Overview of the
Schnorr Signature Scheme

Alexandre Connat

January, 2017

1 Introduction

The Schnorr Signature Scheme, is a very simple cryptographic signature scheme, based on the hardness of the discrete logarithm problem. It is used notably in the Bictoin Protocol.
In this document we used the multiplicative notation to describe all the operations performed on the group. But if we work in additive groups, like Elliptic Curves, we could easily draw a parallel with the additive notation.

The goal of this scheme is for a sender to be able to sign a message M with a signature S.
The sender wants to prove its identity and the authenticity of the message by sending the message M alongside with the signature S, over a insecure channel. The receiver should be able to verify the signature with quasi-certainty.

2 Protocol

Both sender and receiver must agree on a group to perform all the following operations. In this document, we'll focus on the group $G = Z_p^*$ with p a big prime number.

We assume the sender has generated a key-pair (Private Key, Public Key) = $(x,\, y)$ with $y = g^x \bmod p$. We assume the receiver is able to access the Public Key y of the sender.

2.1 Sender-Side:

Compute :

- $t = g^r$, where $r \in Z_p^*$ random
- $e = \text{Hash}(M \| t)$, where Hash() is any hash function (Sha1, Sha256, ...) and $\|$ is the 'append' sign
- $s = r - xe$

$\Rightarrow$ The signature is $S = (s, e)$

2.2 Receiver-Side:

Compute :

- $l = g^s y^e$, with s and e from S, and y the Public Key of the sender
- $e_v = \text{Hash}(M\|l)$

$\Rightarrow$ The signature is considered as : $\begin{cases} \text{VALID} & \text{, if } e_v = e \\ \text{INVALID} & \text{, if } e_v \neq e \end{cases}$

3 Proof - Correctness

If we consider a valid key-pair (x, y), with $y = g^x$, we have :

$$\begin{aligned} l &= g^s \cdot y^e \\ &= g^{r-xe} \cdot (g^x)^e \\ &= g^r \cdot g^{-xe} \cdot g^{xe} \\ &= g^r \\ &= t \end{aligned}$$

Hence, we have $\text{Hash}(M\|l) \equiv \text{Hash}(M\|t)$
Hence, we have $e_v \equiv e$

4 Error Probability

- if $e_v \neq e$, signature is INVALID with probability 1.
- if $e_v = e$, signature is VALID with probability $1 - \epsilon$. ($\epsilon \approx 0$ for large p)

5 Proof - Error Probability

Assume the key-pair is invalid. Let's define a random $x' \neq x$, s.t: $y \neq g^{x'}$

We will prove that there are two possibilities to forge a signature :
Either find the exact $x' = x$, which is the real Private Key (with probability $\frac{1}{|G|} = \frac{1}{p-1}$) or, by "chance" $e = \text{Hash}(M\|t)$ is a multiple of p, and its reduction modulo p is 0 (with probability $\frac{1}{p}$).

We receive a forged signature : $S = (s', e) = (r - x'e, e)$
We compute : $l = g^{s'} \cdot y^e = g^r \cdot g^{-x'e} \cdot g^{xe} = g^r \cdot g^{e(x-x')}$

$$\begin{aligned} \Rightarrow Pr(l = g^r) = Pr(g^{e(x-x')} = 1) &= Pr(e = 0) + Pr(x = x') \\ &= \frac{1}{p} + \frac{1}{p-1} \approx \frac{2}{p} \underset{p \to +\infty}{\longrightarrow} 0 \end{aligned}$$

6 Implemented Protocol

To write the actual code, we worked with an Elliptic Curve E, instead of $\mathbb{Z}_p$ and the protocol we used to sign and verify messages slightly differs from what we explained in Part 2.

We worked with the curve $Ed25199$, defined over the prime field $\mathbb{F}_q$, of prime order $q = 2^{255} - 19$ and a base point $G = (x, -4/5)$
with $\ell = 2^{252} + 27742317777372353535851937790883648493$ the prime order of the base point.

We used Sha256 as our Hash Function, which outputs the result as a 256-bit string, which is further reduced modulo q, to be used as a scalar value in the group.

The key-pair is (Private Key, Public Key) = (x, Y), with $Y = x \cdot G$, the point obtained by multiplying the base point G by the scalar x.

6.1 Sender-Side

Compute :

- $R = k \cdot G$, $k \in \mathbb{Z}_\ell^*$ random
- $e = \text{Hash}(M \| R)$, M the message, and "Hash" the Sha256 function
- $s = k + xe$, x the private key of the sender

$\Rightarrow$ The signature is $S = (R, s)$

6.2 Receiver-Side

Compute :

- $e = \text{Hash}(M \| R)$, M the message, and R from the signature S
- $sg_v = R + e \cdot Y$, Y the public key of the sender
- $sg = s \cdot G$, s from the signature S

$\Rightarrow$ The signature is considered as : $\begin{cases} \textsc{Valid} \text{ , if } sg_v = sg \\ \textsc{Invalid} \text{ , if } sg_v \neq sg \end{cases}$

7 Implementation

We based our implementation on the Crypto library of the DEDIS laboratory (www.github.com/dedis/crypto)

We used the cryptographic suite `ed25519.NewAES128SHA256Ed25519()`

Keys are generated using the method `config.NewKeyPair()`
`keypair.Secret` is of type `abstract.Scalar`
`keypair.Public` is of type `abstract.Point`

We provide 2 public methods to the user programmer :

- `SignMessage(m, x)` which takes a message `m` (of type `string`), and a private key `x` (of type `abstract.Scalar`) as input, and outputs a signature `S` (of type `Signature`).
- `VerifySignature(m, S, Y)` which takes a message `m` (of type `string`), a signature `S` (of type `Signature`), and a public key `Y` (of type `abstract.Point`) as input, and outputs a boolean value, representing whether the given signature is a VALID signature (`true`) or an INVALID signature (`false`) for the given message, with this public key.
  (Alternatively, you can use the function `Verify(m, Y)` directly on the Signature S itself)

The signature itself is held in a custom type struct :

```
type Signature struct {
    R abstract.Point
    s abstract.Scalar
}
```

8 Testing

We tested the code with :
- a valid key-pair, a valid message, a valid signature
- a valid message, a valid signature, an invalid key-pair
- a valid valid key-pair, a valid signature, an invalid message to verify

We also tested if the code raised exceptions when those events occurred :
- empty message to sign or verify
- empty or partially empty signature (missing R or s)
- private key equals to zero
- public key equals to the neutral element

And finally we tested the String representation of the Signature struct, as well as its Binary Marshaling and Unmarshaling functions.