Threshold CGGMP

Bolt Labs

1 Introduction

In the original threshold signing protocol of CGGMP [2], the authors briefly mention how to extend their the n out of n scheme to threshold access structures, where only t parties are needed to sign messages. At a high level, the changes to achieve this are conceptually simple. For key generation, we merely swap in a t out of n key generation protocol like the one in [3]. For signing, a non-interactive share conversion protocol is added at the beginning of presign, to convert the t out of n key shares into t out of t shares. This is enough to allow us to run the original presign and sign protocols of CGGMP with no further modification. Because it would be inefficient to use [3] "out of the box" we tweak the protocol similarly to [1] and present the full overview here. Our only meaningful change is always requiring secure broadcast (this is just a optional reliability check in that work). For key refresh, we use some ideas from the updated version of [2] and the key generation protocol described here.

2 Preliminaries

Before we go through the protocols, we describe the primitives we will need and any changes in notation from that used in [2].

As our only piece of potentially new notation, we define $[n] = \{1, \dots, n\}$.

Feldman's VSS relies on the hardness of finding the discrete log in some group G. Let $|G| = q$, g be a generator of G, and suppose we want to share some secret $x \in \mathbb{F}_q$ with threshold t where t parties should be able to reconstruct x. The share algorithm is as follows:

1. First sample coefficients $a_1 ... a_{t-1} \leftarrow \mathbb{F}_q$ set $a_0 = x$. Let $a(z) = \sum_{i=0}^{t-1} a_i \cdot z^i$ and set $sh_i = a(i)$ for $i \in [n]$
2. Calculate $A_i \leftarrow g^{a_i}$ for $i \in \{0, \dots, t-1\}$
3. Output $(\{A_j\}_{j \in \{0,\dots,t-1\}}, sh_i)$ to P_i for $i \in [n]$

Notice that although each share contains $A_0 = g^x$, if the discrete log problem is hard for G, x cannot be recoverd from A_0. Party P_i checks its received share sh_i as:

$$g^{sh_i} \stackrel{?}{=} \prod_{j=0}^{t-1} A_j^{i^j}$$

We also make use of a simple hash-based commitment scheme. We represent this simply by using a hash function $\mathcal{H}$. To commit to x, a party first samples $u \leftarrow \{0,1\}^\kappa$ and computes $y \leftarrow \mathcal{H}(x\|u)$. To reveal the commitment, we reveal x' and u' and the other party checks that $\mathcal{H}(x'\|u') \stackrel{?}{=} y$. This scheme is secure in the ROM.

Finally, we make use of the Schnorr protocol, which allows a party to prove they know the private exponent of some group element with respect to a fixed generator. We use a notation very similar to the one described in the paper, which provides a ZK-module $\mathcal{M}$ for Σ protocols Π (which includes Schnorr). At a high level, this model is a slight modification of a non-interactive version of the Σ protocol, where the first round message of the protocol can be produced in advance on its own.

The interface is described in more detail below.

$\mathcal{M}(\texttt{com}, \Pi) \to Z, \mathsf{st}$ indicates running the algorithm that produces the first round message from the prover to the verifier. The output is the message Z and private state for the prover st.

$\mathcal{M}(\texttt{prove}, \Pi, \mathsf{aux}, x, Z; w, \mathsf{st}) \to \pi$ indicates running the algorithm to produce the proof π where the verifier message is created by hashing together the statement x, auxiliary information aux, and the first round message Z. This algorithm also takes as input a witness w, such that $(x, w) \in \mathsf{R}$ and the private state of the prover st.

$\mathcal{M}(\texttt{vrfy}, \Pi, \mathsf{aux}, x, \pi) \to \{0,1\}$ indicates the algorithm run by the verifier to check if the proof π provided by the prover really shows that the statement $x \in L$, bound to auxiliary information aux. The output is a boolean value.

3 Description of Modifications

3.1 Key Generation

We give the description of the changed distributed key generation protocol in Figure 1. In words, at the start of the protocol, parties first randomly choose a value s_i. That value is then t of n secret shared using a *verifiable secret sharing* or VSS. Parties also generate randomness for the first round of the Schnorr, along with a share of a random value that will be used as context for that proof. They then commit to the public values produced at this phase. After receiving commitments from all other parties, everyone decommits and also sends P_j their share of the secret value they generated in the first round. Parties then check that all shares they received are consistent with the public VSS parameters. If they are, they construct their final share of the key and the public parts of every other's parties key. This step is likely the most computationally expensive and should be optimized. Finally, each party computes Schnorr over their output key share and parties check these proofs. If the checks pass, the protocol is considered to be successful.

The protocol Π_{keygen}

- On input $(\mathtt{keygen}, sid, i)$ from P_i, uniformly choose $s_i \leftarrow \mathbb{Z}_q$. Compute

$$\{A_{i,z}\}_{z\in\{0,\dots,t-1\}}, \{sh_{i,j}\}_{j\in[n]} \leftarrow \mathsf{FeldmanVSS}(n,t,s_i)$$
$$Y_i, y_i \leftarrow \mathcal{M}(\mathtt{com}, \Pi_{\mathsf{sch}})$$

  Let $A_i = \{A_{i,z}\}_{z\in\{0,\dots,t-1\}}$. Sample $\mathsf{rid}_i, u_i \leftarrow \{0,1\}^\kappa$ and produce $C_i \leftarrow \mathcal{H}(sid, i, \mathsf{rid}_i, A_i, Y_i, u_i)$. C_i is *broadcast* to all parties.

- Upon receiving C_j from P_j $\forall j \in [n]$, send to all $\mathsf{rid}_i, A_i, Y_i, u_i$. To P_j send $sh_{i,j}$.

- For each party P_j, check that $C_j \overset{?}{=} H(sid, j, \mathsf{rid}_j, A_j, Y_j, u_j)$. For each value $sh_{j,i}$ received from each P_j check that $sh_{j,i}$ is consistent with A_j using Feldman's VSS. If the check passes, compute

$$\mathsf{rid} \leftarrow \bigoplus_{j=1}^{n} \mathsf{rid}_j$$
$$x_i \leftarrow \sum_j sh_{j,i}, X_i \leftarrow g^{x_i}$$
$$\pi_i \leftarrow \mathcal{M}(\mathtt{prove}, (sid, i, \mathsf{rid}), X_i, Y_i; x_i, y_i)$$
$$X_j \leftarrow g^{sh_{i,j}} \cdot \prod_{k\in[n]\setminus\{i\}} \Big(\prod_{z=0}^{t-1} A_{k,z}^{j^z}\Big), \forall j \in [n] \setminus \{i\}$$

  Send π_i to all parties.

- Receive π_j from each P_j. Check that

  1. $\forall j, \mathcal{M}(\mathtt{vrfy}, \Pi_{\mathsf{sch}}, (ssid, j), X_j, \pi_j) == 1$ and $\pi_j = (Z_j, \dots)$ where $Z_j = Y_j$

  If all the checks succeed, store $\mathsf{rid}, X = (X_1 \dots X_n), x_i$.

Figure 1: Key Generation

3.2 Presign

The last section describes how a key is produced so that each party holds a t out of n sharing of the private key. Suppose that t parties have decided to participate in signing. Supporting this is as simple as converting the t out of n shares to t out of t additive shares. If we can do this conversion, note that we can reuse the old n out of n presign and signing protocols with no further modification. Below we describe how to do this conversion at the beginning of presign.

Let $T \subset [n]$ be a subset of t parties that would like to participate in presign and sign. Each party holds their key share x_i. Our goal is to construct new shares x'_i s.t. $\sum_{i \in T} x_i = x$ where x is the ECDSA private key. Recall that we can always recover *any* point on a degree $t-1$ polynomial, using t points. In particular, we know that the key shares $x_i = f(i)$ for some degree $t-1$ polynomial f with $x = f(0)$. The formula to recover the polynomial $f(a)$ given points $\{(i, x_i)\}_{i \in T}$ is

$$f(a) = \sum_{i \in T} x_i \cdot L_i(a)$$

where $L_i(a) = \prod_{j \in T \setminus \{i\}} \frac{(a - x_j)}{(x_i - x_j)}$. Therefore $x = \sum_{i \in T} x_i \cdot L_i(0)$ where $L_i(0)$ is some (different) constant for each i. P_j's new share is then just $x'_i = x_i \cdot L_i(0)$ and can be obtained non-interactively, without communicating with any other party. This also applies to the new public keys that must be used. In particular, each party P_i holds $X = (X_1, \ldots, X_n)$. For $j \in T \setminus \{i\}$, P_i must calculate $X'_j = X_j^{L_j(0)}$. Note that $L_j(0)$ is public. In the rest of presign, P_i uses x'_i, X'_i as its public private key shares for the ECDSA key and X'_j as the public keys for the other parties participating in the protocol. Besides this, everything is equivalent to the n out of n setting.

4 Proof Sketch of Security

Because substantial changes exist in the key generation protocol, we must at least show how the changes made here do not lead to an insecure protocol. In particular, we must change the "standalone simulator" for key generation. This simulator is supposed to take as input an sid, C, L, X. The goal is to correctly simulate the responses from honest parties to the adversary, extract the shares of the private key for the corrupt parties, and to ensure the output public ECDSA key is X. This is done by the having the simulator choose a special honest party P_b and having them choose a public key share X_b s.t. $X_b = X \cdot (\prod_{j \neq b} X_j)^{-1}$. Below we give a more formal description of the simulator, in line with what was given in the original work.

The simulator $\mathcal{S}_1(sid, C, L, X)$ takes as input a session identifier sid, a subset of corrupt parties $C \subset P$, a list L of simulate oracle queries (with answers), and the public key X that ultimately should be the output of the protocol. In what follows, let $H = P \setminus C$ be the set of honest parties that are being simulated by $\mathcal{S}_1$.

Round 1 Set $\mathsf{ext} = 0$. Sample commitments $\{V_i\}_{i \in H}$ from the correct distribution. Send $(ssid, i, V_i)$ to $\mathcal{Z}$, $\forall i \in H$

Round 2 When receiving (sid, j, C_j) from $j \in C$,

- If $\mathsf{ext} = 0$, send the correctly sampled values $\{(sid, i, \mathsf{rid}_i, A_i, Y_i, u_i)\}_{i \in H}$ and $\{sh_{i,j}\}_{i \in H, j \in C}$ to $\mathcal{Z}$

(†) Otherwise, choose $P_b \in H$. Let $A_{j,0}$ be the values committed by all parties but P_b in the first round. Set $A^*_{b,0} = X \cdot \prod_{j \neq b} A^{-1}_{j,0}$. For P_b, sample $A^*_{b,1} \dots A^*_{b,t-1}, \{sh^*_{b,j}\}_{j \in C}$ s.t. the Feldman VSS tests will pass for the at most $t-1$ shares that must be given to corrupt parties. This is basically simulating Feldman's VSS for the sharing of the log of $A^*_{b,0}$.

- P_b should send to $\mathcal{Z}$ $(sid, b, \mathsf{rid}_b, A^* = \{A^*_{b,j}\}_{j \in [t-1]}, Y_b, u_b)$ where the opening u_b is correctly programmed into the random oracle. Also send to $\mathcal{Z}$ the shares $sh^*_{b,j}$ where $j \in C$.
- For other honest parties, send correctly sampled values $\{(sid, i, \mathsf{rid}_i, A_i, Y_i, u_i)\}_{i \in H \setminus \{P_b\}}$ and $\{sh_{i,j}\}_{i \in H, j \in C}$ to $\mathcal{Z}$.

Add relevant tuples to L.

Round 3 First, calculate

$$X_j \leftarrow \Big(\prod_{k \in [n] \setminus \{b\}} \prod_{z=0}^{t-1} A^{j^z}_{k,z} \Big) \cdot \prod_{z=0}^{t-1} (A^*_{b,z})^{j^z}, \forall j \in [n]$$

$$\mathsf{rid} \leftarrow \bigoplus_{j=1}^{n} \mathsf{rid}_j$$

1. If $\mathsf{ext} = 0$, send correctly sampled values for all $j \in H$
2. Else, invoke the ZK-simulator $\mathcal{S}^{\mathsf{sch}}(X_j, \dots)$ to get $\pi_j = (Y_j, \dots)$ for all $j \in H$. Give to $\mathcal{Z}$ the values $\{\pi_j\}_{j \in H}$.

Add relevant tuples to L.

Output If $\mathsf{ext} = 0$, set $\mathsf{ext} = 1$ and go to (†) in Round 2. Delete the pairs added to L. Otherwise, extract $\{x_j\}_{j \in C}$, output $b, L, rid, \{x_k\}_{k \neq b}$.

Notice that the stand-alone simulator presented here has the same input and output behavior as the previous one and that all values produced by the simulator appear appropriately distributed to the adversary. Note the environment does not get to see shares sent between honest parties and that is because we assume secure channels exist for these values to be sent across to the correct counter-party.

5 Key Refresh

In this section, we briefly discuss how key refresh can be done by adapting the key generation protocol presented in Figure 1. Suppose each party holds the share (i, x_i). We then non-interactively transform the $\{x_i\}_{i \in [n]}$ values into an n out of n sharing, using the lagrangian coefficients as specified in Section 3.2 i.e. the new share of a party would be $x'_i = x_i \cdot L_i(0)$ where $L_i(a) = \prod_{j \in [n] \setminus \{i\}} \frac{a - x_j}{x_i - x_j}$. Now instead of using a randomly chosen s_i value in the key generation protocol, parties use x'_i. In addition, a check should be done to ensure that every other party's Feldman's VSS was done using the secret exponent corresponding to their share of the old public key i.e. $\forall j, A_{j,0} = X_j^{L_j(0)}$.

References

[1] Cggmp specification. `https://dfns.github.io/cggmp21/cggmp21-spec.pdf`.

[2] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. Uc non-interactive, proactive, threshold ecdsa with identifiable aborts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1769–1787, 2020.

[3] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ecdsa with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1179–1194, 2018.