TLS

TheLastStand

Decentralized fifinancial networks of today remain isolated in silos that cannot trustlessly

Contract

TheLastStand

Contract

0xDE38d51bAA18313103D177c0ae40ba1766A3702

Abstract

Decentralized fifinancial networks of today remain isolated in silos that cannot trustlessly communicate with each other and meaningfully exchange value. This leads custodial services to offffer potentially dangerous products that makes the overall ecosystem handing over control of funds to third parties goes against the ethos that cryptocurrency was built on. **TheLastStand**a viable solution to this pain point utilising **TheLastStand** proofs, Bitcoin script, and Ethereum contracts by means of a unique solution called SPV simulation that enables trustless two-way pegs across Bitcoin, Ethereum, Polkadot, and more.

Contents

Chapter 1

Introduction

Since the early days of Nakamoto consensus, the ideal of trustless agreement across blockchains has captured imaginations and inspired innovation. The intrigue of joint consensus, however, does not diminish the marvel of universal concurrence over a single blockchain.

A 2-way peg (2WP) allows a transfer of an asset from base chain to a secondary blockchain and vice-versa. The “transfer” is in fact an illusion: base layer assets are not transferred, but temporarily locked on the base blockchain while the same amount of equivalent tokens are unlocked in a secondary blockchain. The base layer assets can be unlocked when the equivalent amount of tokens on the second blockchain are locked again (in the secondary blockchain). This is essentially the 2WP promise.

The problem with this promise is that it can only be theoretically realized if the secondary blockchain has settlement fifinality. Therefore, any 2WP system must be resistant to corruption and rely on assumptions about the honesty of the actors involved in the 2WP. The most important assumptions are that the primary blockchain is censorship resistant, and that the majority of miners are assumed to be honest. Another required assumption may be that the majority of third parties that will hold custody of locked assets is also assumed to be honest. If these assumptions do not hold, then base layer assets and their equivalent secondary blockchain tokens could be both unlocked at the same time, thus allowing a malicious double-spend. Any 2WP system must choose an implementation so that the parties being assumed to behave honestly have economic and legal incentives to do so. This involves analysing the cost of an attack by these critical parties and consequences of an attack. The security of a 2WP implementation depends on the incentives to enforce the 2WP promise by the critical parties taking part of the 2WP system.

To achieve a new-level of a trustless 2WP system, Clover created a model called built-in SPV chain simulation technology to enable trustless two-ways pegs between Turing-complete and non-Turing-complete blockchains. Contrary to popular belief, an EVM can verify Bitcoin transactions directly, by enforcing some standards and dissecting Bitcoin transactions and block headers. Clover is building advanced tools and opcodes for simplifying the overall process for third-party developers. This is so that Clover can natively inspect a Bitcoin or Ethereum transaction without storing/checking the entire external blockchain history, allowing trustless two-way pegs for Bitcoin and Ethereum. Bitcoin and Ethereum transactions are both included in a Merkle tree, whose block header contains the root of the Merkle tree for that block’s transactions. Given a header and a transaction, Clover can validate a Merkle path from the root to the leaf that holds the transaction, which is called the Merkle based inclusion proof. This means that Clover only needs the base layer block header, transaction, and its inclusion proof to be stored in the Clover contract.

A user willing to peg-out some Bitcoin or Ether sends funds to a predetermined covenant/contract address that escrows funds for further peg-ins, along with any respective proof data Clover needs for verifification. Clover verififies an outside transaction inside a smart contract. This part is a quick run-through of transaction components, to ensure the caller is not trying to sneak through fake content. Clover then can verify that the transaction is included in a block by checking a Merkle proof of inclusion, then checking each block references the previous one, and then calculate the diffiffifficulty level of that chain.

Clover validators, while securing the network, simultaneously notarise Clover block headers with a secure *n* of *m* threshold signature scheme. Clover uses BIP-340 compatible Schnorr signatures to provide a high level of decentralization for the threshold notarisation, as opposed to today’ s federated cross-chain bridges that are limited by either 15 signatories in Bitcoin scripting or massive gas consumption on Ethereum. A user willing to peg-in Bitcoins or Ethers sends tokenised assets to Clover-deployed peg-in contract whose proof of inclusion along with the notary proof can redeem real assets back on their own chain.

In summary, this document describes the Clover threshold based signatures for the bridge between Ethereum and Clover.

- Chapter 2 provides the requirements and assumptions of Clover.
- Chapter 3 presents the underlying cryptographic primitives including the threshold Schnorr signatures.
- Chapter 7 provides a description of the bridge between Clover and Ethereum

Chapter 2

Requirements and Assumptions

2.1 Requirements

- The communication between the signers are done through gRPC services with mutual SSL/TLS protocol. This protocol will only work with the certificates generated by thesame internal Certificate Authority (CA). Each certificate must contain domain name,unique serial number, and other required attributes.
- For each signing server (i.e., signatory) Serveri, an online backup server must be main┐ tained.
- Servers must be controlled by different governors from different locations.

2.1.1 (t, n)-Threshold Signatures

1. Clover uses (t, n)-threshold signature scheme for notarization purposes.

2.2 Assumptions

- The underlying cryptographic primitives (e.g., Schnorr signatures, SHA256, Blakeb) are assumed to be secure.
- The users are supposed to use secure random number generators.
- The security of the system relies on the threshold number of honest signatories (i.e., atleast the threshold number of signatories are assumed to be secure at all times).
- CLV is the native token of the Clover network and plays a critical role in incentivizing signatories for acting non-maliciously

Chapter 3

Cryptographic Primitives

3.1 Shamir Secret Sharing Scheme with a Dealer

In a (t, n) secret sharing scheme, a dealer distributes a secret s to n players $P1, \dots, Pn$ in such a way that any group of at least t players can reconstruct the secret s, while any group of less than t players do not get any information about s.

3.1.1 Secret sharing

- A dealer chooses $s \in R\, \mathbb{Z}q$ (where $n < q$).
- The dealer chooses a random polynomial $f(.)$ over $\mathbb{Z}q$ of degree at most $t - 1$ satisfying $f(0) = s$.
- Each player Pi receives $si = f(i)$ as his share.
- The dealer computes the public key of the players as $P = s.G$.
- The public and the private key shares: $(pk, (sk1, \dots, skn)) = (P, (s1, \dots, sn))$.

3.1.2 Secret Reconstruction

An arbitrary group P of t participants can reconstruct the polynomial $f(.)$ by Lagrange’s interpolation as follows:

$$f(u) = \sum_{i \in \mathcal{P}} f(i)\omega_i(u) \text{ where } \omega_i(u) = \prod_{j \in \mathcal{P}, j \neq i} \frac{u-j}{i-j} \mod q.$$

Since it holds that $s = f(0)$, the group P can reconstruct the secret as follows:

$$s = f(0) = \sum_{i \in \mathcal{P}} f(i)\omega_i \text{ where } \omega_i = \omega_i(0) = \prod_{j \in \mathcal{P}, j \neq i} \frac{j}{j-i} \mod q.$$

3.2 Verifiable Secret Sharing Schemes

We use elliptic curve notation for the discrete logarithm problem. Suppose q is a large prime and G, H are generators of a subgroup of order q of an elliptic curve E. We assume that E is chosen in such a way that the discrete logarithm problem in the subgroup generated by G is hard, so it is infeasible to compute the integer d such that $G = dH$.

3.2.1 Verifiable Secret Sharing Scheme to Prevent the Dealer From Cheating

A Verifiable Secret Sharing scheme (VSS) prevents the dealer from cheating. In a VSS scheme, each player can verify his share. If the dealer distributes inconsistent shares, he will be detected. Assume the dealer has a secret $s0 \in Zq$ and a random number $s0 \in Zq$, and is committed to the pair $(s, s0)$ through public information $C0 = sG + s0H$. The secret s can be shared among $P1, \cdot \cdot \cdot, Pn$ as follows.
The dealer performs the following steps

1. Choose random polynomials $f(u) = s + f_1u + \ldots + f_{t-1}u^{t-1}$ and $f'(u) = s + f_1'u + \ldots + f_{t-1}'u^{t-1}$ where $s, s', f_k, f_k' \in \mathbb{Z}_q$ for $k \in \{1, \ldots, t-1\}$.
2. Compute $(s_i, s_i') = (f(i), f'(i))$ for $i \in \{1, \ldots, n\}$.
3. Send (s, s_i') secretly to player $\mathcal{P}_i$ for $i \in \{1, \ldots, n\}$.
4. Broadcast the values $C_k = f_kG + f_k'H$ for $k \in \{1, \cdots, t-1\}$.

Each player $\mathcal{P}_i$ performs the following steps

1. Verify

$$s_iG + s_i'H = \sum_{k=0}^{t-1} i^k C_k. \qquad (1)$$

If this is false, broadcast a complaint against the dealer.
2. For each complaint from a player *i*, the dealer defends himself by broadcasting the values $f(i)$, $f0(i)$ that satisfy the above equality in step 1.
Reject the dealer if

- he received at least t complaints in step 1, or
- he answered to a complaint in step 2 with values that violate step 1.

Pedersen proved that any coalition of less than t players cannot get any information about the shared secret, provided that the discrete logarithm problem in E is hard.

3.2.2 Generating a Random Secret without Trusted Dealer

We can also generate a random shared secret in a distributed way which can be achieved by the following protocol [GJKR07].

The *KeyGen* protocol will be represented as

$$(s_1, \cdots, s_n) \xleftarrow{(t,n)} KeyGen((r|Y, a_kG, H_0), k \in \{1, \ldots, t-1\}).$$

The notation here means that:

- *H*0: the set of players that have not been detected to be cheating.
- *sj* is *Pj* ' s share of the secret *r* for each *j* ∈ *H*0.
- The values *akG* are the public commitments of the sharing polynomial *f*() (they can be computed using public information).
- (*r, Y*): *r* is a private key and *Y* is the corresponding public key.

KeyGen():

1. Each player $\mathcal{P}_i$ chooses $r_i, r_i' \in \mathbb{Z}_\ell$ at random and verifiably secret shares (r_i, r_i'), acting as the dealer according to Pedersen's VSS scheme. Let the sharing polynomials be $f_i(u) = \sum_{k=0}^{t-1} a_{ik}u^k$ and $f_i'(u) = \sum_{k=0}^{t-1} a_{ik}'u^k$ where $a_{i0} = r_i, a_{i0}' = r_i'$, and let the public commitments be $C_{ik} = a_{ik} = a_{ik}G + a_{ik}'H$ for $k \in \{0, \ldots, t-1\}$.

2. The distributed secret value r is not explicitly computed by any player, but it equals $r = \sum_{i \in H_0} r_i$. Each player $\mathcal{P}_i$ sets his share of the secret as $s_i = \sum_{j \in H_0} f_j(i) \mod q$, and the value $s_i' = \sum_{i \in H_0} f_j'(i) \mod q$.

3. Extracting $Y = \sum_{j \in H_0} r_jG$: Each player in H_0 exposes $Y_i = s_iG$ via Feldman's scheme:

   - Each player $\mathcal{P}_i$ for $i \in H_0$ broadcasts $A_{ik} = a_{ik}G$ for $k \in \{0, \ldots, t-1\}$.
   - Each player $\mathcal{P}_j$ verifies the values broadcast by the other players in H_0. In particular, every player $\mathcal{P}_i$ for $i \in H_0, \mathcal{P}_j$ checks if

   $$f_i(j)G = \sum_{k=0}^{t-1} j^k A_{ik}. \qquad (2)$$

   If the check fails for an index i, $\mathcal{P}_j$ complains against $\mathcal{P}_i$ by broadcasting the values $f_i(j), f'i(j)$ that satisfy (1) but do not satisfy (2).

- For players $\mathcal{P}_i$ who received at least one valid complaint, i.e., values which satisfy (1) but do not satisfy (2), the other players run the reconstruction phase of Pedersen's VSS scheme to compute $r_i, f_i(.), A_{ik}$ for $k = 0, \ldots, t-1$ in the clear. All players in H_0 set $Y_i = r_iG$.

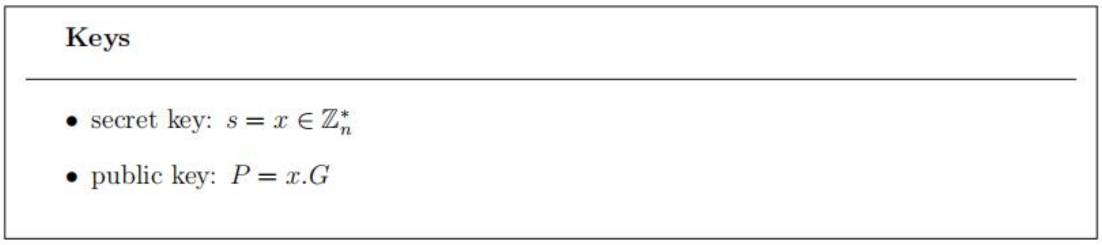
After executing this protocol, the following equations hold:

$Y = rG$
$f(u) = r + a_1u + \ldots + a_{t-1}u^{t-1}$ where $a_k = \sum_{j \in H_a} a_{jk}$ for $k \in \{1, \ldots, t-1\}$
$f(j) = s_j$ for $j \in H_0$.

This scheme has been proven to be robust under the assumption that $t \leq n/2$, i.e., if less than t players are corrupted, the values computed by the honest players satisfy the above equations.

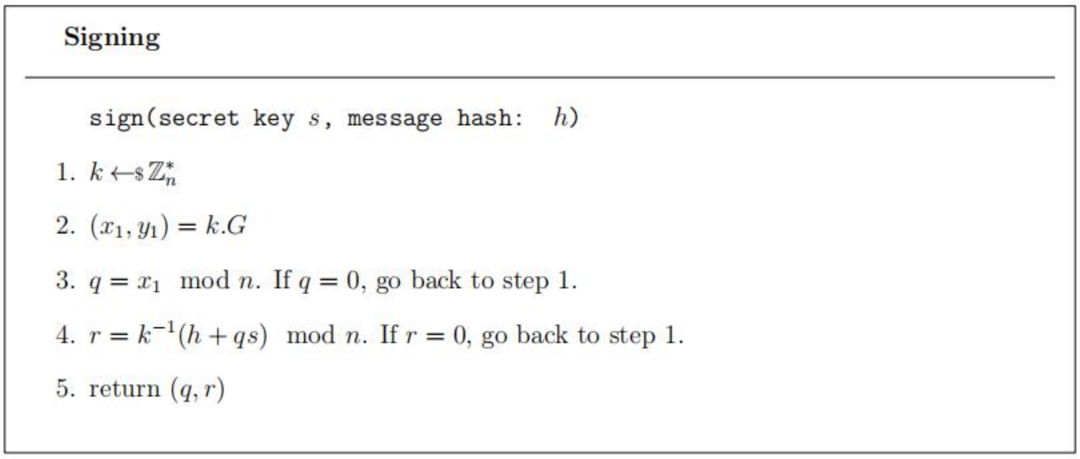
3.3 Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is a standard cryptographic algorithm [Nat13] used by almost all cryptocurrencies to guarantee that funds can only be spent by their rightful owners.



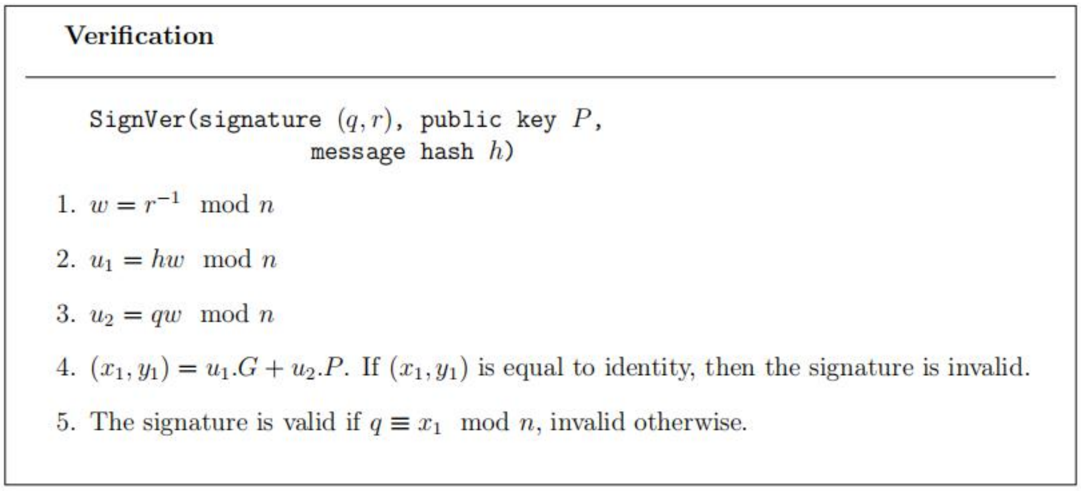
Keys

- secret key: $s = x \in \mathbb{Z}_n^*$
- public key: $P = x.G$



Signing

```
   sign(secret key $s$, message hash: $h$)
1. $k \leftarrow_\$ \mathbb{Z}_n^*$
2. $(x_1, y_1) = k.G$
3. $q = x_1 \mod n$. If $q = 0$, go back to step 1.
4. $r = k^{-1}(h + qs) \mod n$. If $r = 0$, go back to step 1.
5. return $(q, r)$
```



Verification

```
   SignVer(signature $(q, r)$, public key $P$,
                   message hash $h$)
1. $w = r^{-1} \mod n$
2. $u_1 = hw \mod n$
3. $u_2 = qw \mod n$
4. $(x_1, y_1) = u_1.G + u_2.P$. If $(x_1, y_1)$ is equal to identity, then the signature is invalid.
5. The signature is valid if $q \equiv x_1 \mod n$, invalid otherwise.
```

Chapter 4

A (t, n) Threshold Schnorr Signatures

4.1 Schnorr Signatures

Let (x, Y) be a user’ s key pair, let m be a message, let $h()$ be a one-way hash function, and let G be a generator of an elliptic curve group having prime order q. Then a user generates a Schnorr signature on the message m as follows.

1. Select $e \in \mathbb{Z}_q$ at random
2. Compute $V = eG$
3. Compute $\sigma = e + h(m||V)x \mod q$
4. Define the signature on m to be (V, σ)

A verifier accepts a signature (V, σ) on a message m if and only if $\sigma \in \mathbb{Z}_q$ and

$$\sigma G = V + h(m||V)Y.$$

4.2 BIP340

4.2.1 Public Key Generation

Input: The secret key *sk*: a 32-byte array, freshly generated uniformly at random.
The algorithm $PubKey(sk)$ is defifined as:

- Let $d' = int(sk)$.
- Fail if $d' = 0$ or $d' \geq n$.
- Return $bytes(d' \cdot G)$.

Remark. *Note that we use a very diffffferent public key format (32 bytes) than the ones used by existing systems (which typically use elliptic curve points as public keys, or 33-byte or 65-byte encodings of them). A side effffect is that* $PubKey(sk) = PubKey(bytes(n - int(sk))$*, so every public key has two corresponding secret keys.*

4.2.2 Signing

Input:

- The secret key *sk*: a 32-byte array
- The message *m*: a 32-byte array
- Auxiliary random data *a*: a 32-byte array

The algorithm *Sign*(*sk, m*) is defifined as:

1. Let $d0 = int(sk)$
2. Fail if $d0 = 0$ or $d0 \geqslant n$
3. Let $P = d0 \cdot G$
4. Let $d = d0$ if $haseveny(P)$, otherwise let $d = n - d0$.
5. Let t be the byte-wise XOR of $bytes(d)$ and $hashBIP0340/aux(a)$.
6. Let $rand = hashBIP0340/nonce(t||bytes(P)||m)$.
7. Let $k0 = int(rand)$ mod n.
8. Fail if $k0 = 0$.
9. Let $R = k0 \cdot G$.
10. Let $k = k0$ if $has\_even\_y(R)$, otherwise let $k = n - k0$.
11. Let $e = int(hashBIP0340/challenge(bytes(R)||bytes(P)||m))$ mod n.
12. Let $sig = bytes(R)||bytes((k + ed)$ mod $n)$.
13. If $V erify(bytes(P), m, sig)$ returns failure, abort.
14. Return the signature *sig*.

4.2.3 Verifification

Input:

- The public key *pk*: a 32-byte array
- The message *m*: a 32-byte array
- A signature *sig*: a 64-byte array

The algorithm *V erify*(*pk, m, sig*) is defifined as:

1. Let $P = lif\ tx(int(pk))$; fail if that fails.
2. Let $r = int(sig[0 : 32])$; fail if $r \geqslant p$.
3. Let $s = int(sig[32 : 64])$; fail if $s \geqslant n$.
4. Let $e = int(hashBIP0340/challenge(bytes(r)//bytes(P)//m))$ mod n.
5. Let $R = s \cdot G - eP$.
6. Fail if $is\_inf\ inite(R)$.
7. Fail if not $has\_even\_y(R)$.
8. Fail if $x(R)$ $6 = r$.
9. Return success if and only if no failure occurred before reaching this point.

For every valid secret key sk and message m, $V erify(P ubKey(sk), m, Sign(sk, m))$ will succeed.

Remark. *Note that the correctness of verifification relies on the fact that lif tx always returns a point with an even Y coordinate.*

4.3 A (t, n) Threshold Schnorr Signatures

We are now ready to describe (t, n) threshold digital signature scheme for Schnorr signatures [SS01].

Our protocol consists of a key generation protocol, a signature issuing protocol, and a verification protocol. Let $P_1, \cdots, P_n$ be the set of players issuing a signature and let G be a generator of an elliptic curve group of order q.

4.3.1 Key Generation Protocol

All n players have to co-operate to generate a public key, and a secret key share for each P_j. Let the output of the protocol be

$$(\alpha_1, \cdots, \alpha_n) = (x|Y, b_k G, H_0), k \in \{1, \cdots, t-1\}.$$

For each $j \in H0$, αj is the secret key share of Pj and will be used to issue a partial signature for the key pair (x, Y).

4.3.2 Signature Issuing Protocol

Let *m* be a message and let *h*() be a one-way hash function. Suppose that the players with index set *H*1 ⊂ *H*0 wants to issue a signature. They use the following protocol:

1. If $|H_1| < t$, stop. Otherwise, the subset H_1 generates a random shared secret. Let the output be
   $(\beta_1, \cdots, \beta_n) = (e|V, c_k G, H_2), k \in \{1, \cdots, t-1\}$.
2. If $|H_2| < t$, stop. Otherwise, each P_i for $i \in H_2$ reveals
   $$\gamma_i = \beta_i + h(m||V)\alpha_i.$$
3. Each P_i for $i \in H_2$ verifies that $\gamma_j G = V + \sum_{k=1}^{t-1} c_k j^k G + h(m||V)(Y + \sum_{k=1}^{t-1} b_k j^k G)$ for all $j \in H_2$.
4. Let H_3 be the index set of players not detected to be cheating at step 3.
5. If $|H_3| < t$, then stop. Otherwise, each P_i for $i \in H_3$ selects an arbitrary subset $H_4 \subset H_3$ with $|H_4| = t$ and computes σ satisfying $\sigma = e + h(m||V)x$,
   where
   $$\sigma = \sum_{j \in H_4} \gamma_j \omega_j \text{ and } \omega = \prod_{i = \neq j, \ell H_4} \frac{\ell}{\ell - j}.$$
   The signature is (σ, V).

4.3.3 Signature Verification Protocol

The signature can be verified as in Schnorr's original scheme:

$$\sigma G = V + h(m||V)Y \text{ and } \sigma \in Z_q.$$

Chapter 5

Gennaro et al.'s Threshold ECDSA with Trustless Setup

5.1 A Share Conversion Protocol

Assume that we have two parties Alice and Bob holding two secrets $a, b \in \mathbb{Z}_q$ respectively which we can think of as multiplicative shares of a secret $x = ab \mod q$. Alice and Bob would like to compute secret additive shares α, β of x, that is random values such that $\alpha + \beta = x = ab \mod q$ with Alice holding a and Bob holding b.

We assume that Alice is associated with a public key E_A for an additively homomorphic scheme E over an integer N. Let $K > q$ also be a bound which will be specified later. In the following, we will refer to this protocol as an MtA (for Multiplicative to Additive) share conversion protocol. In our protocol we also assume that $B = g^b$ might be public. In this case an extra check for Bob is used to force him to use the correct value b. We refer to this enhanced protocol as MtAwc (as MtA "with check").

1. Alice initiates the protocol by

   - sending $c_A = E_A(a)$ to Bob
   - proving in ZK that $a < K$ via a range proof

2. Bob computes the ciphertext $c_B = b \times_E c_A +_E E_A(\beta') = E_A(ab + \beta')$ where $\beta' \in \mathbb{Z}_N$. Bob sets his share to $\beta = -\beta' \mod q$. He responds to Alice by

   - sending c_B
   - proving in ZK that $b < K$
   - only if $B = g^b$ is public proving in ZK that he knows b, β' such that $B = g^b$ and $c_B b \times_E c_A +_E E_A(\beta')$

3. Alice decrypts c_B to obtain α' and sets $\alpha = \alpha' \mod q$.

5.2 Key Generation Protocol

- **Phase 1.** Each Player P_i selects $u_i \in_R \mathbb{Z}_q$; computes $[KGC_i, KGD_i] = Com(g^{u_i})$ and broadcast KGC_i. Each Player P_i broadcasts E_i the public key for Paillier's cryptosystem.
- **Phase 2.** Each Player P_i broadcasts KGD_i. Let y_i be the value decommitted by P_i. The player P_i performs a (t,n) Feldman-VSS of the value u_i. The public key is set to $y = \prod_i y_i$. Each player adds the private shares received during the n Feldman VSS protocols. The resulting values x_i are a (t,n) Shamir's secret sharing of the secret key $x = \sum_i u_i$. Note that the values $X_i = g^{x_i}$ are public.
- **Phase 3.** Let $N_i = p_i q_i$ be the RSA modulus associated with E_i. Each player P_i proves in ZK that he knows x_i using Schnorr's protocol [Sch21] and that he knows p_i, q_i using any proof of knowledge of integer factorization.

5.3 Signature Generation

Let $S \subset [1 \cdots n]$ be the set of players participating in the signature protocol. We assume that $|S| = t$. For the signing protocol we can share any ephemeral secrets using a (t,t) secret sharing scheme, and do not need to use the general (t,n) structure. We note that using the appropriate Lagrangian coefficients λ_i, S each player in S can locally map its own (t,n) share x_i of x into a (t,t) share of x, $w_i = (\lambda_{i,S})(x_i)$, i.e. $x = \sum_{i \in S} w_i$. Since $X_i = g^{x_i}$ and $\lambda_{i,S}$ are public values, all the players can compute $W_i = g^{w_i} = X_i^{\lambda_{i,S}}$.

1. **Phase 1.** Each Player P_i selects $k_i, \gamma_i \in_R \mathbb{Z}_q$; computes $[C_i, D_i] = Com(g^{\gamma_i})$ and broadcast C_i. Define $k = \sum_{i \in S} k_i$, $\gamma = \sum_{i \in S} \gamma_i$. Note that

$$k\gamma = \sum_{i,j \in S} k_i \gamma_j \mod q$$

$$kx = \sum_{i,j \in S} k_i w_j \mod q$$

2. Every pair of players P_i, P_j engages in two multiplicative-to-additive share conversion subprotocols

   - P_i, P_j run MtA with shares k_i, γ_j respectively. Let α_{ij} [resp. β_{ij}] be the share received by player P_i [resp. P_j] at the end of this protocol, i.e.

$$k_i \gamma_j = \alpha_{ij} + \beta_{ij}.$$

   Player P_i sets $\gamma_i = k_i \gamma_i + \sum_{j \neq i} \alpha_{ij} + \beta_{j \neq i} \psi_{ji}$. Note that the σ_i are a (t,t) additive sharing of $kx = \sum_{i \in S} \sigma_i$.

3. **Phase 3.** Every player P_i broadcasts δ_i and the players reconstruct $\delta = \sum_{i \in S} \delta = k\delta$. The players compute $\delta^{-1} \mod q$.

4. Each Player P_i broadcasts D_i. Let Γ_i be the values decommitted by P_i who proves in ZK that he knows γ_i such that $\Gamma_i = g^{\gamma_i}$ using Schnorr's protocol.

   The players compute

$$R = [\prod_{i \in S} \Gamma_i]^{\delta^{-1}} = g^{(\sum_{i \in S} \gamma_i) k^{-1} \gamma^{-1}} = g^{\gamma k^{-1} \gamma^{-1}} = g^{k-1}$$

   and r = H'(R).

5. **Phase 5.** Each player P_i sets $s_i = mk_i + r\sigma_i$. Note that

$$\sum_{i \in S} s_i = m \sum_{i \in S} k_i + r \sum_{i \in S} \sigma_i = mk + rkx = k(m + xr) = s$$

i.e. the s_i are a (t, t) sharing of s.

- **(5A)** Player P_i chooses $\ell_i, \rho_i \in \mathbb{Z}_q$ computes $V_i = R^{s_i} g^{\ell_i}, A_i = g^{\rho_i}$, $[\hat{C}_i, \hat{D}_i] = Com(V_i, A_i)$ and broadcasts $\hat{C}_i$. Let $\ell = \sum_i \ell_i$ and $\rho = \sum_i \rho_i$.
- **(5B)** Player P_i broadcasts $\hat{D}_i$ and proves in ZK that he knows s_i, ℓ_i, ρ_i such that $V_i = R^{s_i} g^{\ell_i}$ and $A_i^{\rho_i}$. If a ZK proof fails, the protocol aborts. Let $V = g^{-m} y^{-r} \sum_{i \in S} V_i$ (this should be $V = g^{\ell}$) and $A = \prod_{i \in S} A_i$.
- **(5C)** Player P_i computes $U_i = V^{\rho_i}$ and $T_i = A^{\ell_i}$. It commits $[\hat{C}_i, \hat{D}_i] = Com(U_i, T_i)$ and broadcasts $\hat{D}_i$.
- **(5D)** Player P_i broadcasts $\hat{D}_i$ to decommit to U_i, T_i if $\prod_{i \in S}[Ti] \neq \prod_{i \in S} s_i$ the protocol aborts.
- **(5E)** Otherwise player P_i broadcasts s_i. The players compute $s = \sum_{i \in S} s_i$. If (r, s) is not a valid signature the players abort, otherwise they accept and end the protocol.

Chapter 6

Secure Communication Between Signers Through Schnorr-Signed ElGamal Encryption

In this section, we propose an end-2-end secure communication protocol between signers. This protocol is necessary for signers to be able to apply threshold key generation privately.

6.1 Schnorr-Signed ElGamal Encryption

The following algorithm is called Schnorr-Signed ElGamal Encryption [SJ00] where Alice is the sender and Bob is the receiver.

6.1.1 System Setup:

- Generate a group $\mathbb{G}$ of prime order q.
- Choose an arbitrary integer $P \in \mathbb{G}$.
- Alice has long-term public and private key pair (aG, a).
- Bob has long-term public and private key pair (bG, b).
- Let $Cert_A = Sign_{\mathsf{CA}}(\cdots, A, aG, \cdots)$ be a certificate of Alice and $Cert_B = Sign_{\mathsf{CA}}(\cdots, B, bG, \cdots)$ be a certificate of Bob where CA is the Certificate Authority. We assume that Alice holds Bob's certificate and Bob holds Alice's certificate.
- Choose a hash function $H_1 : \mathbb{G} \to \{0,1\}^k$ (where |k|=256).
- Choose a hash function $H_2 : \mathbb{G}^3 \to \mathbb{Z}_q$.

The system parameters are $params =< \mathbb{G}, q, P >$.

6.1.2 Encryption and Signing:

Alice encrypts and signs a message M as follows:

- Chooses random $r, s \in \mathbb{Z}_q$.
- Computes $R = rG$, $R' = rB$, $S = sG$.
- Computes $K = H_1(R')$.
- Computes $C = Enc_K(M)$ where Enc is AES256.
- Computes $e = H_2(S, A, C)$.
- Computes $z = s + ea$.
- Outputs $(R, C, e, z) \in \mathbb{G} \times M \times \mathbb{Z}_q^2$.

6.1.3 Decryption and Verification:

Receiving (R, C, e, z), Bob decrypts the ciphertext C and verifies the signature (e, z) as follows:

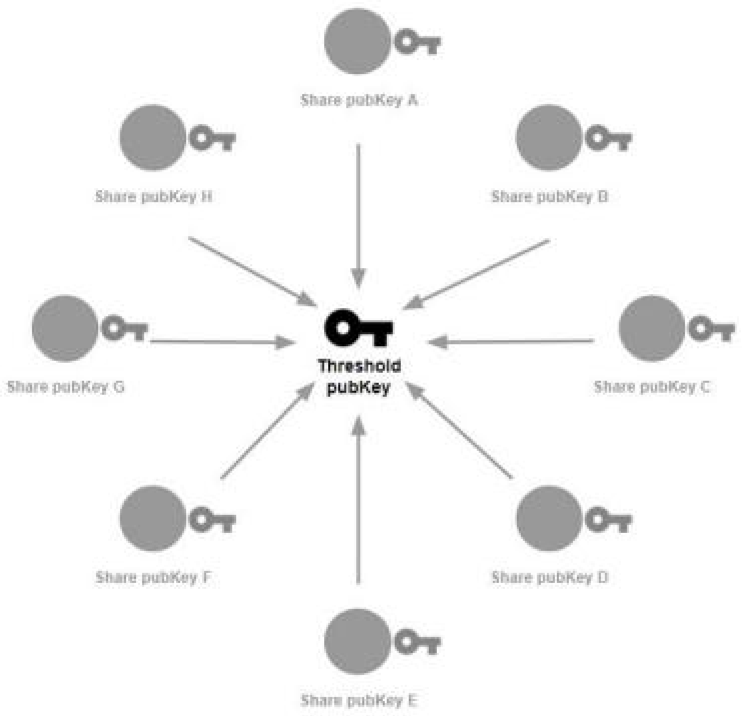
- Accept the signature if and and only if $e \overset{?}{=} H_2(zG - eA, A, C)$. Otherwise, abort the protocol.
- Computes $R' = bR$
- Computes $K = H_1(R')$.
- Decrypts C as $M = Dec_K(C)$.

Chapter 7

Clover Bridge Flow

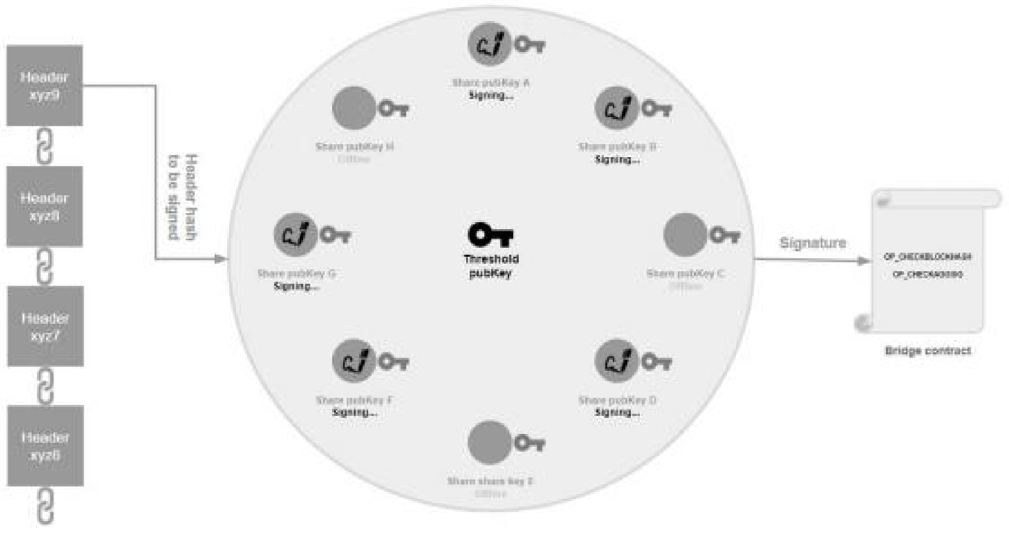
The Clover bridge is a threshold signature based protocol for notarizing the Clover and Ethereum block headers.

7.1 Setup



- During the key generation setup, the protocol in Section 4.3 will be executed (see Section 7.1) between the participants $\mathcal{P}_i$ for $i = 1, \cdots, n$.
- During the initialization phase, the participants $\mathcal{P}_i$ for $i = 1, \cdots, n$ will be chosen by the Clover foundation. At the end of the setup, followings will be generated:
  - Each $\mathcal{P}_i$ receive a partial private key sk_i.
  - The overall public key is pk which has a lifetime of 2592000 seconds (= 30 days).
  - The overall private key sk is unknown to everyone due to trustless generation.
- A bridge contract will be deployed on the Clover network.
- pk will be added to the Clover bridge contract.

7.2 Block Header Notarization



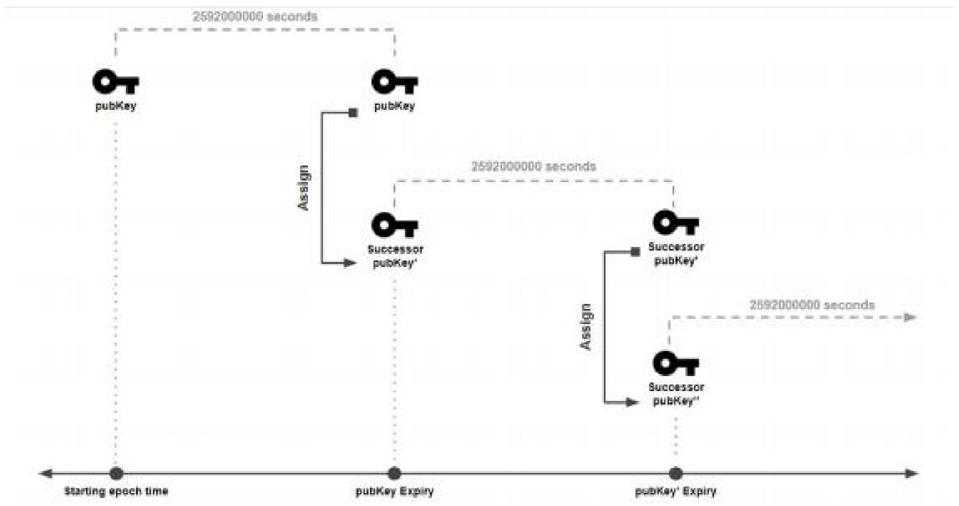
1. Each $\mathcal{P}_i$ run full nodes to listen both the Ethereum mainnet and the Clover mainnet.

2. Whenever a new block $block_j$ is observed $\mathcal{P}_i$s run a signing ceremony. At the end of the ceremony, a valid signature for the block header hash is generated. XXX**TBD: we do not need to sign each block because we dont care the blocks which does not include any tx related to our smart contract**XXX

3. The jointly generated signature $Signature_j = Sign_{sk}(Hash(block_j))$ along with the block header itself is submitted to the Clover bridge contract. In particular, *notarize_ethereum_block_header* method is called when submitting an Ethereum block header and *notarize_clover_block_header* method is called when submitting a Clover block header.

4. The Clover deployed bridge contract validates $Signature_j$ using the overall public key pk. If the signature is valid, the block header and its notary signature is stored inside the bridge contract. Note that users can read this notary signature from the bridge contract.

7.3 Providing Dynamic Signatories: Updating Signing Keys of $\mathcal{P}_i$s



1. $\mathcal{P}_i$s constantly check the expiry date of pk. If one day (24 hours) is left to the expiry date of pk, a new key generation ceremony will be executed to generate a new key pk'.

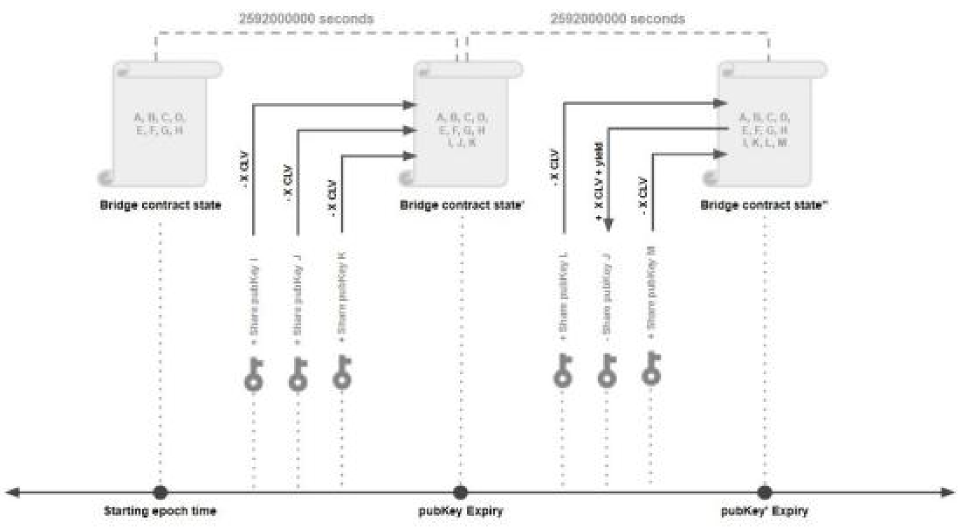
2. pk' will be signed using $\mathcal{P}_i$s's old partial private keys as $Signature = Sign_{sk}(pk')$.

3. $Signature$ along with pk' will be submitted to the Clover bridge contract. In particular, *update_ overall_ key* method is called for this event.

4. The Clover bridge contract verifies the signature $Signature$ using the public key pk. If it is valid, the current public key pk and is replaced with pk' by the contract.

5. Once $\mathcal{P}_i$s observe in the contract that pk is successfully replaced with pk', they will start using their new partial private keys for future notarizations. Note that the old partial private keys can be discarded after certain period of time as it is no longer valid.

7.4 Procedures to be a Signatory



1. During the lifetime of a public key pk, anyone with X amount of CLV can request to participate in the protocol. X amount of CLV is locked until the the key update ceremony day.

2. One day left to the expiry date of the public key pk, existing signatories vote on whether to permit new participant who requested to join.

3. If the new signatory $\mathcal{P}_k$ is permitted to join, a new key generation ceremony will be executed with the new signatory $\mathcal{P}_k$ that now commits to pk'. $\mathcal{P}_k$'s funds kept locked until next the key update.

4. If the new signatory $\mathcal{P}_k$ is not permitted to join, locked X CLV can then be redeemed from the Bridge contract.

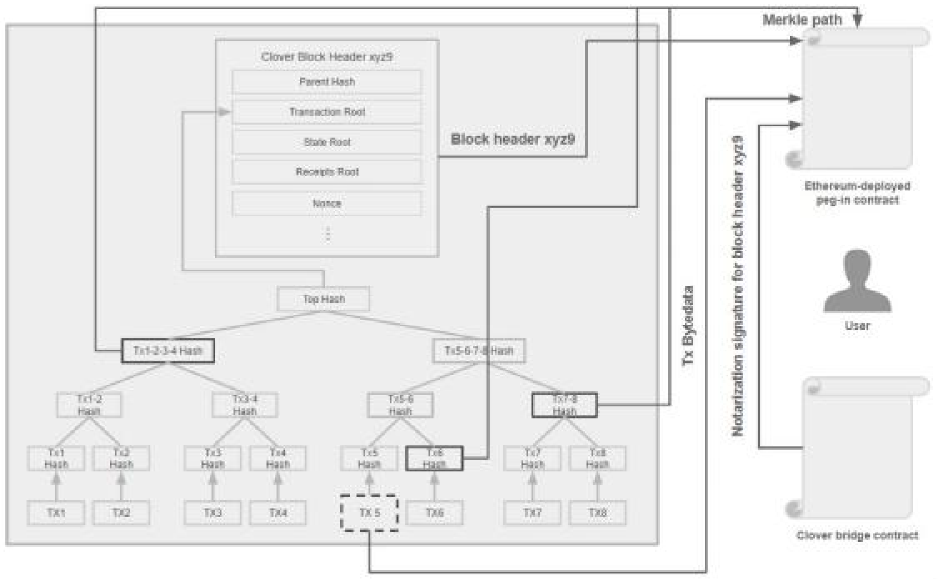
7.5 Procedures to leave from Signatory

1. During the lifetime of a public key pk, an existing signatory can request to leave the protocol.

2. One day left to the expiry date of the public key pk, a new key generation ceremony will be executed without the existing signatory.

3. The signatory who left the protocol can redeem their CLV plus some yield from the Bridge contract.

7.6 How users peg in?



1. A user willing to redeem their ethers must send cETHs to Clover-deployed peg out contract.

2. User waits several confirmation for this transaction.

3. In order to convince Ethereum-deployed peg-in contract, user must submit SPV proof for this transaction (i.e., TX5) which sent cETHs to Clover-deployed peg out contract.

4. This transaction commits to receipts root of whichever block it is part of. User submits the block header, respective Merkle path, transaction raw bytedata, and notarization signature for the block header it is part of.

5. Ethereum-deployed peg-in contract already knows the overall key which is being updated in every key update event (2592000 seconds), therefore peg in contract checks notarization signature against this public key and redeem escrowed ethers back to user if all checks are correct.

6. Users cover their own gas fee when performing this redemption.

Bibliography

[GJKR07] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin, *Secure distributed key generation for discrete-log based cryptosystems*, Journal of Cryptology **20** (2007), 51–83.

[Nat13] National Institute of Standards and Technology, *FIPS PUB 186-4: Digital signature standard*, 2013.

[Sch21] Berry Schoenmakers, *Lecture notes on cryptographic protocols*, 2021.

[SJ00] Claus Peter Schnorr and Markus Jakobsson, *Security of signed elgamal encryption*, Advances in Cryptology — ASIACRYPT 2000 (Berlin, Heidelberg), Springer Berlin Heidelberg, 2000, pp. 73–89.

[SS01] Douglas R. Stinson and Reto Strobl, *Provably secure distributed schnorr signatures and a (t, n) threshold scheme for implicit certificates*, Springer-Verlag, 2001.