Borromean Ring Signature Algorithm and Implementation

No Author Given

No Institute Given

1 Borromean Ring Signature Algorithm

1.1 Signing

Suppose a signer has a collection of verification keys $P_{i,j}$ for $0 \le i \le n-1, 0 \le j \le m_i - 1$, and wants to create a signature of knowledge of the n keys $\{P_{i,j^*}\}_{i=0}^{n-1}$, where j_i^*s are some fixed and unknown (to a verifier) indices. Denote the secret key to P_{i,j_i^*} by x_i. The signing process is as follows:

1. Compute M as the hash of the message to be signed and the set of verification keys.
2. For each $0 \le i \le n-1$:
   (a) Choose a scalar k_i uniformly at random.
   (b) Set $e_{i,j_i^*+1} = H(M||k_iG||i||j_i^*)$.
   (c) For j such that $j_i^* \le j < m_i - 1$ choose $s_{i,j}$ at random and compute

$$e_{i,j+1} = H(M||s_{i,j}G + e_{i,j}P_{i,j}||i||j)$$

3. Choose s_{i,m_i-1} for each i at random and set

$$e_0 = H(s_{0,m_0-1}G + e_{0,m_0-1}P_{0,m_0-1}||...||s_{n-1,m_{n-1}-1}G + e_{n-1,m_{n-1}-1}P_{n-1,m_{n-1}-1})$$

4. For each $0 \le i \le n-1$:
   (a) For j such that $0 \le j < j_i^*$ choose $s_{i,j}$ at random and compute

$$e_{i,j+1} = H(M||s_{i,j}G + e_{i,j}P_{i,j}||i||j)$$

   where $e_{i,0}$ means e_0.
   (b) Set $s_{i,j_i^*} = k_i - x_ie_{i,j_i^*}$.

The resulting signature on m consistes of

$$\sigma = \{e_0, s_{i,j} : 0 \le i \le n-1, 0 \le j \le m_i - 1\}$$

1.2 Verification

The verifier verifies the signature σ based on the message m and a collection $\{P_{i,j}\}$ as follows:

1. Compute M as the hash of the message and the set of verification keys.
2. For each $0 \le i \le n-1$, for each $0 \le j \le m_j - 1$, compute $R_{i,j} = s_{i,j}G + e_{i,j}P_{i,j}$ and $e_{i,j+1} = H(M||R_{i,j}||i||j)$. $e_{i,0} = e_0$.
3. Compute $e_0' = H(R_{0,m_0-1}||...||R_{n-1,m_{n-1}-1})$ and return 1 iff $e_0' = e_0$

1.3 Explanation on the Algorithm

Let's formulate the ring based on the signing algorithm. For i_{th} ring, it starts from node j_i^* and the signer calculates the next hash as $e_{i,j_i^*+1} = H(M||k_iG||i||j_i^*)$. Then the signer goes on calculating e_{i,j_i^*+2}, ..., e_{i,m_i-1}, until he gets to the last node $m_i - 1$ of ring i. The following equation is used in the next hash calculation:

$$e_{i,j+1} = H(M||s_{i,j}G + e_{i,j}P_{i,j}||i||j) \quad (1)$$

The next hash of last node $m_i - 1$ of ring i is defined as e_0, which links the last and first node thus forming a ring:

$$e_0 = H(s_{0,m_0-1}G + e_{0,m_0-1}P_{0,m_0-1}||...||s_{n-1,m_{n-1}-1}G + e_{n-1,m_{n-1}-1}P_{n-1,m_{n-1}-1}) \tag{2}$$

This loops back to node 0 of ring i and the signer goes on calculating $e_{i,1}$,..., e_{i,j_i^*} until it gets back to node j_i^*, using the same Eq.1. Following this equation, we should have the next hash of node j_i^* of ring i to be $e_{i,j_i^*+1} = H(M||s_{i,j_i^*}G+e_{i,j_i^*}P_{i,j_i^*}||i||j_i^*)$. But in the first calculation, we already have $e_{i,j_i^*+1} = H(M||k_iG||i||j_i^*)$. To make sure both of the next hash equations are satisfied, the signer makes use of the knowledge of x_i and sets s_{i,j_i^*} as: $s_{i,j_i^*} = k_i - x_ie_{i,j_i^*}$. By multiplying both sides of this equation with G, we have:

$$s_{i,j_i^*}G = k_iG - x_ie_{i,j_i^*}G \tag{3}$$

$$k_iG = s_{i,j_i^*}G + e_{i,j_i^*}P_{i,j_i^*} \tag{4}$$

It is easy to see that both of the next hash equations are satisfied in this way. In calculating the ring signature, the $s_{i,j}$ of nodes for which the signer does not know the private key is randomly picked, while the s_{i,j_i^*} of node for which the signer knows the private key is calculated using the knowledge of x_i. In this way, any verifier, starting from e_0, can calculate the next hash from node 0, ..., until node $m_i - 1$ of ring i, using Eq.1. The verifier finally computes the e'_0 using Eq.2. If $e'_0 = e_0$, meaning all the n rings can be connected, the signer must know at least one private key of each ring i.

2 Implementation

We used the GO's btcec library (https://godoc.org/github.com/btcsuite/btcd/btcec) which conntains the parameters and methods to compute with the elliptic curve secp256k1. We use Go's built-in SHA-256 implementation for our cryptographic hash function.