Cryptography Lecture 10
Elliptic curve cryptography, key distribution and trust
## Key length

### Table 7.2: Key-size Equivalence.

<table>
<thead>
<tr>
<th>Security (bits)</th>
<th>RSA</th>
<th>DLOG Field Size</th>
<th>DLOG Subfield</th>
<th>EC</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>480</td>
<td>480</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>56</td>
<td>640</td>
<td>640</td>
<td>112</td>
<td>112</td>
</tr>
<tr>
<td>64</td>
<td>816</td>
<td>816</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>80</td>
<td>1248</td>
<td>1248</td>
<td>160</td>
<td>160</td>
</tr>
<tr>
<td>112</td>
<td>2432</td>
<td>2432</td>
<td>224</td>
<td>224</td>
</tr>
<tr>
<td>128</td>
<td>3248</td>
<td>3248</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>160</td>
<td>5312</td>
<td>5312</td>
<td>320</td>
<td>320</td>
</tr>
<tr>
<td>192</td>
<td>7936</td>
<td>7936</td>
<td>384</td>
<td>384</td>
</tr>
<tr>
<td>256</td>
<td>15424</td>
<td>15424</td>
<td>512</td>
<td>512</td>
</tr>
</tbody>
</table>

### Table 7.3: Effective Key-size of Commonly used RSA/DLOG Keys.

<table>
<thead>
<tr>
<th>RSA/DLOG Key</th>
<th>Security (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>50</td>
</tr>
<tr>
<td>768</td>
<td>62</td>
</tr>
<tr>
<td>1024</td>
<td>73</td>
</tr>
<tr>
<td>1536</td>
<td>89</td>
</tr>
<tr>
<td>2048</td>
<td>103</td>
</tr>
</tbody>
</table>

Elliptic curves

- An elliptic curve is the set of solutions to the equation

\[ y^2 = x^3 + ax^2 + bx + c \]

- These solutions are not ellipses, the name elliptic is used for historical reasons and has to do with the integrals used when calculating arc length in ellipses:

\[ \int_a^b \frac{dx}{\sqrt{x^3 + ax^2 + bx + c}} \]
Elliptic curves

- An elliptic curve is the set

\[ E = \{(x, y) : y^2 = x^3 + ax^2 + bx + c\} \]

- Examples:

![Graphs of examples](image-url)
Elliptic curves

- Most of the time a “depressed” cubic is enough

\[ E = \{(x, y) : y^2 = x^3 + bx + c\} \]

- Examples:
Elliptic curves

- You do not want “singular curves” with multiple roots

\[ E = \{(x, y) : y^2 = x^3 + bx + c\} \]

- Examples:
Elliptic curves

- An elliptic curve is the set

\[ E = \{ (x, y) : y^2 = x^3 + bx + c \} \]

- Previously we have used integers (mod \( p \)) and multiplication
Elliptic curves

▶ An elliptic curve is the set

\[ E = \{(x, y) : y^2 = x^3 + bx + c\} \]

▶ Previously we have used the multiplicative group of integers mod \( p \)

▶ We need a group operation on points of \( E \), we’ll call it “addition”
Addition on elliptic curves

- Given two elements in the group, construct a third

Given two points on the elliptic curve $E$, draw a straight line through them. This line will intersect the curve at a third point. If adding a point to itself, use the tangent line.
Addition on elliptic curves

- Given two elements in the group, construct a third
- Draw a straight line through the two points, it will intersect the elliptic curve in a third point.
Addition on elliptic curves

- Given two elements in the group, construct a third

- Draw a straight line through the two points, it will intersect the elliptic curve in a third point. Mirror that in the x-axis
Addition on elliptic curves

- Given two elements in the group, construct a third

- Draw a straight line through the two points, it will intersect the elliptic curve in a third point. Mirror that in the $x$-axis

- If adding a point to itself, use the tangent line
Addition on elliptic curves

- Given two elements in the group, construct a third.

- There is one special case: if the line through the two points is vertical, it will not intersect the elliptic curve again.

- We add the point \((\infty, \infty)\) to \(E\).

- This is the neutral element, the “0”.
Addition on elliptic curves

- Given two elements in the group, construct a third

- The point \((\infty, \infty)\) to \(E\) is the neutral element, the “0”

- That is,
  \[(\infty, \infty) + (x, y) = (x, y)\]

- This also means that \(- (x, y)\) is \((x, -y)\)
Addition on elliptic curves

**Addition law:** On the elliptic curve

\[ E = \{(x, y) : y^2 = x^3 + bx + c\}, \]

\[(x_3, y_3) = (x_1, y_1) + (x_2, y_2)\]

is calculated as follows:

- If \((x_1, y_1) = (x_2, -y_2)\), then \((x_3, y_3) = (\infty, \infty)\)
- If \((x_1, y_1) = (\infty, \infty)\), then \((x_3, y_3) = (x_2, y_2)\) (and the other way around)
- If \((x_1, y_1) = (x_2, y_2)\), then let \(m = (3x_1^2 + b)/(2y_1)\), otherwise let \(m = (y_2 - y_1)/(x_2 - x_1)\), and let

\[(x_3, y_3) = (m^2 - x_1 - x_2, m(x_1 - x_3) - y_1)\]
Multiplication on elliptic curves

- Multiplication with an integer is defined through repeated addition

\[ 3(x, y) = (x, y) + (x, y) + (x, y) \]
Multiplication on elliptic curves

- Multiplication with an integer is defined through repeated addition

\[ 3(x, y) = (x, y) + (x, y) + (x, y) \]
Multiplication on elliptic curves

- Multiplication with an integer is defined through repeated addition

\[ 3(x, y) = (x, y) + (x, y) + (x, y) \]
Discrete elliptic curves

- We want to have a discrete set of points. We arrange this by having coordinates mod $p$

$$E = \{(x, y) : y^2 = x^3 + bx + c \mod p\}$$

- This is not so easy to draw in a diagram, remember, it is $y^2$ mod $p$
Discrete elliptic curves

Example:

\[ E = \{(x, y) : y^2 = x^3 + 4x + 4 \mod 5\} \]

The points in \( E \) are

- \( x = 0 \) gives \( y^2 = 4 \) so that \( y = 2 \) or \( y = 3 \)
- \( x = 1 \) gives \( y^2 = 9 = 4 \) so that \( y = 2 \) or \( y = 3 \)
- \( x = 2 \) gives \( y^2 = 20 = 0 \) so that \( y = 0 \)
- \( x = 3 \) gives \( y^2 = 43 = 3 \), no square root
- \( x = 4 \) gives \( y^2 = 84 = 4 \) so that \( y = 2 \) or \( y = 3 \)
- \( x = \infty \) gives \( y = \infty \)
Discrete elliptic curves

Example:

\[ E = \{(x, y) : y^2 = x^3 + 4x + 4 \mod 5\} \]

The points in \( E \) are

\( (\infty, \infty) \)
Elliptic curves

- Addition as we defined it still works on this set (but lines mod \( p \) need to be handled)

- We now have the group operations to use instead of integer multiplication and exponentiation

- Hasse’s Theorem: The number of points \( N \) in an Elliptic curve \( E \) mod \( p \) obeys

\[
p - 1 - 2\sqrt{p} < N < p - 1 + 2\sqrt{p}
\]
Elliptic curves

- Addition as we defined it still works on this set (but lines mod $p$ need to be handled)

- We now have the group operations to use instead of integer multiplication and exponentiation

- Hasse’s Theorem: The number of points $N$ in an Elliptic curve $E \mod p$ obeys

$$p - 1 - 2\sqrt{p} < N < p - 1 + 2\sqrt{p}$$
Addition on elliptic curves

Addition law: On the elliptic curve

\[ E = \{(x, y) : y^2 = x^3 + bx + c\}, \]

\[(x_3, y_3) = (x_1, y_1) + (x_2, y_2)\]

is calculated as follows:

- If \((x_1, y_1) = (x_2, -y_2)\), then \((x_3, y_3) = (\infty, \infty)\)
- If \((x_1, y_1) = (\infty, \infty)\), then \((x_3, y_3) = (x_2, y_2)\) (and the other way around)
- If \((x_1, y_1) = (x_2, y_2)\), then let \(m = \frac{3x_2^2 + b}{2y_1}\), otherwise let \(m = \frac{y_2 - y_1}{x_2 - x_1}\), and let

  \[(x_3, y_3) = (m^2 - x_1 - x_2, m(x_1 - x_3) - y_1)\]
Addition on elliptic curves

Addition law: On the elliptic curve

\[ E = \{(x, y) : y^2 = x^3 + bx + c\}, \]

\[ (x_3, y_3) = (x_1, y_1) + (x_2, y_2) \]

is calculated as follows:

- If \((x_1, y_1) = (x_2, -y_2)\), then \((x_3, y_3) = (\infty, \infty)\)
- If \((x_1, y_1) = (\infty, \infty)\), then \((x_3, y_3) = (x_2, y_2)\) (and the other way around)
- If \((x_1, y_1) = (x_2, y_2)\), then let \(m = (3x_1^2 + b)/(2y_1)\), otherwise let \(m = (y_2 - y_1)/(x_2 - x_1)\), and let

\[ (x_3, y_3) = (m^2 - x_1 - x_2, m(x_1 - x_3) - y_1) \]
Elliptic curves

- Addition as we defined it still works on this set (but lines mod \( p \) need to be handled)

- We now have the group operations to use instead of integer multiplication and exponentiation

- Hasse’s Theorem: The number of points \( N \) in an Elliptic curve \( E \) mod \( p \) obeys

\[
p - 1 - 2\sqrt{p} < N < p - 1 + 2\sqrt{p}
\]
Discrete Logarithms on elliptic curves

- Remember the discrete logarithm problem: given $x$ and a primitive root $g$, find $k$ so that

\[ x = g^k \mod p \]

- There is an analog on elliptic curves: given two points $A$ and $B$ on an elliptic curve, find $k$ so that

\[ B = kA = A + A + \ldots + A \]

- This might seem different, but is the equivalent problem. The only difference is the group operation name ("multiplication or "addition")
Discrete Logarithms on elliptic curves

- The discrete logarithm for elliptic curves: given two points $A$ and $B$ on an elliptic curve, find $k$ so that

\[ B = kA = A + A + \ldots + A \]

- There is an analog for the Polig-Hellman algorithm. This works well when the smallest integer $n$ such that $nA = \infty$ has only small factors
Discrete Logarithms on elliptic curves

- The discrete logarithm for elliptic curves: given two points $A$ and $B$ on an elliptic curve, find $k$ so that

\[ B = kA = A + A + \ldots + A \]

- There is an analog for the Polig-Hellman algorithm

- The baby step-giant step algorithm works, but is impractical since it needs a lot of memory
Discrete Logarithms on elliptic curves

- The discrete logarithm for elliptic curves: given two points $A$ and $B$ on an elliptic curve, find $k$ so that

$$B = kA = A + A + \ldots + A$$

- There is an analog for the Polig-Hellman algorithm

- The baby step-giant step algorithm is impractical

- But most importantly, there is no analog for the index calculus
  - Integer mod $p$ index calculus is based on using small base numbers (not small exponents as in Polig-Hellman)
  - But there are no points on $E$ that are closer to “0” than any other points, the distance to $(\infty, \infty)$ is the same for all other points
## Key length

Table 7.2: Key-size Equivalence.

<table>
<thead>
<tr>
<th>Security (bits)</th>
<th>RSA</th>
<th>DLOG</th>
<th>EC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>field</td>
<td>size</td>
<td>subfield</td>
</tr>
<tr>
<td>48</td>
<td>480</td>
<td>480</td>
<td>96</td>
</tr>
<tr>
<td>56</td>
<td>640</td>
<td>640</td>
<td>112</td>
</tr>
<tr>
<td>64</td>
<td>816</td>
<td>816</td>
<td>128</td>
</tr>
<tr>
<td>80</td>
<td>1248</td>
<td>1248</td>
<td>160</td>
</tr>
<tr>
<td>112</td>
<td>2432</td>
<td>2432</td>
<td>224</td>
</tr>
<tr>
<td>128</td>
<td>3248</td>
<td>3248</td>
<td>256</td>
</tr>
<tr>
<td>160</td>
<td>5312</td>
<td>5312</td>
<td>320</td>
</tr>
<tr>
<td>192</td>
<td>7936</td>
<td>7936</td>
<td>384</td>
</tr>
<tr>
<td>256</td>
<td>15424</td>
<td>15424</td>
<td>512</td>
</tr>
</tbody>
</table>

Table 7.3: Effective Key-size of Commonly used RSA/DLOG Keys.

<table>
<thead>
<tr>
<th>RSA/DLOG Key</th>
<th>Security (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>50</td>
</tr>
<tr>
<td>768</td>
<td>62</td>
</tr>
<tr>
<td>1024</td>
<td>73</td>
</tr>
<tr>
<td>1536</td>
<td>89</td>
</tr>
<tr>
<td>2048</td>
<td>103</td>
</tr>
</tbody>
</table>

Trapdoor one-way functions

- A trapdoor one-way function is a function that is easy to compute but computationally hard to reverse
  - Easy to calculate $x_A$ from $x$
  - Hard to invert: to calculate $x$ from $x_A$

- A trapdoor one-way function has one more property, that with certain knowledge it *is* easy to invert, to calculate $x$ from $x_A$

- There is no proof that trapdoor one-way functions exist, or even real evidence that they can be constructed
Standard \((m \mod p)\) ElGamal encryption

- Choose a large prime \(p\), and a primitive root \(\alpha \mod p\). Also, take a random integer \(a\) and calculate \(\beta = \alpha^a \mod p\).

- The public key is the values of \(p\), \(\alpha\), and \(\beta\), while the secret key is the value \(a\).

- Encryption uses a random integer \(k\) with \(\gcd(k, p - 1) = 1\), and the ciphertext is the pair \((\alpha^k, \beta^k m)\), both \(\mod p\).

- Decryption is done with \(a\), by calculating

\[
(\alpha^k)^{-a}(\beta^k m) = (\alpha^{-ak})(\alpha^{ak} m) = m \mod p
\]
Elliptic curve ElGamal encryption

- Choose an elliptic curve $E \mod p$ a large prime and a point $\alpha$ on $E$. Also, take a random integer $a$ and calculate $\beta = a\alpha$

- The public key is $E$ and the values of $p$, $\alpha$, and $\beta$, while the secret key is the value $a$

- Encryption uses a random integer $k$, and the ciphertext is the pair $(k\alpha, k\beta + m)$

- Decryption is done with $a$, by calculating

$$-a(k\alpha) + (k\beta + m) = -ak\alpha + k(a\alpha) + m = m$$
Representing plaintext on elliptic curves

- Unfortunately, it is not simple to represent a given plaintext as a point on $E$

- Even worse, there is actually no polynomial time algorithm that can write down all points of an elliptic curve

- There is a method that will work with high probability:
  - The message $m$ should be in the $x$-coordinate, but there is no guarantee that $m^3 + bm + c$ is a square mod $p$
  - Each number $x$ has a probability of about 1/2 that $x^3 + bx + c$ is a square, so put a few bits at the end of $m$ and run through all possible values
  - If the number of possible values is $K$, the risk of failure is $2^{-K}$
Standard (integer mod p) Diffie-Hellman key exchange

- Use two one-way functions $f$ and $g$: exponentiation mod $p$ (of a primitive root $\alpha$), the symmetry is

$$\left(\alpha^a\right)^b = \left(\alpha^b\right)^a \mod p$$

- This cannot be used for encryption/signing because one does not recover $a$ or $b$.

- But it can be used for key exchange: parameters $p$ and $\alpha$
  - Alice takes a secret random $a$ and makes $\alpha^a$ public
  - Bob takes a secret random $b$ and makes $\alpha^b$ public
  - Both can now create $k = \left(\alpha^a\right)^b = \left(\alpha^b\right)^a \mod p$
Elliptic curve Diffie-Hellman key exchange

- Use two one-way functions $f$ and $g$: multiplication on an elliptic curve $E$ (of a point $\alpha$), the symmetry is

$$b(a\alpha) = a(b\alpha)$$

- This cannot be used for encryption/signing because one does not recover $a$ or $b$.

- But it can be used for key exchange: parameters $E$, $p$ and $\alpha$
  - Alice takes a secret random $a$ and makes $a\alpha$ public
  - Bob takes a secret random $b$ and makes $b\alpha$ public
  - Both can now create $k = b(a\alpha) = a(b\alpha)$
Standard (mod p) ElGamal signatures

- Choose a large prime \( p \), and a primitive root \( \alpha \mod p \). Also, take a random integer \( a \) and calculate \( \beta = \alpha^a \mod p \).
- The public key is the values of \( p, \alpha, \) and \( \beta \), while the secret key is the value \( a \).
- Signing uses a random integer \( k \) with \( \gcd(k, p - 1) = 1 \), and the signature is the pair \((r, s)\) where
  \[
  r = \alpha^k \mod p \\
  s = k^{-1}(m - ar) \mod (p - 1)
  \]
- Verification is done comparing \( \beta^r r^s \) and \( \alpha^m \mod p \), since
  \[
  \beta^r r^s = \alpha^{ar} \alpha^{k(m-ar)/k} = \alpha^m \mod p
  \]
Elliptic curve ElGamal signatures

- Choose an elliptic curve $E \text{ mod } p$, and a point $\alpha$ on $E$. Also, take a random integer $a$ and calculate $\beta = a\alpha$

- The public key is $E$ and the values of $p$, $\alpha$, and $\beta$, while the secret key is the value $a$

- Signing uses a random integer $k$ with $\gcd(k, n) = 1$ where $n$ is the number of points on $E$. The signature is created by inverting $k \text{ mod } n$ and forming the pair $(r, s)$ as

$$r = k\alpha$$
$$s = k^{-1}(m - ar_x)$$

- Verification is done comparing $r_\times \beta + sr$ and $m\alpha$, since

$$r_\times \beta + sr = r_\times (a\alpha) + (k^{-1}(m - ar_x))(k\alpha)$$
$$= r_\times (a\alpha) + m\alpha - ar_\times \alpha = m\alpha$$
A trapdoor one-way function is a function that is easy to compute but computationally hard to reverse

- Easy to calculate $f(x)$ from $x$
- Hard to invert: to calculate $x$ from $f(x)$

A trapdoor one-way function has one more property, that with certain knowledge it is easy to invert, to calculate $x$ from $f(x)$

There is no proof that trapdoor one-way functions exist, or even real evidence that they can be constructed. Examples:

- RSA (factoring)
- Knapsack (NP-complete but insecure with trapdoor)
- Diffie-Hellman + ElGamal (discrete log)
- EC Diffie-Hellman + EC ElGamal (EC discrete log)
Key Management

- The first key in a new connection or association is always delivered via a courier.
- Once you have a key, you can use that to send new keys.
- If Alice shares a key with Trent and Trent shares a key with Bob, then Alice and Bob can exchange a key via Trent (provided they both trust Trent).
Key distribution center

- If Alice shares a key with Trent and Trent shares a key with Bob, then Alice and Bob can exchange a key via Trent (provided they both trust Trent)

Trent
Key distribution center
\( K_{AT}, K_{BT} \)

Alice, \( K_{AT} \)  
Bob, \( K_{BT} \)
If Alice shares a key with Trent and Trent shares a key with Bob, then Alice and Bob can exchange a key via Trent (provided they both trust Trent)
If Alice shares a key with Trent and Trent shares a key with Bob, then Alice and Bob can exchange a key via Trent (provided they both trust Trent)
If Alice shares a key with Trent and Trent shares a key with Bob, then Alice and Bob can receive a key from Trent (provided they both trust Trent)
If Alice shares a key with Trent and Trent shares a key with Bob, then Alice and Bob can receive a key from Trent (provided they both trust Trent)
If Alice shares a key with Trent and Trent shares a key with Bob, and Alice and Bob each have a public id $r_A$, $r_B$, they can receive key-generation info from Trent (provided they both trust Trent)

$$\begin{align*}
a_U &= a + br_U, \\
b_U &= b + cr_U
\end{align*}$$

Key distribution center, Blom key pre-distribution

Trent
Key distribution center
$K_{AT}, K_{BT}, a, b, c$

$$K_{AB} = a_A + b_A r_B = a_B + b_B r_A$$
Key distribution center, Station-To-Station (STS) protocol

▶ What about Diffie-Hellman key exchange?

<table>
<thead>
<tr>
<th>Alice, $K_{AT}$</th>
<th>$\alpha^a \mod p$</th>
<th>Trent</th>
<th>Key distribution center $K_{AT}, K_{BT}$</th>
<th>Bob, $K_{BT}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$\alpha^b \mod p$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key distribution center, Station-To-Station (STS) protocol

- What about Diffie-Hellman key exchange?
- Eve can do an “intruder-in-the-middle”

\[ \alpha^a \mod p \]
\[ \alpha^e \mod p \]
\[ \alpha^b \mod p \]
Key distribution center, Station-To-Station (STS) protocol

- If Alice shares a key with Trent and Trent shares a key with Bob, then Alice and Bob can use Trent to verify that they exchange key with the right person.

\[ K_{AT}, K_{BT} \]
Key distribution center, Station-To-Station (STS) protocol

- If Alice shares a key with Trent and Trent shares a key with Bob, then Alice and Bob can use Trent to verify that they exchange key with the right person

![Diagram showing the STS protocol with Key distribution center, Trent, Alice, K_{AT}, K_{BT}, Bob, K_{BT}, and the exchange of keys α^a, E_{K_{AB}}(\text{sig}_A(α^a, α^b)), and α^b, E_{K_{AB}}(\text{sig}_B(α^a, α^b)).]
Key distribution center, Station-To-Station (STS) protocol

- If Alice shares a key with Trent and Trent shares a key with Bob, then Alice and Bob can use Trent to verify that they exchange key with the right person.
Key distribution center, Station-To-Station (STS) protocol

- If Alice shares a key with Trent and Trent shares a key with Bob, then Alice and Bob can use Trent to verify that they exchange key with the right person.
If Alice shares a key with Trent and Trent shares a key with Bob, then Alice and Bob can exchange a key via Trent (provided they both trust Trent)
But perhaps Eve has broken a previously used key, and intercepts Alice’s request.
Key distribution center, replay attacks

- But perhaps Eve has broken a previously used key, and intercepts Alice’s request.
- Then she can fool Bob into communicating with her.
Key distribution center, wide-mouthed frog

- Alice and Trent add time stamps to prohibit the attack
Key distribution center, wide-mouthed frog

- Alice and Trent add time stamps to prohibit the attack
Key distribution center, wide-mouthed frog

- Alice and Trent add time stamps to prohibit the attack.
- But now, Eve can pretend to be Bob and make a request to Trent.

![Diagram of key distribution center]
Key distribution center, wide-mouthed frog

- Alice and Trent add time stamps to prohibit the attack
- But now, Eve can pretend to be Bob and make a request to Trent, who will forward the key to Alice
Key distribution center, Needham-Schroeder key agreement

- Another variation is to use nonces to prohibit the replay attack
Key distribution center, Needham-Schroeder key agreement

- Another variation is to use nonces to prohibit the replay attack

1. ID_A || ID_B || r_1
2. E_{K_{AT}}(K_S || ID_B || r_1 || E_{K_{BT}}(K_S || ID_A))
Key distribution center, Needham-Schroeder key agreement

- Another variation is to use nonces to prohibit the replay attack

\[
\begin{align*}
1: & \text{Alice, } K_{AT} \\
2: & E_{K_{AT}}(K_S || ID_B || r_1 || E_{K_{BT}}(K_S || ID_A)) \\
3: & E_{K_{BT}}(K_S || ID_A)
\end{align*}
\]
Key distribution center, Needham-Schroeder key agreement

- Another variation is to use nonces to prohibit the replay attack

```
1: ID_A||ID_B||r_1

2: E_{K_{AT}}(K_S||ID_B||r_1||E_{K_{BT}}(K_S||ID_A))

3: E_{K_{BT}}(K_S||ID_A)

4: E_{K_S}(r_2)
```
Key distribution center, Needham-Schroeder key agreement

Another variation is to use nonces to prohibit the replay attack

\[
1: \text{ID}_A || \text{ID}_B || r_1 \\
2: E_{K_{AT}}(K_S || ID_B || r_1 || E_{K_{BT}}(K_S || ID_A)) \\
3: E_{K_{BT}}(K_S || ID_A) \\
4: E_{K_S}(r_2) \\
5: E_{K_S}(r_2 - 1)
\]
Key distribution center, Needham-Schroeder key agreement

- Another variation is to use nonces to prohibit the replay attack
- If Eve ever breaks one session key, she can get Bob to reuse it

Alice, $K_{AT}$  Eve  Bob, $K_{BT}$

1: $E_{K_{BT}}(K_S || ID_A)$
2: $E_{K_S}(r_2)$
3: $E_{K_S}(r_2 - 1)$
Kerberos

- Trent $K_C, K_G$
- Grant $K_G, K_S$
- Cliff $K_C$
- Serge $K_S$

- A client, Cliff
- An authentication server, Trent
- An authorization server, Grant
- A service server, Serge
Kerberos

- A client, Cliff
- An authentication server, Trent
- An authorization server, Grant
- A service server, Serge
- They share keys $K_C$, $K_G$, $K_S$
1. Cliff sends Trent $ID_C||ID_G$

2. Trent responds with $E_{K_G}^{\text{KGS}}(ID_C||TGT)$ where $TGT = E_{K_G}^{\text{KGS}}(ID_C||t_1||K_{GC})$

3. Cliff sends Grant $E_{K_G}^{\text{KGS}}(ID_C||t_2)||TGT$

4. Grant responds with $E_{K_S}^{\text{KCS}}(K_{CS})||ST$ where $ST = E_{K_S}^{\text{KCS}}(ID_C||t_3||t_{\text{expir.}}||K_{CS})$

5. Cliff sends Serge $E_{K_S}^{\text{KCS}}(ID_C||t_4)||ST$ and can then use Serge's services.
Kerberos

1. Cliff sends Trent $ID_C \| ID_G$

2. Trent responds with $E_{K_C}(K_{CG}) \| TGT$ where $TGT = ID_G \| E_{K_G}(ID_C \| t_1 \| K_{GC})$

3. Cliff sends Grant $E_{K_C}(K_{CG}) \| TGT$

4. Grant responds with $E_{K_C}(K_{CS}) \| ST$ where $ST = E_{K_S}(ID_C \| t_2 \| t_{exp} \| K_{CS})$

5. Cliff sends Serge $E_{K_S}(ID_C \| t_4) \| ST$ and can then use Serge’s services
Kerberos

1. Cliff sends Trent $ID_C || ID_G$
2. Trent responds with $E_{KC}(KC || TGT)$ where $TGT = ID_G || E_{KG}(ID_C || t_1 || KG)$
3. Cliff sends Grant $E_{KCG}(ID_C || t_2) || TGT$

Cliff sends Serge $E_{KS}(ID_C || t_4) || ST$ and can then use Serge’s services
1. Cliff sends Trent $ID_C \| ID_G$

2. Trent responds with $E_{KC}(KC_G) \| TGT$ where $TGT = ID_G \| E_{KG}(ID_C \| t_1 \| KG_C)$

3. Cliff sends Grant $E_{KC_G}(ID_C \| t_2) \| TGT$

4. Grant responds with $E_{KC_G}(KC_S) \| ST$ where $ST = E_{KS}(ID_C \| t_3 \| t_{expir.} \| KS)$
Kerberos

1. Cliff sends Trent $ID_C || ID_G$

2. Trent responds with $E_{KC}(KC_G) || TGT$ where $TGT = ID_G || E_{KG}(ID_C || t_1 || KG_C)$

3. Cliff sends Grant $E_{KC_G}(ID_C || t_2) || TGT$

4. Grant responds with $E_{KC_G}(KS) || ST$ where $ST = E_{KS}(ID_C || t_3 || t_{expir.} || KS)$

5. Cliff sends Serge $E_{KC_S}(ID_C || t_4) || ST$ and can then use Serge’s services
Public key distribution

- Public key distribution uses a Public Key Infrastructure (PKI)

Certification Authority

\[ s_T, \{ e_i \} \]

Alice, \( v_T, d_A \) 

Bob, \( v_T, d_B \)
Public key distribution, using Certification Authorities

- Public key distribution uses a Public Key Infrastructure (PKI)
- Alice sends a request to a Certification Authority (CA) who responds with a certificate, ensuring that Alice uses the correct key to communicate with Bob

\[ ID_A, v_T, d_A \]

\[ s_T, \{ e_i \} \]

\[ ID_B, e_B, \text{sign}(ID_B, e_B) \]

\[ e_B, v_T, d_B \]
Public key distribution, using X.509 certificates

- The CAs often are commercial companies, that are assumed to be trustworthy
- Many arrange to have the root certificate packaged with IE, Mozilla, Opera,…
- They issue certificates for a fee
- They often use Registration Authorities (RA) as sub-CA for efficiency reasons
Public key distribution, X.509 certificates in your browser
Public key distribution, using web of trust

- No central CA
- Users sign each other’s public key (hashes)
- This creates a “web of trust”
Public key distribution, using web of trust (PGP and GPG)

- No central CA
- Users sign each other’s public key (hashes)
- This creates a “web of trust”
- Each user keeps a keyring with the keys (s)he has signed
- The secret key is stored on a secret keyring, on her/is\{er,is\} computer
- The public key(s) and their signatures are uploaded to key servers
Public key distribution, a web-of-trust path
Secure Sockets Layer (SSL) and Transport Layer Security (TLS)

- This is a client-server handshake procedure to establish key
- The server (but not the client) is authenticated (by its certificate)
Secure Sockets Layer (SSL) and Transport Layer Security (TLS)

**ClientHello:** highest TLS protocol version, random number, suggested public key systems + symmetric key systems + hash functions + compression algorithms

---

![Diagram of SSL/TLS handshake](image-url)
Secure Sockets Layer (SSL) and Transport Layer Security (TLS)

**ClientHello:** highest TLS protocol version, random number, suggested public key systems + symmetric key systems + hash functions + compression algorithms

**ServerHello, Certificate, ServerHelloDone:** chosen protocol version, a (different) random number, system choices, public key
Secure Sockets Layer (SSL) and Transport Layer Security (TLS)

**ClientHello:** highest TLS protocol version, random number, suggested public key systems + symmetric key systems + hash functions + compression algorithms

**ServerHello, Certificate, ServerHelloDone:** chosen protocol version, a (different) random number, system choices, public key

**ClientKeyExchange:** PreMasterSecret, encrypted with the server’s public key
Secure Sockets Layer (SSL) and Transport Layer Security (TLS)

**ClientHello:** highest TLS protocol version, random number, suggested public key systems + symmetric key systems + hash functions + compression algorithms

**ServerHello, Certificate, ServerHelloDone:** chosen protocol version, a (different) random number, system choices, public key

**ClientKeyExchange:** PreMasterSecret, encrypted with the server’s public key

(Master secret): creation of master secret using a pseudorandom function, with the PreMasterSecret as seed

(Session keys): session keys are created using the master secret, different keys for the two directions of communication
Secure Sockets Layer (SSL) and Transport Layer Security (TLS)

**ClientHello:** highest TLS protocol version, random number, suggested public key systems + symmetric key systems + hash functions + compression algorithms

**ServerHello, Certificate, ServerHelloDone:** chosen protocol version, a (different) random number, system choices, public key

**ClientKeyExchange:** PreMasterSecret, encrypted with the server’s public key

(Master secret): creation of master secret using a pseudorandom function, with the PreMasterSecret as seed

(Session keys): session keys are created using the master secret, different keys for the two directions of communication

**ChangeCipherSpec, Finished** authenticated and encrypted, containing a MAC for the previous handshake messages
Secure Sockets Layer (SSL) and Transport Layer Security (TLS)

- SSL 1.0 (no public release), 2.0 (1995), 3.0 (1996), originally by Netscape
- TLS 1.0 (1999), changes that improve security, among other things how random numbers are chosen
  - Sensitive to CBC vulnerability discovered 2002, demonstrated by BEAST attack 2011
  - Current problem: TLS 1.0 is fallback if either end does not support higher versions
Secure Sockets Layer (SSL) and Transport Layer Security (TLS)

- ▶ TLS 1.1 (2006), added protection against CBC attacks by explicit IV specification
- ▶ TLS 1.2 (2008), e.g., change MD5-SHA1 to SHA256
- ▶ Later (2011), never fall back to SSL 2.0