

The following repetitive **point additions** of $\vec{P}$ would again result in integer points if following is true:

$$\boxed{y_Q - y_P = n(x_Q - x_P) \text{ with } n \in \mathbb{N}^+} \tag{14}$$

So if (13) and (14) are fulfilled we have a **discrete abelian group** $E(\mathbb{Z})$ for our operation.

It is possible to choose the elliptic curve and the starting point (**generator** $\vec{G}$) the way that equation (13) and (14) are fulfilled. An example is the curve with $c = 0$ and $d = 1$:

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

The x component of the generator point is at $x_G = 2$ and so $y_G^2 = 8 + 1$ results in $y_G = \pm 3$, and the choice will be $y_G := -3$. This would produce following results (for the last result please refer to equation (1):

$\vec{P_1}$	$= \vec{G} = (2, -3)$
$\vec{P_2}$	$= 2\vec{P_1} = (0, -1)$
$\vec{P_3}$	$= 3\vec{P_1} = \vec{P_2} + \vec{G} = (-1, 0)$
$\vec{P_4}$	$= 4\vec{P_1} = \vec{P_3} + \vec{G} = (0, 1)$
$\vec{P_5}$	$= 5\vec{P_1} = \vec{P_4} + \vec{G} = (2, 3)$
$\vec{P_6}$	$= 6\vec{P_1} = \vec{P_5} + \vec{G} = (2, 3) + (2, -3) = \vec{P_\infty}$

So the order q of our generator $\vec{G}$ is $q = 6$.

With the discrete points on such a curve $E(\mathbb{Z})$ and and such a generator $\vec{G}$ a **cyclic discrete abelian group** can be build easily by defining:

$$\boxed{\vec{P_\infty} := \vec{G}} \tag{15}$$

For practical use we want just **positive integer numbers** and a limited field size n ($\mathbb{F}_n$) and thus a new order q' for our curve. We then define that **the cycle starts already when n is overstepped**:

$$\vec{R} := \begin{cases} \vec{G}, & \text{if } x_R - |x_0| > n \text{ or } y_R - |y_0| > n \\ \vec{R}, & \text{else} \end{cases} \tag{16}$$

$$x'_R = x_R + |x_0| \tag{17}$$

$$y'_R = y_R + |y_0| \tag{18}$$

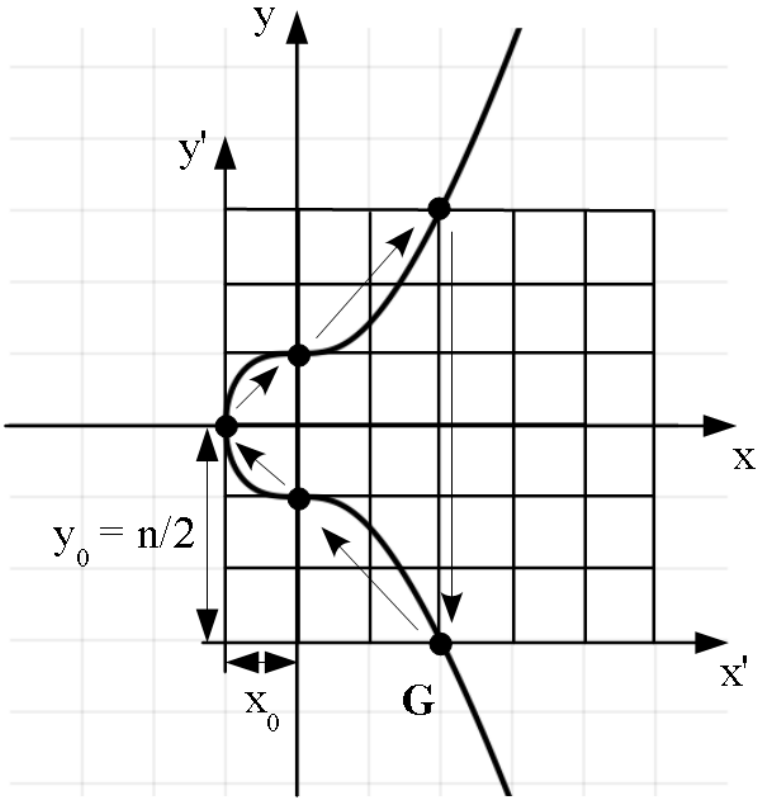
Whenever the n is overstepped by any coordinate, the algorithm steps back to $\vec{G}$ again. To get back to normal coordinates:

$$x_R = x'_R - |x_0| \tag{19}$$

$$y_R = y'_R - |y_0| \tag{20}$$

With the values of the above example and a field size of **n := 6**, we are going to have q' = 5 integer points in a cycle:

Figure 7: ECDH with Finite Field



This cycle now has **the same effect as the mod p operation with the exponential function**: the more often we apply the operation, the more often we will see the same values, which makes the operation **irreversible**.
So now, that we know how to compute our stuff, we look at the **key agreement** using an elliptic curve. For that first of all we define our **domain parameters**. That is an elliptic curve (**coefficients c and d**) and a **base point on the curve** ($\vec{G}$), so that we get a high **order q**. Using a **finite field** $\mathbb{F}_n$ **with an appropriate field size n** we define a **discrete cyclic abelian group** by getting back to $\vec{G}$ in the moment the operation oversteps the field limits. The new order by that is q'.

Now we can defines the random numbers a and b as private keys for user A and user B. Those are used to build the secret key $\vec{K}$ by point addition:

$$\vec{K} := ab\vec{G}$$

The public keys are build by both users like this:

$$\text{User A: } \vec{K}_{pub,A} := a\vec{G}$$

$$\text{User B: } \vec{K}_{pub,B} := b\vec{G}$$

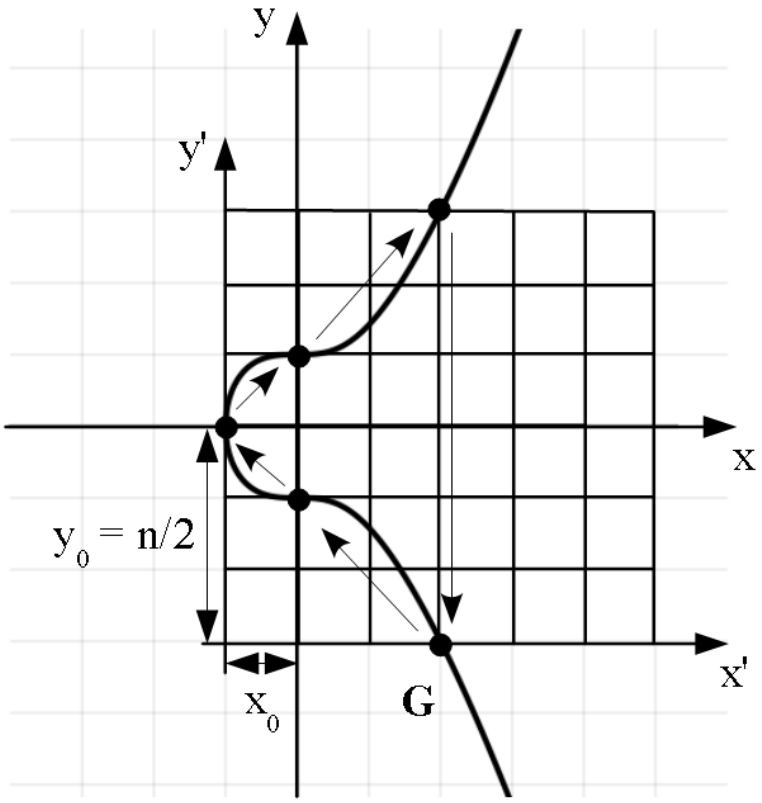
After exchanging their public keys, both users will be able to build the secret key:

$$\vec{K} = (x_K, y_K) = ab\vec{G} = a\vec{K}_{pub,B} = b\vec{K}_{pub,A}$$

Finally x'_K is used as secret encryption/decryption key by both users:

$$k := x'_K = x_K + |x_0|$$

Using the above curve as example with $\vec{G} = (2, -3)$, $a = 7$ and $b = 13$



this would result in:

$$\vec{K}_{pub,A} = 7\vec{G} = (0, -1)$$

$$\vec{K}_{pub,B} = 13\vec{G} = (-1, 0)$$

$$\vec{K} = a\vec{K}_{pub,B} = (-1, 0) \oplus 7\vec{G} = (2, 3)$$

$$\vec{K} = b\vec{K}_{pub,A} = (0, -1) \oplus 13\vec{G} = (2, 3)$$

$$x'_K = x_K + |x_0| = 2 + 1 = 3$$

Practically there are different implementations, for example from the National Institute of Standards and Technology (NIST). NIST P-521 defines the filed size n as prime p using a binary polynomial with 521 bits:

NIST P-521: $n = p = 2^{521} - 1$

Following decimal numbers are used in NIST P-521:

```
 p = 17976931348623159077083915679378745319786029604875
     60117064444236841971802161585193689478337958649255
     41502180565485980503646440548199239100050792877003
     35581663922955313623907650873575991482257486257500
     74253020774477125895509579377784244424266173347276
     29299387668709205606050270810842907692932019128194
     467627007

 c = 68647976601306097149819007990813932172694353001433
     05409394463459185543183397656052122559640661454554
     97729631139148085803712198799971664381257402829111
     5057148

 d = 10938490380737342745111123907668055699362075989516
     83748994586394495953116150735016013708737573759623
     24859213229670631330943845253159101291214232748847
     8985984

xG = 26617408020502170632287687167233609607298591687569
     73147706671368418802944996427808491545080627771902
     35209424122506555866215711354557091681416163731589
     5999846

yG = 37571800257700204635455072244911836035944551347697
     62486694567779615544477440556316691234405012945539
     56214444453728942852258566672919658081012434427757
     8376784

 q = 68647976601306097149819007990813932172694353001433
     05409394463459185543183397655394245057746333217197
     53296399637136332111386476861244038034037280889270
     7005449
```

4 Comparing DH and ECDH

Traditionally the Diffie-Hellman Key Agreement Method **(DH)** publishes the **base value g** and the **prime p** (number size). The Elliptic Curve Diffie-Hellman Key Agreement Method **(ECDH)** publishes the elliptic curve by the **coefficients c and d**, the **base value** $\vec{G}$ and the **field size n**.

Table 1: **Comparing DH and ECDH**

Name	DH	ECDH
Function/Curve	Function $y = g^x$	Curve $y^2 = x^3 + cx + d$
Size	prime p (different values)	order q (different points)
Base Value	$g \in [2, p-1]$	$\vec{G}$ with $x_G, y_G \in [0, n]$
Private keys	a,b (random numbers)	a,b (random numbers)
Public keys	$g^a \bmod p$, $g^b \bmod p$	$a\vec{G}$, $b\vec{G}$ (reduced to n x n)
Secret Key	$g^{ab} \bmod p$	$ab\vec{G}$ (x component)

ECDH is harder to crack, compared to traditional DH. The reason is that the traditional DH uses an exponential function $y = g^a \bmod p$, and guessing which value a was reduced from g^a to y by $\bmod p$ (discrete logarithm problem) is an easier task then doing the same kind of thing for the point addition which is cycling on an elliptic curve in a finite field (elliptic curve discrete logarithm problem).

5 DHE, ECDHE & PFS

To increase security, the principle of having a private key just for one message was introduced (RFC 2631), so the lifetime of a private key is short (= the key is **ephemeral**). To do so, the server (= sender) generates a new private key for each message. With the traditional Diffie-Hellmann Key Agreement Method this is called **DHE**. When using elliptic curves is is called **ECDHE**.

This much increases security, because without it the situation is as follows. If an attacker records messages encrypted with a **static private key** and keeps them stored until computing performance and algorithms become efficient enough, then one day he might be able to find the private key and decrypt the recorded traffic. But if the keys are **ephemeral**, meaning every message is encrypted with another private key, the attacker would be able to see just one message. For all other messages he would have to invest again the same effort for cracking the private key. To not be able to decrypt all messages, if one message was cracked, is called **Perfect Forward Secrecy (PFS)**, and means that cracking one message has no consequences for the security of the follow up messages (German: perfekt fortgesetzte Geheimhaltung).

Technically the server needs to offer appropriate **cipher suites** to the client. Using OpenSSL, the configuration tells the server (e.g. HTTPS server) by following parameters that **ephemeral** private keys are an option, but a **fallback** to other suites may be agreed, if requested by the client (web-browser) during the initial handshake:

```
SSLCipherSuite ECDHE-RSA-AES128-SHA:DHE-RSA-AES128-SHA:AES128-SHA:RC4-SHA
```

This setup is compatible with most web-browsers. ECDHE stands for Elliptic Curve Diffie-Hellmann Ephemeral (DHE for Diffie-Hellmann Ephemeral), RSA specifies the authentication algorithm, AES128 the encryption algorithm and SHA the hashing algorithm.

But if **PFS** and thus **ephemeral privates keys is a must** for the server, then a setup like this is required:

```
SSLCipherSuite ECDHE-RSA-AES128-SHA:DHE-RSA-AES128-SHA
```

REMARK: Due to the overhead, server's using **ephemeral privates keys** increase the message delivering time. So compared to using static keys is is slower. However, OpenSSL provides for example a 64 bit optimized version of NIST P-224, which costs only about 15% overhead in time.

6 The Man In The Middle Attack

The above mentioned key exchange method still has a weak part: it is possible that a hacker (user C) hooks into the traffic of user A and user B. When A initiates a conversation, C fetches all messages and pretends to be B. Knowing the key exchange method, C will send back $k_{pub,C}$ while spoofing the address of B, and A will think that this is user B at the other end of the connection.

To overcome the man-in-the-middle attack, the mechanism of sending the fingerprint of the public part of the RSA host key was introduced. That fingerprint is a short representation of host's public key, and can be used by a person to manually compare it to a fingerprint that was written down before. So before the actual encrypted connection (SSH, SCP or SFTP) is setup, the user is presented the fingerprint of the host of the other user. Now, by checking his notes, he can manually verify the correctness of that fingerprint. Of course any kind of manual transmission of the authentic fingerprint is required before, possibly by a phone call to the other user. Verifying the fingerprint is a fairly safe method to find out if the other side is really the host we want to contact. But is is not always possible, as it requires to get in contact with the other side. And of course the other side can be a host (e.g. a file server), which runs in a server room and the administrator might not be reachable for normal users.

Using OpenSSL, a machine's RSA fingerprint can be determined like this:

```
$ ssh-keygen -l -f /etc/ssh/ssh_host_rsa_key.pub
2048 c3:f6:2a:01:cd:39:61:7f:df:53:57:3e:d8:e4:99:36 /etc/...st_rsa_key.pub
```