2021-03-19

Julia Key Agreement in SageMath

This worksheet accompanies the paper *Julia: Fast and Secure Key Agreement for IoT Devices* by XXX and XXX, 2021. SageMath 9.2 was used to create this worksheet. The latest version is hosted at XXX.

The example code uses the well-known Curve25519. Any other crypto group could have been chosen. This code should be seen as example code to illustrate the Julia Key Agreement (JKA) as it is described in the paper. Details, such as how to serialize a group element into bytes, is not of importance here. A full secure channel specification and implementation would need to consider many details that are out of scope here.

Init

Run the code blocks below to initialize the field (field), the elliptic curve (ec), the generator (G), and some functions.

In [1]:
```
field = GF(2^255-19)
ec = EllipticCurve(field, [0,486662,0,1,0])
G = ec([9, 14781619447589544791020593568409986887264606134616475288964881837755586237401])
show(ec)
```

$$y^2 = x^3 + 486662x^2 + x$$

In [2]:
```
# Generates and returns a secret key.
def secret(): return 2^254 + 8*randint(1, 2^251-1)
```

In [3]:
```
# Returns a random byte array of length 32 (256 bits).
def rand():
    array = bytearray(32)
    for i in range(len(array)):
        array[i] = randint(0, 255)
    return array
```

In [4]:
```
import hashlib

# Ordinary hash function, returns byte array
def hash1(a):
    hasher = hashlib.sha256()
    hasher.update(a)
    return hasher.digest();

# Hashes and converts to 256-bit integer
def hash2(a):
    digest = hash1(a)
    return int.from_bytes(digest, byteorder='big', signed=False)

# Converts a string, Integer, or group element to a byte array.
def to_bytes(A):
    if isinstance(A, Integer):
        return str(A).encode("US-ASCII")
    elif isinstance(A, str):
        return A.encode("US-ASCII")
    else:
        return str(A[0]).encode("US-ASCII") + str(A[1]).encode("US_ASCI
I") + str(A[2]).encode("US_ASCII")
```

Notes

The following sections illustrate a number of key agreement protocols. Many details are not included, some of which are essential for a real implementation of a full secure channel handshake.

Note, the following:

- Two parties, P1 and P2, communicate over a realible, but insecure communication channel.
- The attacker, Mallory has full power to modify and resend messages. She has access to all sessions ever executed and all sessions currently executing.
- The communication can be assumed to be half-duplex. Only one of the parties send data at a time.
- P1 initiates the communication and P2 responds to the initial message from P1.
- The computation of one or multiple keys for the symmetric cryptography that follows in full secure channel protocol is not included.
- To be concrete, we can assume that a shared symmetric key is computed from a hash of all session data and the result of one or multiple scalar multiplications that can be computed by both parties.
- The goal of the handshake is to achieve one or more secrets that are shared between P1 and P2, but are not available to Mallory. For more information on the security properties of JKA, see the paper.

Key agreement with one scalar multiplication

Each party computes only one scalar multiplication. A shared secret (D) is attained, but forward secrecy and compromised-key impersonation is not support.

In [5]:
```
# ==== Static keys ====

# P1:
s1 = secret()
S1 = s1*G

# P2:
s2 = secret()
S2 = s2*G
```

In [6]:
```
# ==== Pre-handshake ====

# P1:
c1 = rand()

# P2:
c2 = rand()
```

In [7]:
```
# ==== Handshake ====

# P1:
# -> Send c1

# P2:
# <- Send c2

# P1:
D1 = s1*S2

# P2:
D2 = s2*S1
```

In [8]:
```
D1 == D2
```

Out[8]:
```
True
```

Key agreement with three scalar multiplications

This key agreement is conceptually equivalent to protocol "XX" of the Noise Protocol Framework noiseprotocol.org (https://noiseprotocol.org/).

In [9]:
```
# ==== Static keys ====

# P1:
s1 = secret()
S1 = s1*G

# P2:
s2 = secret()
S2 = s2*G
```

In [10]:
```
# ==== Pre-handshake ====

# P1:
e1 = secret()
E1 = e1*G

# P2:
e2 = secret()
E2 = e2*G
```

In [11]:
```
# P1:
# -> Send E1

# P2:
D21 = E1*e2
D22 = E1*s2
# <- Send E2, S2

# P1:
D11 = e1*E2
D12 = e1*S2
D13 = s1*E2
# -> Send S1, app1

# P2:
D23 = S1*e2
```

In [12]:
```
D11 == D21, D12 == D22, D13 == D23
```

Out[12]:
```
(True, True, True)
```

A common secret for encrypting application data can computed based on the session hash (hash of all data transferred) and the Dxx values. The application data is encrypted using all three scalar products (D11, D12, D13) while D11, D12 are used to encrypt S1 and S2.

Julia Key Agreement

Between P1 and P2. The baseline version. t is computed jointly by P1 and P2.

In [13]:
```
# ==== Static keys ====

# P1:
s1 = secret()
S1 = s1*G

# P2:
s2 = secret()
S2 = s2*G
```

In [14]:
```
# ==== Pre-handshake ====

# P1:
e1 = secret()
E1 = e1*G
t1 = hash2(to_bytes("t1") + to_bytes(E1))
h1 = hash1(to_bytes("h1") + to_bytes(E1))

# P2:
e2 = secret()
E2 = e2*G
t2 = hash2(to_bytes("t2") + to_bytes(E2))
```

In [15]:
```
# ==== Handshake ====

# P1:
# -> Send h1

# P2:
# <- Send E2

# P1:
t2 = hash2(to_bytes("t1") + to_bytes(E2))
t = t1+t2
D1 = (t*s1+e1)*(t*S2+E2)
# -> Send E1, app1

# P2:
# Verify h1 = hash(to_bytes("h1") + to_bytes(E1))
t = t1+t2
D2 = (t*S1+E1)*(t*s2+e2)
```

In [16]:
```
D1 == D2
```

Out[16]:
```
True
```

Julia Key Agreement, one scalar multiplication

t1 and t2 are computed independently by the parties. P1 can pre-compute t1, P2 can pre-compute t2.

In [19]:
```
# ==== Pre-handshake ====

# P1:
e1 = secret()
E1 = e1*G
t1 = hash2(to_bytes("t1") + to_bytes(E1))
h1 = hash1(to_bytes("h1") + to_bytes(E1))

# P2:
e2 = secret()
E2 = e2*G
t2 = hash2(to_bytes("t2") + to_bytes(E2))
```

In [20]:
```
# ==== Handshake ====

# P1:
# -> Send h1

# P2:
# <- Send E2

# P1:
t2 = hash2(to_bytes("t1") + to_bytes(E2))
D1 = (t2*s1+e1)*(t1*S2+E2)
# -> Send E1, app1

# P2:
# Verify h1 = hash(to_bytes("h1") + to_bytes(E1))
t = t1+t2
D2 = (t2*S1+E1)*(t1*s2+e2)
```

In [21]:
```
D1 == D2
```

Out[21]:
```
True
```