

SECURITY - Security and Cryptography - Examples

The following examples support the completion of the exercises, which together develop a technical understanding of security protocols including their strengths and shortcomings.

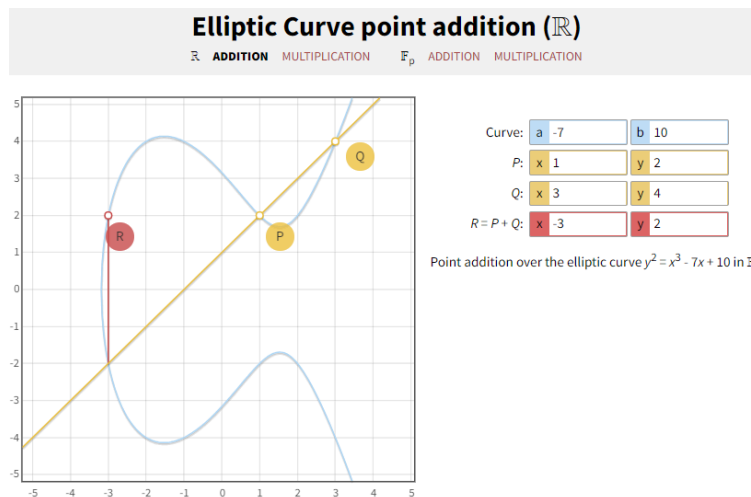
Two tools will be used; let's check that you can open both tools straight away.

Example 0: Opening the tools for the first time

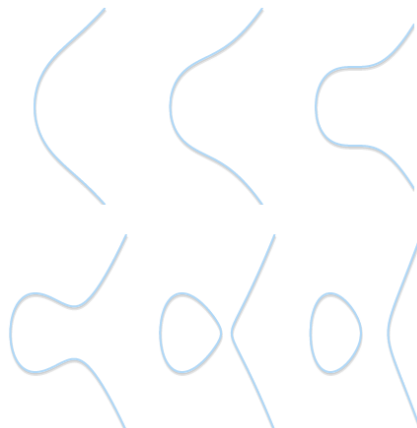
This is mainly just an early test that you are able to access and run the tools.

Later examples will then help you understand how to use them to do more useful things.

Open a browser and navigate to [Elliptic Curve point addition \(\$\mathbb{R}\$ \)](#).



An elliptic curve comprises all points (x, y) for which $y^2 = x^3 + a \cdot x + b$ where a and b constant values (plus an additional point, called the point at infinity). The elliptic curves where $b=1$ and a varies from 2 to -3 are shown below:

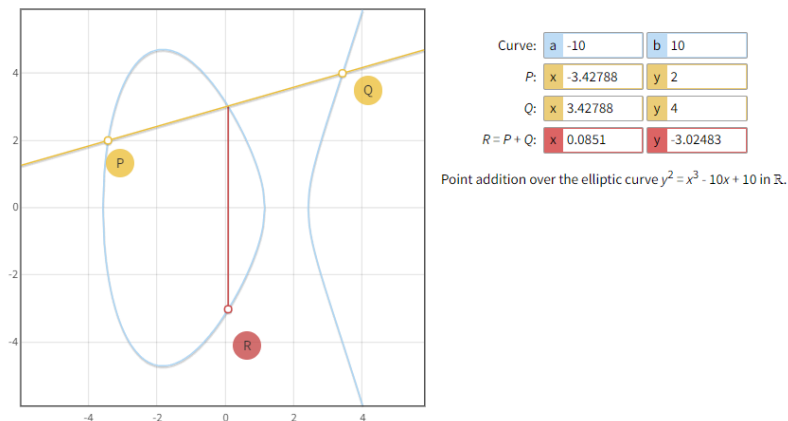


For the arithmetic we use to work, we must only use values for a and b such that $4*a^3+27*b^2 \neq 0$ in order to avoid singular curves, such as those illustrated below.



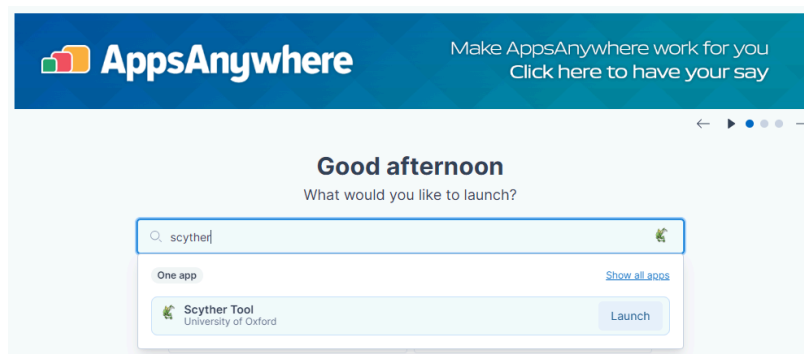
Observe the Elliptic curve curve $y^2 = x^3 - 10x + 10$ in \mathbb{R} .

- > Enter the value $a = -10$ leave the other values unchanged from their defaults (see diagram above)
- > Press Enter



From the diagram you can see that the curve has changed shape because we changed one of the constant values that defines the curve.

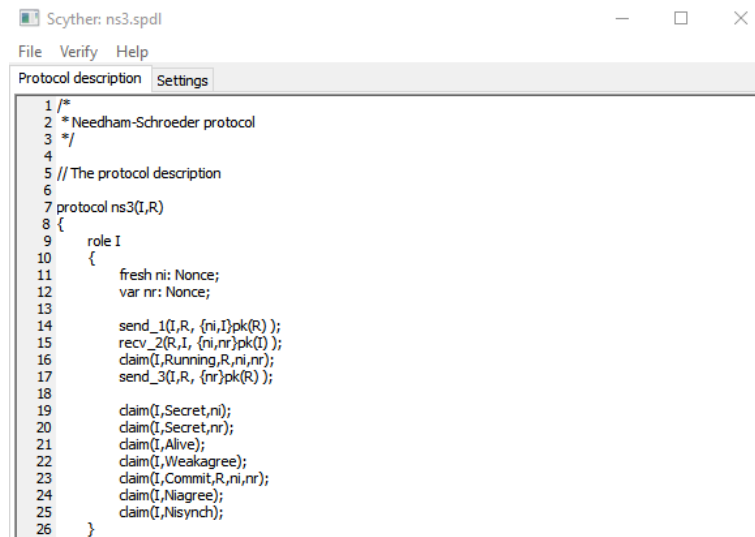
Open a browser, navigate to appsanywhere.port.ac.uk and launch Scyther



(If you are using your own device and your Operating System is not supported then remote access a lab computer by following [these instructions](#) to then open scyther via appsanywhere before then proceeding with the following.)

Click File -> Open to open ns3.spdl

Scyther provides a text editor to read and edit a protocol description specified in Security Protocol Description Language (SPDL).



```
1 /*
2  * Needham-Schroeder protocol
3  */
4
5 // The protocol description
6
7 protocol ns3(I,R)
8 {
9   role I
10  {
11    fresh ni: Nonce;
12    var nr: Nonce;
13
14    send_1(I,R, {ni,I}pk(R));
15    recv_2(R,I, {ni,nr}pk(I));
16    claim(I,Running,R,ni,nr);
17    send_3(I,R, {nr}pk(R));
18
19    claim(I,Secret,ni);
20    claim(I,Secret,nr);
21    claim(I,Alive);
22    claim(I,Weakagree);
23    claim(I,Commit,R,ni,nr);
24    claim(I,Niagree);
25    claim(I,Nisynch);
26  }
```

Scyther enables you to automatically verify the security protocol specified in the file against automatically generated security claims.

Click Verify -> Verify automatic claims (or use shortcut key F6)



Claim	Status	Comments	Patterns
ns3 I ns3,I2 Secret ni	Ok	Verified	No attacks.
ns3,I3 Secret nr	Ok	Verified	No attacks.
ns3,I4 Alive	Ok	Verified	No attacks.
ns3,I5 Weakagree	Ok	Verified	No attacks.
ns3,I6 Niagree	Ok	Verified	No attacks.
ns3,I7 Nisynch	Ok	Verified	No attacks.
R ns3,R2 Secret nr	Fail	Falsified	At least 1 attack. <button>1 attack</button>
ns3,R3 Secret ni	Fail	Falsified	At least 1 attack. <button>1 attack</button>
ns3,R4 Alive	Ok	Verified	No attacks.
ns3,R5 Weakagree	Fail	Falsified	At least 1 attack. <button>1 attack</button>
ns3,R6 Niagree	Fail	Falsified	At least 1 attack. <button>1 attack</button>
ns3,R7 Nisynch	Fail	Falsified	At least 1 attack. <button>1 attack</button>

Done.

The green Ok values show that from the perspective of role I (the initiator of the protocol) the protocol meets all desired claims, which a later example will introduce.

However, from the perspective of role R (the responder within the protocol) the protocol fails to meet most desirable claims.

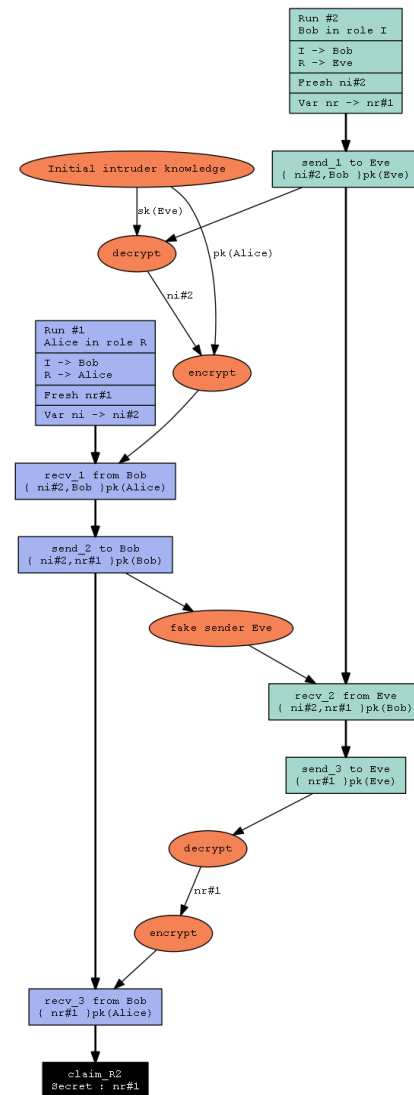
For each failed claim, a counter example is generated to illustrate the protocol failing to achieve the desired security property.

Click the topmost '1 attack' button

Example 2 will describe the counterexample illustrated here; Example 2 analyses the 3-message Needham Schroeder Public Key Protocol, which is specified in ns3.spdl.

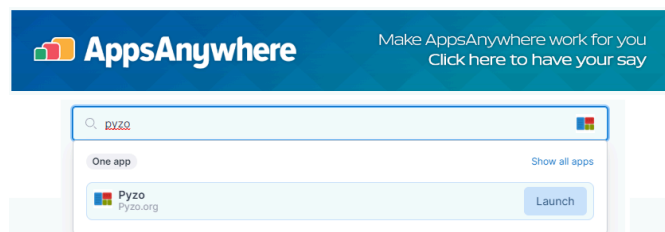
For Exercise 3 and Exercise 4 you will be asked to specify protocols in Security Protocol Description Language (SPDL) and to analyse your protocol specifications using Scyther.

The protocols that you will be specifying are somewhat unique to you and are based on the last three digits of your student number (like in the Cryptography exercises).



Download (and extract) security_protocols.zip from Moodle (available after consolidation week)

Open a browser, navigate to appsanywhere.port.ac.uk and launch Pyzo



(If you are using your own device and your Operating System is not supported then remote access a lab computer by following [these instructions](#) to then open pyzo via appsanywhere before then proceeding with the following.)

Press Ctrl+O (or click File -> Open File) to open protocol_generator.py

Press Ctrl+Shift+E (or click Run -> Run file as script)

The python file `protocol_generator.py` includes a function to generate the values to use in your calculations for the key exchange within Exercise 1.

Make sure you input the last three digits of your student number to generate your values.

Generate `generate_ECDH_values` **for use in Exercise 1**

```
>>> generate_ECDH_values('123') # make sure you change 123!!
```

The python file `protocol_generator.py` includes another two functions: one to generate your protocol for Exercise 3 and one to generate your protocol for Exercise 4.

Make sure you input the last three digits of your student number to generate your protocol.

Generate `PROTOCOL_THREE` **for use in Exercise 3**

```
>>> generate_protocol_three('123') # make sure you change 123!!
```

Generate `PROTOCOL_FOUR` **for use in Exercise 4**

```
>>> generate_protocol_four('123') # make sure you change 123!!
```

Example 1: Diffie Hellman Key Exchange

Diffie Hellman Key Exchange enables two protocol participants to determine a shared secret key value over an insecure channel.

Prior to 1976 there was no (publicly) known solution to the following problem stated by Diffie and Hellman in their paper on [New Directions in Cryptography](#):

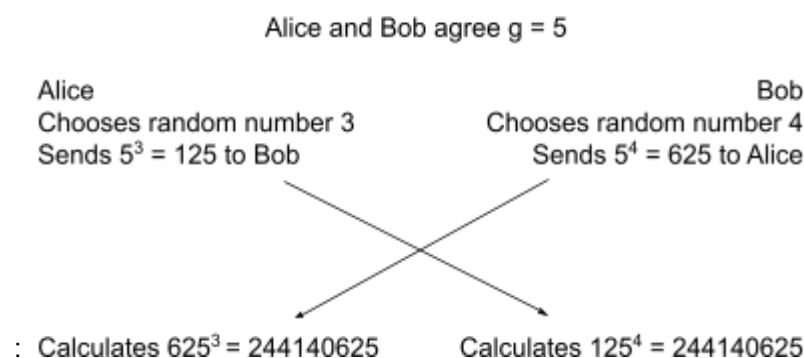
“A private conversation between two people with no prior acquaintance-is a common occurrence in business, however, and it is unrealistic to expect initial business contacts to be postponed long enough for keys to be transmitted by some physical means. The cost and delay imposed by this key distribution problem is a major barrier to the transfer of business communications to large teleprocessing networks.”

Diffie and Hellman proposed Public Key Cryptography as a way of establishing/distributing shared symmetric keys through the use of pairs of sets of public and private key values:

“Public key distribution systems offer a different approach to eliminating the need for a secure key distribution channel. In such a system, two users who wish to exchange a key communicate back and forth until they arrive at a key in common. A third party eavesdropping on this exchange must find it computationally infeasible to compute the key from the information overheard.”

Diffie-Hellman Key Exchange

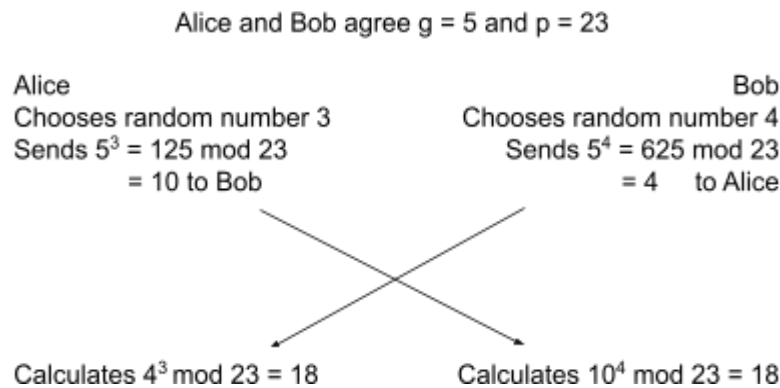
Alice and Bob first agree upon a generator value, g , which can be publicly known. Alice (respectively Bob) then chooses a random number a (resp. b) and sends the value $X=g^a$ (resp. $Y=g^b$) to Bob (resp. Alice). Alice then calculates Y^a and Bob calculates X^b , which turn out to be the same number because $Y^a = (g^b)^a = g^{ba} = g^{ab} = (g^a)^b = X^b$



To make it hard to calculate the original random numbers from the values sent by Alice and Bob, the exponentiation is calculated modulo p (like g , the value p is agreed between Alice and Bob in advance and can be considered public knowledge.)

Alice and Bob first agree upon a generator value g and modulus p , which can be publicly known. Alice (respectively Bob) then chooses a random number a (resp. b) and sends the value $X=g^a \bmod p$ (resp. $Y=g^b \bmod p$) to Bob (resp. Alice). Alice then calculates $Y^a \bmod p$ and

Bob calculates $X^b \bmod p$, which turn out to be the same number because $Y^a \bmod p = (g^b)^a \bmod p = g^{ba} \bmod p = g^{ab} \bmod p = (g^a)^b \bmod p = X \bmod p$



Calculate the value Alice and Bob agree via Diffie Hellman Key Exchange assuming the following parameters:

Public parameters:

Generator $g = 7$

Modulus $p = 41$

Private parameters:

Alice's private number $a = 9$

Bob's private number $b = 14$

Most scientific calculators will have dedicated modulus (mod) and exponentials (x^y) buttons.

For example, Microsoft Calculator (installed by default on Windows) can be used in the labs.

> Calculate $X = 7^9 \bmod 41$

= 13

> Calculate $Y = 7^{14} \bmod 41$

= 2

> Calculate $Z = Y^9 \bmod 41$

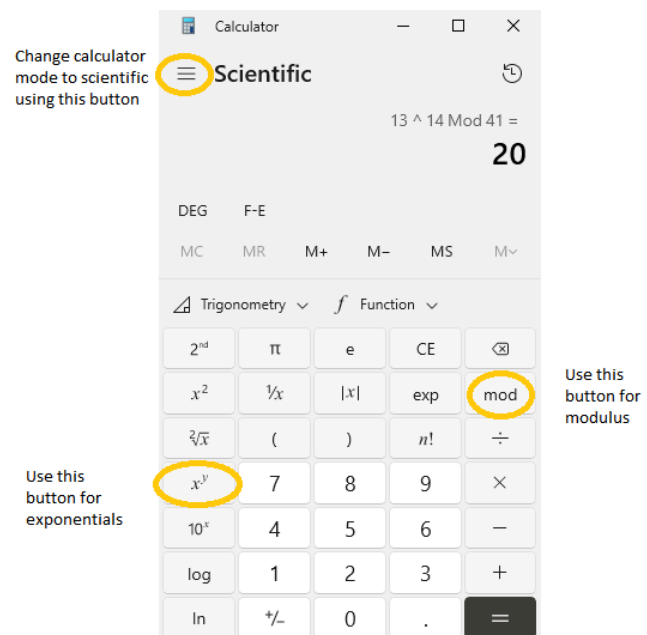
= 20

> Calculate $Z = X^{14} \bmod 41$

= 20

Alice and Bob have agreed upon the shared value 20.

Alice and Bob wished to exchange a shared secret, so they communicated back and forth until they arrived at a shared secret value. A third party eavesdropping on this exchange should find it computationally infeasible (in practice much larger values need to be used) to compute the same value from the information sent over the public channel.



Elliptic Curve Diffie-Hellman Key Exchange

Diffie-Hellman (DH) Key Exchange worked by mixing together three numbers: a publicly known generator; a random number chosen by Alice; and a random number chosen by Bob. Alice and Bob each shared two of these numbers mixed together (using exponentiation) such that they could both calculate all three numbers mixed together (using exponentiation). To make it hard for an eavesdropper to 'unmix' mixed values to uncover the three numbers mixed together, we used modular arithmetic.

Elliptic Curve Diffie-Hellman (ECDH) Key Exchange works in a similar way. Alice and Bob mix together three values: a publicly known generator coordinate (x,y); a random number chosen by Alice; and a random number chosen by Bob. Alice and Bob each share their random number and the coordinate (x,y) mixed together (using scalar multiplication) such that they can both calculate all three values mixed together (using scalar multiplication). To make it hard for an eavesdropper to 'unmix' mixed values to uncover the three values mixed together, we use modular arithmetic.

To understand the mixing in DH Key Exchange, we needed to know how to apply exponentiation (which was just a shorthand way of writing a number of multiplications all at once).

To understand the mixing in ECDH Key Exchange, we need to know how to apply Scalar Multiplication of points, which will just be a shorthand way of writing a number of point additions all at once. So let's start by explaining how two points of an elliptic curve are added together.

Elliptic Curves in \mathbb{R}

An elliptic curve is characterised by the equation $y^2 = x^3 + (a \cdot x) + b$ where a and b are arbitrary constants but for which $4a^3 + 27b^3 \neq 0$.

This just means that the equation for each value of x , we can calculate one or two y values for which the equation holds.

For example, consider the curve with constants, $a = -7$ and $b = 10$.

Where $x = -3$

$$y^2 = (-3)^3 + (-7 \cdot -3) + 10$$

so $y = \sqrt{(-27+21+10)} = \sqrt{4} = 2$

or -2

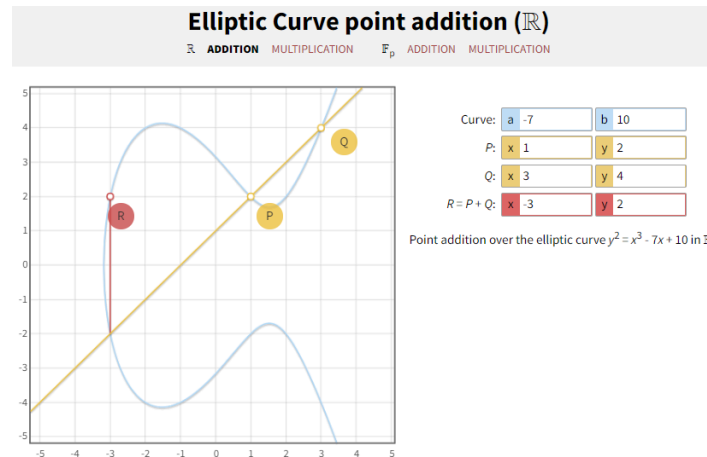
So there are points on the curve at the (x,y) coordinates (-3,2) and (-3,-2).

In general, the set of all points in the curve is specified as:

$$\{(x, y) \in \mathbb{R}^2 \mid y^2 = x^3 + (a \cdot x) + b, 4a^3 + 27b^3 \neq 0\} \cup \{\infty\}$$

View the Elliptic curve $y^2 = x^3 - (7x) + 10$ (i.e, the elliptic curve where $a = -7$ and $b = 10$)

> Type the values $a=-7$ and $b=10$ into [this point addition tool](#) and press Enter



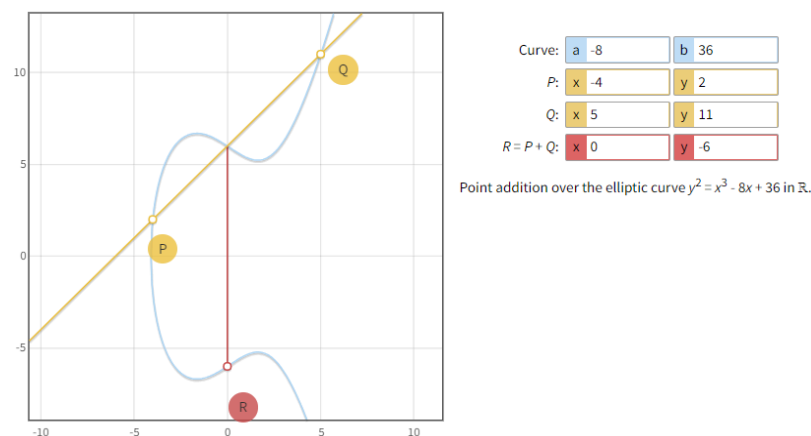
Point Addition (of distinct points that do not share x values) in \mathbb{R}

For almost all pairs of points on the curve, drawing a straight line that crosses the curve at those two points (call them (x_P, y_P) and (x_Q, y_Q)) will also cross the curve at a third point.

The negation of that third point gives another point on the curve (as the curve is reflected across the x-axis). We call this point (x_R, y_R) , which is the result of adding points (x_P, y_P) and (x_Q, y_Q) .

Add points $(-4,2)$ and $(5,11)$ over the Elliptic curve curve $y^2 = x^3 - 8x + 36$ in \mathbb{R} .

- > Enter the values $a = -8$ and $b = 36$ into the blue boxes provided.
- > Type the values $x = -4$ and $y = 2$ for point P in the first of the yellow boxes provided.
- > Type the values $x = 5$ and $y = 11$ for point Q in the second of the yellow boxes provided.
- > Press Enter



Andrea Corbellini's tool then presents the elliptic curve $y^2 = x^3 - 8x + 36$ in \mathbb{R} .

The points labelled in yellow $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ are the two we are adding together.

The straight line that passes between those two points is also drawn in yellow and it crosses the curve at a third point $(0, 6)$. It is the negation of that point (i.e., the reflection of that point across the x-axis) that we call the point $R = (x_R, y_R)$, $= (x_P, y_P) + (x_Q, y_Q)$.

Here $(-4, 2) + (5, 11) = (0, -6)$ because the yellow line crossing $(-4, 2)$ and $(5, 11)$ also crosses at a third point $(0, 6)$ and the negation of that point is $(0, -6)$ (the red line represents the negation, it shows the reflection of the point across the x-axis.)

Point Addition (of distinct points that DO share x values) in \mathbb{R}

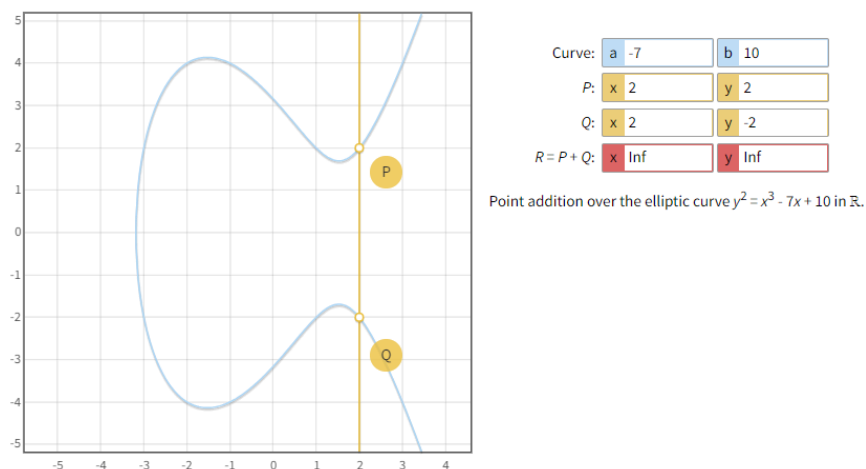
So far we have only added distinct coordinates that do NOT share the same value for x.

Drawing a straight line through any two points (P and Q) on an elliptic curve that DO share x values will not cross the line a third time for any rational extension of the line in either direction. It is said that the line crosses only at the 'point at infinity.' This is the same as when we say parallel lines never meet, or at least they meet only at the 'point at infinity'.

The 'point at infinity' is a special point in addition to the points represented by the coordinates (x,y) for any values x and y. We'll write ∞ to represent the 'point at infinity'.

Add points (2,2) and (2,-2) over the Elliptic curve $y^2 = x^3 - 7x + 10$ in \mathbb{R} .

- > Enter the values $a = -7$ and $b = 10$ into the blue boxes provided.
- > Type the values $x = 2$ and $y = 2$ for point P in the first of the yellow boxes provided.
- > Type the values $x = 2$ and $y = -2$ for point Q in the second of the yellow boxes provided.
- > Press Enter



The tool represents the point at infinity ∞ as the (x,y) coordinate where $x = \text{Inf}$ and $y = \text{Inf}$

Point Negation in \mathbb{R} and the Identity Element (point at infinity)

The negation of a point (x_P, y_P) is the point $-(x_P, y_P)$, such that adding the two points together returns the identity element ∞ , also known as the point at infinity, i.e.,

$$(x_P, y_P) + (-(x_P, y_P)) = \infty$$

To negate the point (x_P, y_P) we just negate its y value, i.e., $(x_P, -y_P)$, because:

$$\begin{aligned}(x_P, y_P) + (-(x_P, y_P)) &= \infty \\(x_P, y_P) + (x_P, -y_P) &= \infty \\(x_P, -y_P) &= \infty - (x_P, y_P) \\(x_P, -y_P) &= -(x_P, y_P)\end{aligned}$$

$$\text{e.g., } -(1, 2) = (1, -2)$$

Adding ∞ to any point (x_P, y_P) results in that point itself (including adding point at infinity to itself. That is:

$$\begin{aligned}\infty + (x_P, y_P) &= (x_P, y_P) \\ \infty + \infty &= \infty\end{aligned}$$

$$\text{e.g., } \infty + (1, 2) = (1, 2)$$

Point Addition (of a point with itself, i.e., doubling a point) in \mathbb{R}

As the two points on a curve get closer together the line passing through both points gets ever closer to the tangent of the line at either of the points. Should the distance between the points diminish to nothing then the two points are equal; the tangent is given as the point addition of a point with itself.

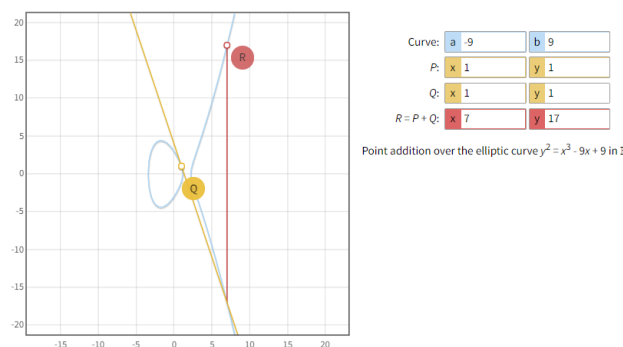
Add point (1,1) to itself over the Elliptic curve $y^2 = x^3 - 9x + 9$ in \mathbb{R} .

> Enter the values $a = -9$ and $b = 9$ into the blue boxes provided.

> Type the values $x = 1$ and $y = 1$ for point P in the first of the yellow boxes provided.

> Type the values $x = 1$ and $y = 1$ for point Q in the second of the yellow boxes provided.

> Press Enter



Scalar Multiplication in \mathbb{R}

We are familiar with multiplication of numbers being the addition of some number to itself some other number of times. For example, the two times table in terms of addition progresses as follows:

$$\begin{array}{ll} 0 = 0 * 2 & \text{two added to itself no times is } 0 * 2 \\ 2 + 0 = 1 * 2 & \text{two added to itself once is } 1 * 2 \\ 2 + 2 + 0 = 2 * 2 & \text{two added to itself twice is } 2 * 2 \\ 2 + 2 + 2 + 0 = 3 * 2 & \text{two added to itself thrice is } 3 * 2 \end{array}$$

Scalar multiplication is defined similarly in terms of point addition: scalar multiplication of points being the addition of some point to itself some other number of times. For example, multiplications of the point (1,2) in terms of point addition progresses as follows:

$$\begin{array}{ll} \infty = 0 \cdot (1, 2) & (1,2) \text{ added to itself no times is } 0 \cdot (1,2) \\ (1, 2) + \infty = 1 \cdot (1, 2) & (1,2) \text{ added to itself once is } 1 \cdot (1,2) \\ (1, 2) + (1, 2) + \infty = 2 \cdot (1, 2) & (1,2) \text{ added to itself twice is } 2 \cdot (1,2) \\ (1, 2) + (1, 2) + (1, 2) + \infty = 3 \cdot (1, 2) & (1,2) \text{ added to itself thrice is } 3 \cdot (1,2) \end{array}$$

In general we have:

$$\begin{array}{l} \infty = 0 \cdot (x_P, y_P) \\ (x_P, y_P) + \infty = 1 \cdot (x_P, y_P) \\ (x_P, y_P) + (x_P, y_P) + \infty = 2 \cdot (x_P, y_P) \\ (x_P, y_P) + (x_P, y_P) + (x_P, y_P) + \infty = 3 \cdot (x_P, y_P) \end{array}$$

For example, the coordinate of the point addition of (1, 2) and (1, 2) over the Elliptic curve $y^2 = x^3 - (7 \cdot x) + 10$ is (-1, -4). The tangent line at $P = Q = (1, 2)$ also crosses at another point (-1, 4); the reflection of that point across the x-axis is (-1, -4). The coordinate of the point addition of (1, 2) and (-1, -4) over the Elliptic curve $y^2 = x^3 - (7 \cdot x) + 10$ is (9, -26) as the line through $P=(1, 2)$ and $Q=(-1, -4)$ also crosses at the third point (9, 26); the reflection of that point across the x-axis is (9, -26).

Determine the coordinate of the point multiplication of $3 \cdot (1,2)$ over the Elliptic curve $y^2 = x^3 - (7 \cdot x) + 10$

> Type the values a=-7 and b=10 into [this scalar multiplication tool](#)

> Also type the values n=3 and x=1 and y=2

> Press Enter

This matches our expectation from having arrived at the same value through successive point additions in [this point addition tool](#).

If we look again at the point additions performed above we determined

$$3 \cdot (1, 2) = (1, 2) + (1, 2) + (1, 2) = (1, -4) + (1, 2) = (9, -26)$$

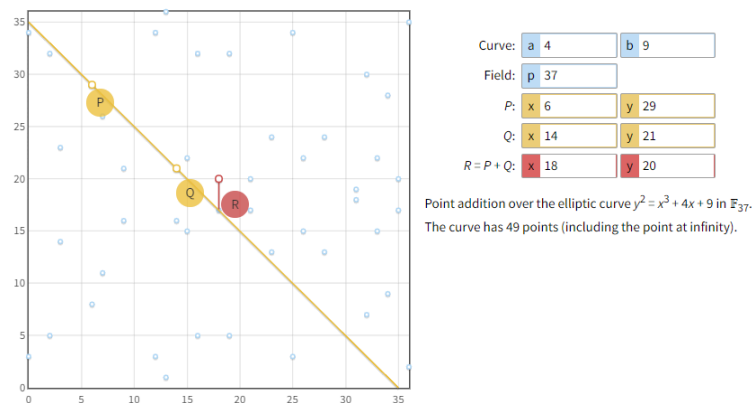
Point Addition and Scalar Multiplication in \mathbb{F}_p

We have just seen that within Elliptic Curve Diffie Hellman (ECDH) Key Exchange the mixed values sent between Alice and Bob shall be the results of scalar multiplication. To make it hard to calculate the original random numbers from the values sent by Alice and Bob, the scalar multiplication is calculated modulo p (the value p is agreed between Alice and Bob in advance and can be considered public knowledge.)

The set of all points in the curve is then specified as:

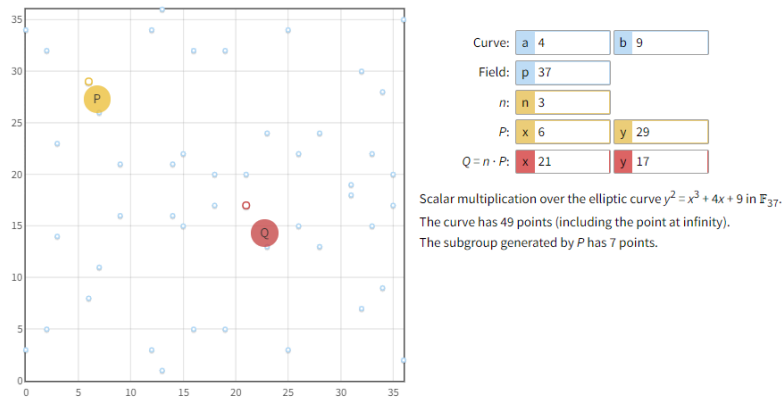
$$\{(x, y) \in \mathbb{R}^2 \mid y^2 = x^3 + (a \cdot x) + b \bmod p, 4a^3 + 27b^3 \neq 0 \bmod p\} \cup \{\infty\}$$

For example, adding points $P = (6, 29)$ and $Q = (14, 21)$ over the Elliptic curve $y^2 = x^3 + 4x + 9$ in \mathbb{F}_{37} (i.e., Alice and Bob agree to modulus $p=37$) returns the point $(18, 20)$.



Determine the coordinate of the scalar multiplication $3 \cdot (6, 29)$ over the Elliptic curve $y^2 = x^3 + (4 \cdot x) + 9$ in \mathbb{F}_{37}

- > Enter the values $a = 4$ and $b = 9$ into the blue boxes to specify the curve.
- > Type the value $p = 37$ in the blue box to specify for the field.
- > Type the value $n = 3$.
- > Type the values $x = 6$ and $y = 29$ for point P in the second set of yellow boxes provided.
- > Press Enter

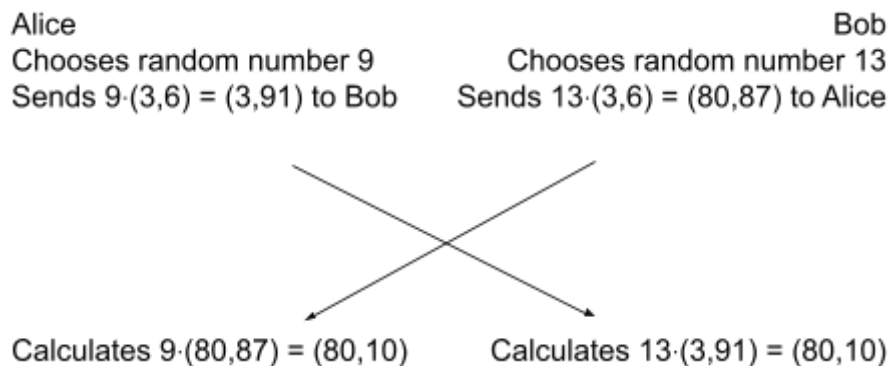


Elliptic Curve Diffie Hellman Key Exchange

Diffie Hellman Key Exchange proceeds in a similar way as before. Alice and Bob calculate a value based on a publicly known constant value and their chosen random number and send it to each other. They then perform a subsequent calculation on what they received from the other participant with their chosen random number to arrive at a secret shared between the participants, which would be hard for an eavesdropper to also calculate based on the publicly shared information.

For example, Alice and Bob may agree the following shared secret using ECDH with the Elliptic Curve $y^2 = x^3 + 2x + 3$ in \mathbb{F}_{97} , Generator point $P = (3, 6)$:

Alice and Bob agree $a=2$, $b=3$, $P = (3,6)$ and $p = 97$



Calculate the value Alice and Bob agree via Elliptic Curve Diffie Hellman Key Exchange assuming the following parameters:

Public parameters:

Elliptic Curve $y^2 = x^3 + 2x + 3$ in \mathbb{F}_{97}

Generator point $P = (3,6)$

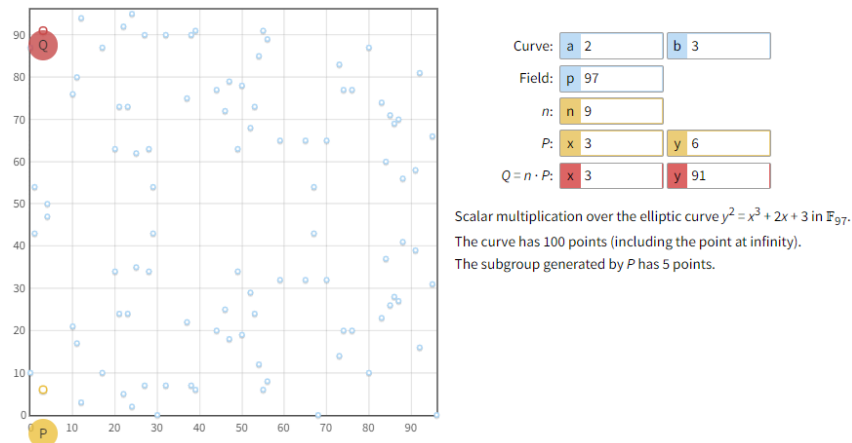
Private parameters:

Alice's private number $a = 9$

Bob's private number $b = 13$

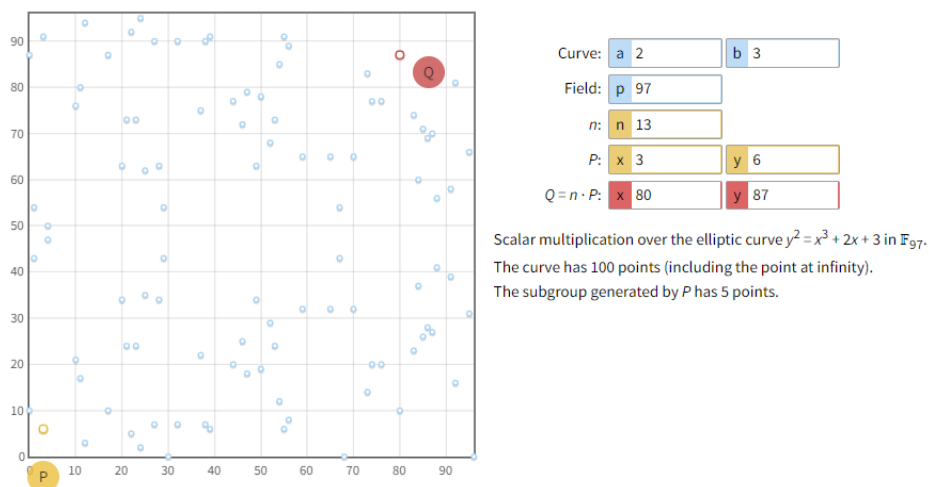
We'll calculate the following in terms of what Bob calculates based on what they receive from Alice, as well as in terms of what Alice calculates based on what they receive from Bob. The result of the calculation of Alice and Bob should match at the end of the protocol run.

- > Type the values $a = 2$, $b = 3$ and $p = 97$ into [this scalar multiplication tool](#)
(we read $a = 2$, $b = 3$ and $p = 97$ directly from the definition of the curve)
- > Type Alice's random value 9 as the multiplier i.e, $n = 9$
- > Type the values $x_P = 3$ and $y_P = 6$ for the generator point P.
- > Press Enter



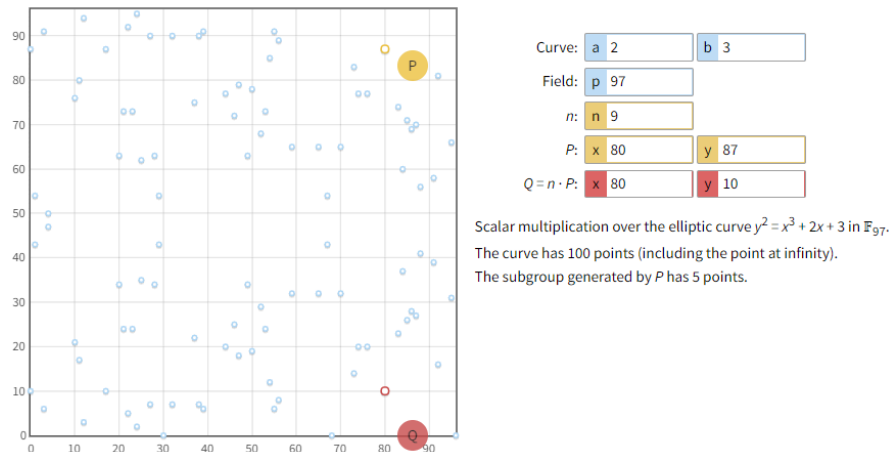
The point that Alice sends to Bob over the public network is $(3, 91)$.

- > Keep the values $a = 2$, $b = 3$ and $p = 97$
- > Type Bob's random value 13 as the multiplier i.e, $n = 13$
- > Keep the values $x_P = 3$ and $y_P = 6$ for the generator point P.
- > Press Enter



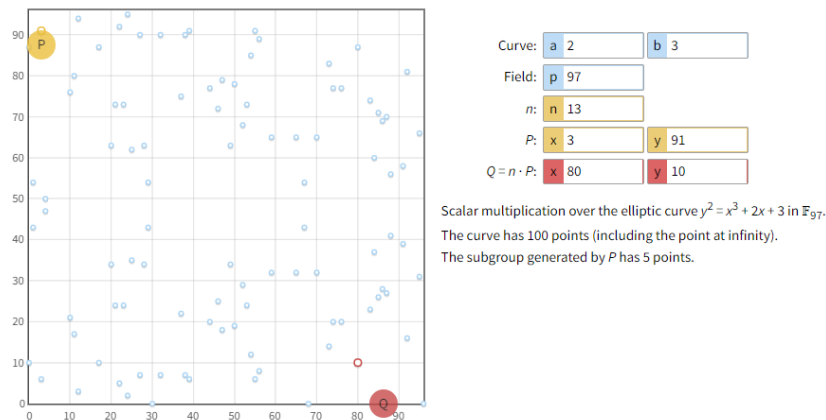
The point that Bob sends to Alice over the public network is $(80, 87)$.

- > Keep the values $a = 2$, $b = 3$ and $p = 97$
- > Type Alice's random value 9 as the multiplier i.e, $n = 9$
- > Type the values $x_P = 80$ and $y_P = 87$ for the point Alice received from Bob.
- > Press Enter



The point that Alice calculates as the shared secret is $(80, 10)$.

- > Keep the values $a = 2$, $b = 3$ and $p = 97$
- > Type Bob's random value 13 as the multiplier i.e, $n = 13$
- > Type the values $x_P = 3$ and $y_P = 91$ for the point Bob received from Alice.
- > Press Enter



The point that Bob calculates as the shared secret is also $(80, 10)$.

Alice and Bob wished to exchange a shared secret, so they communicated back and forth until they arrived at a shared secret point on the elliptic curve. A third party eavesdropping on this exchange should find it computationally infeasible (in practice much larger values need to be used) to compute the same point from the information sent over the public channel.

Example 2: 3-Message NSPK Protocol Analysis

The Needham-Schroeder Public Key Protocol aims to achieve mutual authentication between the protocol initiator and responder. It was specified in the following paper:

[Using encryption for authentication in large networks of computers](#)

An attack on the protocol, in which a responder is able to impersonate another agent, was published (over 15 years later) in the following paper:

[An attack on the Needham-Schroeder public-key authentication protocol](#)

Given in common syntax the 3-message NSPK Protocol is

```
A, B, S :    Principal
Na, Nb :    Nonce
pk :        Principal -> Key
```

1. A->B: {Na, A}pk(B)
2. B->A: {Na, Nb}pk(A)
3. A->B: {Nb}pk(B)

3-message NSPK Protocol Description

Message 1

The initiator of the protocol sends a random number used once challenge to the responder. A sends their identity A along with a random number used once Na freshly generated by A encrypted using the public key of B. As B is the only entity able to decrypt the message, receipt of a subsequent response to this challenge in which the random number used once appears will attest to B having participated in the protocol run.

Message 2

The responder of the protocol responds to the earlier challenge of the first message by returning to the initiator the value B received in that challenge, namely Na. Encrypting the value ensures the secrecy of the random number beyond the end of the protocol, so it could be used as a seed from which to generate a symmetric key used for subsequent bulk encryption between A and B.

The responder of the protocol also sends a random number used once challenge to the responder. B also sends a random number used once freshly generated by B for the second message encrypted using the public key of A. As A is the only entity able to decrypt the message, receipt of a subsequent response to this challenge in which the random number appears will attest to A having participated in the protocol run.

Message 3

Finally, the initiator of the protocol responds to the earlier challenge of the second message by returning to the responder the value A received in that challenge, namely Nb. Encrypting

the value aims to ensure the secrecy of the random number used once beyond the end of the protocol, so that it may be used as a seed from which to generate a symmetric key used for subsequent bulk encryption between A and B.

Open a browser, navigate to appsanywhere.port.ac.uk and launch Scyther

Click File -> Open to open ns3.spdl

Specification of the 3-message NSPK Protocol in SPDL

The outer parentheses of the specification encapsulates the protocol specification labelled 'ns3' and shall include a role for each of the protocol participants, namely the initiator I and responder R .

```
protocol ns3(I,R)
{
    role I
    {
        ...
    }

    role R
    {
        ...
    }
}
```

Within the messages exchanged within a run of the protocol, the initiator I will send a (random number used once) challenge to authenticate the responder, so I generates a fresh random number used once N_i .

```
role I
{
    fresh Ni: Nonce;
```

The initiator must later be ready to receive a (random number used once) challenge from R .

```
var Nr: Nonce;
```

The initiator role specifies that the initiator participates in all three messages of the protocol alternating between sending and receiving each message.

```
send_1(I,R,{Ni,I}pk(R));
recv_2(R,I,{Ni,Nr}pk(I));
send_3(I,R,{Nr}pk(R));
```

```
}
```

Within the messages exchanged within a run of the protocol, the responder R will send a (random number used once) challenge for which R generates a fresh random number used once Nr .

```
role R
{
    fresh Nr: Nonce;
```

The responder must be ready to receive a (random number used once) challenge from I .

```
var Ni: Nonce;
```

The responder role specifies that the responder participates in all three messages of the protocol alternating between receiving and sending each message.

```
recv_1(I, R, {Ni, I}pk(R));
send_2(R, I, {Ni, Nr}pk(I));
recv_3(I, R, {Nr}pk(R));
}
```

Verification of the 3-message NSPK Protocol in Scyther

Click Verify -> Verify automatic claims (or use shortcut key F6)

The protocol does not guarantee to Alice in role R that completing a run of the protocol believing to be communicating with Bob means that Bob has been running the protocol believing to be communicating with Alice.

We'll soon come to know this property (that the protocol fails to achieve) as Weak Agreement. There are a number of notions of authentication, each stronger than the last, of which the 3-message NSPK Protocol only satisfies the weakest of these notions of authentication, namely Aliveness. The protocol satisfies aliveness to the responder, as upon the responder's completion of the protocol in which they believe they have been communicating with the initiator, then the initiator participated in at least one message of the protocol.

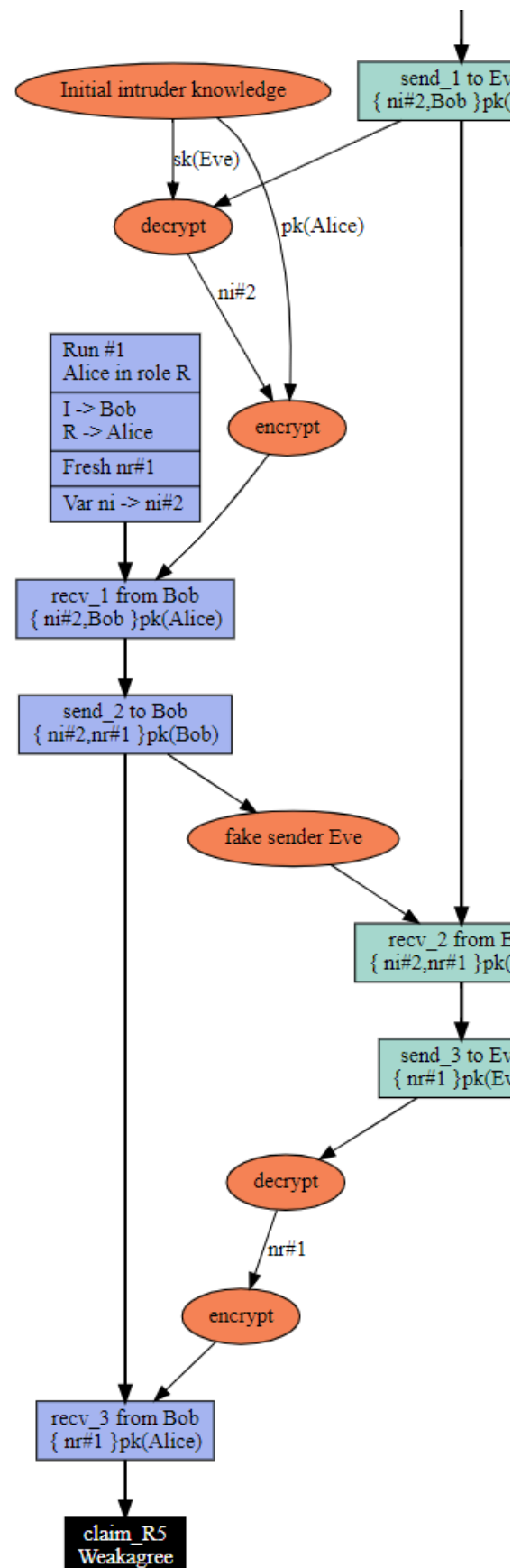
	Aliveness	Weak Agreement	Non-injective Agreement	Non-injective Synchronisation	Secret Ni	Secret Nr
I	Ok	Ok	Ok	Ok	Ok	Ok
R	Ok	Fail	Fail	Fail	Fail	Fail

The protocol does not guarantee to Alice in role R that completing a run of the protocol believing to be communicating with Bob means that Bob has been running the protocol believing to be communicating with Alice (otherwise known as Weak Agreement).

The counter example shows two runs of the protocol, one in which Bob in the initiator role completes a run with Eve, who is then able to complete a run of the protocol with Alice pretending to be Bob. Alice and Bob each complete one run of the protocol, but they disagree on who they think they were communicating with; Alice thinks they were communicating with Bob when Bob thinks they were communicating with Eve.

In Run #2, Bob takes the role of initiator in a protocol run with Eve. Eve acts honestly in the protocol run with Bob, sending and receiving each message in turn acting as herself. However, Eve is a pseudonym of the intruder, who acts dishonestly in a second run of the protocol with Alice.

In Run #1, the intruder pretends to be Bob in initiating a protocol with responder Alice. Instead of generating a fresh random number to provide Alice in the first message of Run#1, the intruder instead decrypts the random number received (as Eve) in the first message of Run#2, namely $nr\#2$. Alice, believing that they are responding to Bob, sends the answer to the random number used once challenge along with their own random number used once challenge, encrypted using the public key of Bob. The intruder does not have knowledge of the decryption key needed to decrypt this message, but they can forward the message onto Bob, passing it off as their own response to Bob's challenge in protocol Run#2. As this response contains the correct answer to Bob's challenge, Bob is satisfied they have been communicating with Eve (they are) and so responds to the challenge that apparently Eve set in Run#2, but in reality it was really a challenge set by Alice in Run#1.



Bob's response in the final message of Run#1 has the information Eve needs to convince Alice that they have been communicating with Bob, when Bob has finished a run of the protocol thinking that they have been communicating with Eve. The intruder is able to decrypt the message sent to Eve, a pseudonym under their control, from which they learn the random number used once to answer Alice's challenge in the final message of Run#2.

Example 3: Full NSPK Protocol Analysis

The previous example considered the 3-message version of the Needham-Schroeder Public Key (NSPK) Protocol, in which the initiator and responder are expected to know each other's public keys in advance of the protocol run. We have witnessed that even in that setting the protocol fails to fulfil its requirements.

The full NSPK protocol does not assume the initiator and responder know each other's public keys in advance; an authentication server distributes public keys by signing a message binding an agent's identity to their public key.

As the 3-message fails to fulfil its requirements, so too does the full NSPK protocol. This example shows how to extend a protocol specification to include a third party authentication server.

Given in common syntax the full NSPK Protocol is.

```
A,B,S :    Principal
Na,Nb :    Nonce
pk :       Principal -> Key
sk :       Principal -> Key
```

1. A->S: A,B
2. S->A: {B,pk(B)}sk(S)
3. A->B: {A,Na}pk(B)
4. B->S: B,A
5. S->B: {A,pk(A)}sk(S)
6. B->A: {Na,Nb}pk(A)
7. A->B: {Nb}pk(B)

Full NSPK Protocol Description

Messages 1 and 2

The initiator of the protocol, A, first sends a certificate request to an authentication server, S, regarding the intended responder of the protocol, B. The request comprises the protocol participants' identities (A and B) in plaintext. S responds by providing A with B's digital certificate, which binds the responder's identity with their public key. The digital certificate is signed by S.

Message 3

The initiator of the protocol next sends a random number used once challenge to the

responder. A sends their identity A along with a random number freshly generated by A encrypted using the public key of B . As B is the only entity able to decrypt the message, receipt of a subsequent response to this challenge in which the random number appears will attest to B having participated in the protocol run.

Messages 4 and 5

The responder of the protocol, B , now sends a certificate request to the authentication server, S , regarding the apparent initiator of the protocol, A . The request comprises the protocol participants' identities (B and A) in plaintext. S responds by providing B with A 's digital certificate signed by S .

Message 6

The responder of the protocol responds to the earlier challenge of the third message by returning to the initiator the value B received in that challenge, namely N_a . Encrypting the value aims to ensure the secrecy of the random number used once beyond the end of the protocol, such that it may be used as a seed from which to generate a symmetric key used for subsequent bulk encryption between A and B .

The responder of the protocol also sends a random number used once challenge to the responder. B also sends a random number freshly generated by B in the sixth message encrypted using the public key of A . As A is the only entity able to decrypt the message, receipt of a subsequent response to this challenge in which the random number used once appears will attest to A having participated in the protocol run.

Message 7

Finally, the initiator of the protocol responds to the earlier challenge of the sixth message by returning to the responder the value A received in that challenge, namely N_b . Encrypting the value aims to ensure the secrecy of the random number used once beyond the end of the protocol, so it too could be used as a seed from which to generate a symmetric key used for subsequent bulk encryption between A and B .

Specification of the full NSPK Protocol in SPDL

Revise the outer parentheses of ns3 to label the protocol ' $ns7$ ' and add a role for the extra protocol participant ' S '.

```
protocol ns7(I, R, S)
{
  role I
  {
    ...
  }
}
```

```

role R
{
    ...
}

```

```

role S
{
    ...
}

```

Within the role I specified for the initiator...

Revise the labels for messages 1, 2 and 3 (which are now messages 3, 6 and 7 of the full protocol) and add the initiator's request for the responder's public key and the authentication server's subsequent response (these become the new message 1 and message 2 of the protocol).

```

send_1(I, S, (I, R));
recv_2(S, I, {pk(R), R}sk(S));
send_3(I, R, {Ni, I}pk(R));
recv_6(R, I, {Ni, Nr}pk(I));
send_7(I, R, {Nr}pk(R));

```

Within the role R specified for the responder...

Revise the labels for messages 1, 2 and 3 (which are now messages 3, 6 and 7 of the full protocol) and add the responder's request for the initiator's public key and the authentication server's subsequent response (these become the new message 4 and message 5 of the protocol).

```

recv_3(I, R, {Ni, I}pk(R));
send_4(R, S, (R, I));
recv_5(S, R, {pk(I), I}sk(S));
send_6(R, I, {Ni, Nr}pk(I));
recv_7(I, R, {Nr}pk(R));

```

Within the role S (newly) specified for the authentication server...

Add each agent's requests for the public keys and the authentication server's subsequent responses for the initiator and responder in turn.


```

role S
{
    recv_1 (I, S, (I, R)) ;
    send_2 (S, I, {pk(R), R} sk(S)) ;
    recv_4 (R, S, (R, I)) ;
    send_5 (S, R, {pk(I), I} sk(S)) ;
}

```

Observe that, unlike in ns3.spdl, none of the roles I , R and S in ns7.spdl participate in *all* messages of the protocol. The specification of each role in an SPDL file must specify *only* the messages in which that role sends or receives.

Verification of the full NSPK Protocol in Scyther

Click Verify -> Verify automatic claims (or use shortcut key F6)

Like the 3-message NSPK protocol, the full NSPK protocol fails to achieve Weak Agreement as it fails to guarantee the identity of the initiator to the responder.

	Aliveness	Weak Agreement	Non-injective Agreement	Non-injective Synchronisation	Secret Ni	Secret Nr
I	Ok	Ok	Ok	Ok	Ok	Ok
R	Ok	Fail	Fail	Fail	Fail	Fail

The counterexample should again show two runs of the protocol, one in which an agent in the responder role in one run of the protocol can then impersonate the initiator of that protocol in a second run of the protocol with another responder.

Example 4 - Hierarchy of Authentication Specifications

In this example, we compare various notions of authentication within the hierarchy of authentication specifications.

Aliveness

Aliveness means that when an agent x completes a run of the protocol believing they have been communicating with y , then y has sent at least one message of the protocol.

Note that aliveness may still be satisfied if even where y fails to complete a protocol run. Aliveness may also still be satisfied if y runs the protocol believing they are communicating with an agent that is not x .

Weak Agreement

Weak agreement means that when an agent x completes a run of the protocol believing they have been communicating with y , then y has been running the protocol believing they have been communicating with x .

The following protocol given in common syntax guarantees aliveness to the responder but fails to guarantee weak agreement:

```
A, B :    Principal
Na :      Nonce
sk :      Principal -> Key
```

1. $A \rightarrow B: \{A, Na\}_{sk(A)}$
2. $B \rightarrow A: \{Na, B, A\}_{sk(B)}$

Specify the protocol in SPDL (save it as `alive.spdl`)

```
protocol alive(I,R)
{
  role I
  {
    fresh Ni:Nonce;
    send_1(I,R,{I,Ni}_{sk(I)});
    recv_2(R,I,{Ni,R,I}_{sk(R)});
  }

  role R
  {
    var Ni:Nonce;
    recv_1(I,R,{I,Ni}_{sk(I)});
    send_2(R,I,{Ni,R,I}_{sk(R)});
  }
}
```

Verify the protocol against the automatic claims using Scyther.

The protocol guarantees aliveness to the responder but fails to guarantee weak agreement.

	Aliveness	Weak Agreement	Non-injective Agreement	Non-injective Synchronisation
I	Ok	Ok	Ok	Ok
R	Ok	Fail	Fail	Fail

Describe a counterexample that shows the protocol specified in `alive.spdl` fails to achieve weak agreement from the perspective of the responder.

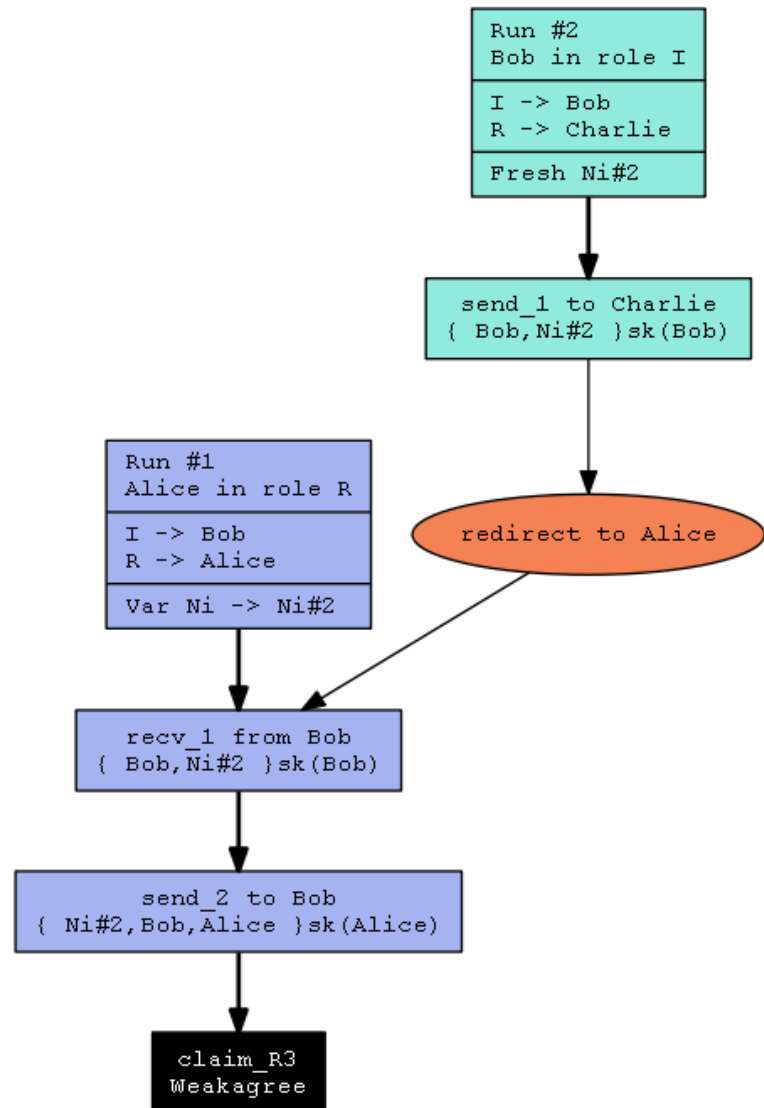
The counterexample shows two runs of the protocol, one in which an agent, Bob, in the initiator role believing they are exchanging messages with the intended recipient Charlie, but in fact the intruder redirects the first message to Alice in the responder role, believing that they are communicating with Bob.

The protocol fails to achieve weak agreement. Weak agreement requires that when Alice completes a run of the protocol believing they have been communicating with Bob, then Bob has been running the protocol believing they have been communicating with Alice. But in the counterexample Bob does not think they have been communicating with Alice; Bob instead believes they have been communicating with Charlie.

The protocol satisfies the Aliveness property, so there is no counterexample demonstrating a failure to achieve the Aliveness property. For the protocol to achieve the Aliveness property, then upon Alice completing their run of the protocol believing they have been communicating with Bob then, Bob must have sent at least one message of the protocol. The counterexample presented is not evidence that the protocol fails to achieve the Aliveness property, as Bob has sent at least one message of the protocol.

Non-injective Agreement

Non-injective agreement means that when an agent x completes a run of the protocol believing they have been communicating with y using a set of data items D , then y has been running the protocol believing they have been communicating with x and is in agreement that the same set of data items D were used in the protocol run.



Note that non-injective agreement does not necessarily mean that there is a one-one relationship between the runs of x and the runs of y (hence the adjective “non-injective”): x may believe that two runs of the protocol were completed, when y has taken part in only a single protocol run.

The following protocol given in common syntax guarantees weak agreement to the responder but fails to guarantee non-injective agreement:

```
A, B :    Principal
Nb :      Nonce
sk       Principal -> Nonce
```

1. $A \rightarrow B$: $\{A, B, Na\}_{sk(A)}$
2. $B \rightarrow A$: $\{Na\}_{sk(B)}, \{A\}_{sk(B)}$

Specify the protocol in SPDL (save it as weakagree.spdl)

```
protocol weakagree(I,R)
{
  role I
  {
    fresh Ni:Nonce;
    send_1(I,R,{I,R,Ni}_{sk(I)});
    recv_2(R,I,{Ni}_{sk(R)},{I}_{sk(R)});
  }

  role R
  {
    var Ni:Nonce;
    recv_1(I,R,{I,R,Ni}_{sk(I)});
    send_2(R,I,{Ni}_{sk(R)},{I}_{sk(R)});
  }
}
```

Verify the protocol against the automatic claims using Scyther.

The protocol guarantees weak agreement to the initiator but fails to guarantee non-injective agreement.

	Aliveness	Weak Agreement	Non-injective Agreement	Non-injective Synchronisation
I	Ok	Ok	Fail	Fail
R	Ok	Ok	Ok	Ok

The counterexample shows Alice participating in two runs of the protocol as responder, one in which they send a signed random number used once Ni#1 to Eve and another in which they send Ni#4 to Bob. Bob completes a single protocol run believing that they have been interacting with Alice, upon receipt of a random number used once Ni#1 signed by Alice.

The protocol fails to achieve non-injective agreement. Non-injective agreement requires that when Bob completes a run of the protocol believing they have been communicating with Alice using Ni#1, then Alice has been running the protocol believing they have been communicating with Bob also using Ni#1. However, in the protocol run in which Alice believes they have been communicating with Bob, it is Ni#4 that Alice sent (not Ni#1) and in the protocol run where Alice did send Ni#1, Alice believed they were communicating with Eve (not Alice).

The protocol satisfies the Weak Agreement property. For the protocol to achieve the Weak Agreement property, then upon Bob completing their run of the protocol they have been communicating with Alice, then Alice must have been running the protocol believing they have been communicating with Bob. The counterexample presented is not evidence that the protocol fails to achieve the Weak Agreement property, as both Alice and Bob to complete their protocol runs believing they have communicated with the other (they just disagree on the random number used once value that they communicated) because of the manipulation of the message exchange by an intruder (able to adopt the pseudonym Eve).

Non-injective Synchronisation

[Non-injective synchronisation](#) means that when an agent x completes a run of the protocol there exist runs for the other roles in the protocol, such that all communications of the protocol must have occurred correctly within these runs.

The following protocol given in common syntax guarantees non-injective agreement to the responder but fails to guarantee non-injective synchronisation.

```
A, B :      Principal
Nb :       Nonce
pk :       Principal -> Key
```

1. A->B: {A,B}pk(B)
2. B->A: {B,Nb}pk(A)
3. A->B: {Nb,A}pk(B)

Verification in Scyther shows that from the perspective of a responder the protocol guarantees non-injective agreement but fails to guarantee non-injective synchronisation.

	Aliveness	Weak Agreement	Non-injective Agreement	Non-injective Synchronisation
I	Fail	Fail	Fail	Fail
R	Ok	Ok	Ok	Fail

Example 5: NSSK Protocol Analysis

The Needham-Schroeder Symmetric Key (NSSK) Protocol aims to achieve mutual authentication between the protocol initiator and responder, but using symmetric key cryptography rather than using public keys. It was specified in the same paper as the NSPK Protocol:

[Using encryption for authentication in large networks of computers](#)

Given in common syntax the NSSK Protocol is:

```
A, B, S :      Principal
Na, Nb :      Nonce
Kas, Kbs, Kab :      Key
dec :         Nonce -> Nonce

1. A->S: A, B, Na
2. S->A: {Na, B, Kab, {Kab, A}Kbs}Kas
3. A->B: {Kab, A}Kbs
4. B->A: {Nb}Kab
5. A->B: {dec(Nb)}Kab
```

Note that A is unable to decrypt $\{Kab, A\}Kbs$ encapsulated within message 2 as they do not hold the decryption key Kbs shared between B and S.

NSSK Protocol Description

Rather than to certify public keys of protocol participants, as was the role of the authentication server in the full NSPK protocol, the authentication server in the NSSK protocol generates session keys. A session key is a short lived symmetric key used by two agents to communicate for a period of time called a session.

Messages 1 and 2

The initiator of the protocol A first sends a freshly generated random number used once Na to the authentication server S along with their own identity A and the identity of the responder B, with whom they wish to communicate and authenticate. The authentication server returns a freshly generated session key Kab for use by A and B encapsulated in a message encrypted using a long term symmetric key shared between A and S. Within that encrypted message S also include the random number used once Na to indicate the protocol run to which this response is attributed, the identity of B and a portion of the message encrypted using a long term symmetric key shared between B and S that A cannot decrypt, $\{Kab, A\}Kbs$. We'll call an encrypted portion of a message for which the recipient doesn't hold the decryption key a 'ticket'. The intention is not for A to ever decrypt the ticket, but simply to forward the ticket to an agent that can for their own consumption.

Message 3

The initiator of the protocol next sends the ticket received in the previous message $\{K_{ab}, A\}_{K_{bs}}$ to the responder B.

Messages 4 and 5

The responder of the protocol B now sends a freshly generated random number used once challenge to A. The initiator responds by decreasing the random number used once received by one and returning it to the responder. Both messages 4 and 5 are encrypted using the session key K_{ab} generated by the authentication server and received by A and B in messages 2 and 3, respectively.

Specification of the NSSK Protocol in SPDL

In the final step the initiator is required to decrement the random number used once received from the responder by one, for this we specify the decrement and increment functions `dec` and `inc` and specify that they are the inverse of each other.

```
const dec,inc: Function;
inversekeys(dec,inc);
```

Specify the outer parentheses of nssk with roles for the three protocol participants.

The outer parentheses of the specification encapsulates the protocol specification labelled 'nssk' and shall include a role for each of the protocol participants I , R , and S .

```
protocol nssk(I,R,S)
{
  role I
  {
    ...
  }
  role R
  {
    ...
  }
  role S
  {
    ...
  }
}
```

Specify the initiator role of nssk

Within the messages exchanged within a run of the protocol, the initiator I will send a random number used once challenge to authenticate the authentication server for which I generates a fresh random number used once N_i .

```

role I
{
    fresh Ni: Nonce;

```

The initiator must later be ready to receive a (random number used once) challenge from R for which R generates a fresh random number used once N_r .

```

    var Nr: Nonce;

```

During the message exchange, I should also expect to receive the symmetric session key K_{ir} , generated by the authentication server S for subsequent encryption of bulk data communications between I and R .

```

    var Kir: SessionKey;

```

In the third message of the protocol, the initiator should expect to receive some encrypted value for which they do not hold the decryption key. This is a ticket given to I by the authentication server S that I will be able to forward to R in a subsequent message of the protocol. As I is unable to decrypt the ticket, they remain unaware of the message contents (they are also unable to validate the number/type of elements included within the encrypted ticket.) instead they receive only some seemingly random value, specified as follows:

```

    var T: Ticket;

```

The initiator role specifies that the initiator participates in all five messages of the protocol alternating between receiving and sending each message.

```

    send_1(I, S, (I, R, Ni));
    recv_2(S, I, {Ni, R, Kir, T}k(I, S));
    send_3(I, R, T);
    recv_4(R, I, {Nr}Kir);
    send_5(I, R, {dec(Nr)}Kir);
}

```

send_3 above specifies that I sends to R whatever value T they received from S in `recv_2` without knowing of what T comprises.

Specify the responder role of nssk

Within the messages of a protocol run, the responder R will send a random number used once challenge for which R generates a fresh random number used once N_r .

```

role R
{
    fresh Nr: Nonce;

```


During the message exchange, R should also expect to receive the symmetric session key K_{ir} , generated by the authentication server S for subsequent encryption of bulk data communications between I and R .

```
var Kir: SessionKey;
```

The responder role specifies that the responder participates in the final three messages of the protocol alternating between receiving and sending each message.

```
recv_3(I, R, {Kir, I}k(R, S));
send_4(R, I, {Nr}Kir);
recv_5(I, R, {dec(Nr)}Kir);
}
```

Specify the authentication server role of nssk

Within the messages of a protocol run, the authentication server S will receive a random number used once from each of the other protocol participants within messages from each encrypted with the symmetric keys the participants share with the authentication server.

```
role S
{
    var Ni, Nr: Nonce;
```

The fresh symmetric session key used by I and R is generated by the authentication server.

```
fresh Kir: SessionKey;
```

The authentication server participates in only the first and second message of the protocol receiving a fresh session key request from I to which S responds by sending a ticket encrypted with the symmetric key shared between R and S , encapsulated within a message encrypted with the symmetric key shared between I and S .

```
recv_1(I, S, (I, R, Ni));
send_2(S, I, {Ni, R, Kir, {Kir, I}k(R, S)}k(I, S));
}
```

NSSK Protocol Analysis

	Aliveness	Weak Agreement	Non-injective Agreement	Non-injective Synchronisation	Secret Ni	Secret Nr	Secret T	Secret Kir
I	Ok	Ok	Ok	Ok	Fail	Ok	Fail	Ok
R	Ok	Ok	Ok	Ok	-	Ok	-	Ok