Helsing: Private Masternode Staking

Aaron Feickert

January 26, 2022

This technical note reflects work in progress and has not undergone independent review. It should be considered experimental and unsuitable for production use.

1 Introduction

Firo masternodes operate by a staking mechanism. To register as a masternode, a user stakes a fixed amount of Firo as collateral to a transparent address. It then produces a registration transaction that specifies the node's identifying information, payout address, signing keys, and other auxiliary data. This transaction is signed by the collateral output's key to prove ownership. If the collateral output is later spent, the masternode is deregistered. Each coinbase transaction on the blockchain includes an output to a selected masternode's payout address, according to consensus rules.

In this technical note, we describe Helsing[1], a protocol extension to Spark [1] that allows for private staking operations not requiring transparent addresses or outputs. Specifically, Helsing provides for Spark-compatible collateral staking and coinbase payouts.

2 Cryptographic components

Throughout this technical note, let $\mathbb{G}$ be a prime-order group where the discrete logarithm problem is hard, and let $\mathbb{F}$ be its scalar field. Let

$$\mathcal{H}_{\text{stake}}, \mathcal{H}_{\text{ser}''}, \mathcal{H}_{\text{val}''}, \mathcal{H}_{\text{payout}} : \{0,1\}^* \to \mathbb{F}$$

be cryptographic hash functions selected uniformly at random. Let

$$\mathcal{H}_k, \mathcal{H}_{\text{div}}, \mathcal{H}_{\text{ser}}, \mathcal{H}_{\text{val}}$$

be the hash functions defined in [1], and let $v_{\max}$ be the maximum value parameter defined therein.

[1] In the Bram Stoker classic novel *Dracula* [2], Abraham van Helsing is a doctor and professor who aids in the pursuit and destruction of the vampiric Count Dracula. It's a great read and worth your time.

2.1 Homomorphic commitment

Helsing requires a homomorphic commitment scheme. Such a commitment scheme (at least) computationally binds a commitment to an input value and mask, and perfectly hides the value.

The commitment scheme is a function $\mathsf{Com} : \mathbb{F}^2 \to \mathbb{G}$ that is homomorphic in the sense that $\mathsf{Com}(v, m) + \mathsf{Com}(v', m') = \mathsf{Com}(v + v', m + m')$ for all values $v, v' \in \mathbb{F}$ and masks $m, m' \in \mathbb{F}$.

We assume the use of the Pedersen commitment scheme for this purpose. Let $pp_{\text{com}} = (\mathbb{G}, \mathbb{F}, G, H)$ be the public parameters for the construction, where $G, H \in \mathbb{G}$ are independent generators with no efficiently-computable discrete logarithm relationship. We then define $\mathsf{Com}(v, m) = vG + mH$ for all $(v, m) \in \mathbb{F}^2$.

Additionally, we extend this definition to a double-masked commitment scheme, which is a function $\mathsf{Comm} : \mathbb{F}^3 \to \mathbb{G}$ with similar homomorphism. We assume a natural extension of the Pedersen commitment scheme for this purpose. Let $pp_{\text{comm}} = (\mathbb{G}, \mathbb{F}, F, G, H)$ be the public parameteres for the construction, where $F, G, H \in \mathbb{G}$ are independent generators with no efficiently-computable discrete logarithm relationship. We then define $\mathsf{Comm}(v, m, m') = vF + mG + m'H$ for all $(v, m, m') \in \mathbb{F}^3$.

These constructions are perfectly hiding and computationally binding. We refer the reader elsewhere for details on the constructions and these well-known security properties.

2.2 Representation proof

Helsing requires a zero-knowledge proving system asserting that the prover knows the discrete logarithm of a given group element with respect to a specified generator.

The representation proving system is a tuple $(\mathsf{RepProve}, \mathsf{RepVerify})$ of algorithms for the following relation:

$$\{pp_{\text{rep}}, G, Y \in \mathbb{G}; y \in \mathbb{F} : Y = yG\}$$

Here $pp_{\text{rep}} = (\mathbb{G}, \mathbb{F})$ is a set of public parameters. The proving system is required to be complete, special honest-verifier zero knowledge, and special sound.

The well-known Schnorr proving system may be used for this purpose. We refer the reader elsewhere for the details and security proofs for this instantiation. Note that a proof context message may be bound to the initial proof transcript using the Fiat-Shamir construction to produce a signature.

2.3 Parallel one-of-many proof

Helsing requires a zero-knowledge proving system asserting that the prover knows the discrete logarithms of a pair of commitments within a set of such pairs. Much like in Spark spend transactions, this proof is used to produce commitment offsets to serial and value commitments of coins comprising the

commitment set. In the context of Helsing, it provides (absent external information) ambiguity as to the coin being staked, and allows the commitment offsets to be used later in an ownership and tag proof.

The proving system consists of a tuple of algorithms (ParProve, ParVerify) for the following relation:

$$\{pp_{\text{par}}, \{S_i, V_i\}_{i=0}^{N-1} \subset \mathbb{G}^2, S', V' \in \mathbb{G}; l \in \mathbb{N}, (s, v) \in \mathbb{F} : \\ 0 \leq l < N, S_l - S' = \mathsf{Com}(0, s), V_l - V' = \mathsf{Com}(0, v)\}$$

Here $pp_{\text{par}} = (\mathbb{G}, \mathbb{F}, n, m, pp_{\text{com}})$ is a set of public parameters for the construction, where $n > 1$ and $m > 1$ are integer-valued size decomposition parameters, and pp_{com} are the public parameters for a Pedersen commitment construction.

The Spark preprint describes an instantiation of such a proving system that is complete, special honest-verified zero knowledge, and special sound.

2.4 Tag proof

Helsing requires a zero-knowledge proving system asserting that for a given serial commitment offset and linking tag, the tag validly corresponds to a serial commitment represented by the offset. In particular, this means that any spend transaction consuming the coin must reveal the same tag. Further, the proof asserts that the prover knows the secret data used to produce the staked coin's serial commitment.

The tag proving system is a tuple of algorithms (TagProve, TagVerify) for the following relation:

$$\{pp_{\text{tag}}, S', T \in \mathbb{G}; (x, y, z) \in \mathbb{F}^2 : S' = xF + yG + zH, U = xT + yG\}$$

Here $pp_{\text{tag}} = (\mathbb{G}, \mathbb{F}, F, G, H, U)$ is a set of a public parameters, where the values $F, G, H, U \in \mathbb{G}$ are independent generators with no efficiently-computable discrete logarithm relationship.

The Spark preprint describes an instantiation of such a proving system that is complete, special honest-verifier zero knowledge, and special sound; it further describes how to bind a message to the proof transcript for a signature-like unforgeability property. We note that the proving system represented by the functions (ChaumProve, ChaumVerify) in [1] provides an aggregated form of this relation that can be used for the non-aggregated relation presented here.

3 Collateral staking

We now introduce a collateral staking protocol that asserts a coin within a given set is uniquely bound to a given stake value, and that a provided tag is correctly generated from the coin. This allows a masternode owner to assert it controls valid unspent collateral for node registration purposes. Further, the tag ensures that any future transaction spending the collateral can be easily detected by network participants in order to deregister the node. The protocol also allows

the owner to bind any registration-specific information, like the owner's payout address and node signing keys, to prevent replay and malicious registration.

To set up the protocol, select $F, G, H, U \in \mathbb{G}$ uniformly at random, and choose integers $n, m > 1$ as input set size parameters. Set the following component public parameters, which we refer to collectively as simply pp subsequently:

$$\begin{aligned} pp_{\text{com}} &= (\mathbb{G}, \mathbb{F}, G, H) \\ pp_{\text{comm}} &= (\mathbb{G}, \mathbb{F}, F, G, H) \\ pp_{\text{rep}} &= (\mathbb{G}, \mathbb{F}) \\ pp_{\text{par}} &= (\mathbb{G}, \mathbb{F}, n, m, pp_{\text{com}}) \\ pp_{\text{tag}} &= (\mathbb{G}, \mathbb{F}, F, G, H, U) \end{aligned}$$

Set $v_{\text{stake}} \in \mathbb{F}$ as the globally-fixed public parameter indicating the stake collateral value required, where we require $v_{\text{stake}} \in [0, v_{\max})$. Set $f \in \mathbb{F}$ as a globally-fixed public parameter indicating the fee required for the staking transaction.

We introduce two functions, Stake and StakeVerify, that comprise the collateral staking protocol.

3.1 Stake

This function is used to assert control of unspent collateral in a signer-ambiguous way.

The inputs to Stake are:

- Public parameters pp
- A full view key $\mathsf{addr}_{\text{full}}$
- A spend key $\mathsf{addr}_{\text{sk}}$
- A set of $N = n^m$ input coins InCoins as part of a cover set
- For the coin Coin of value $v+f$ to stake, its index in InCoins, serial number, tag, and nonce: (l, s, T, k)
- Any implementation-specific context m to be bound to the resulting staking transaction structure

The algorithm outputs a staking transaction.

The algorithm proceeds as follows:

1. Parse the component D from $\mathsf{addr}_{\text{full}}$.
2. Parse the components (s_1, s_2, r) from $\mathsf{addr}_{\text{sk}}$.
3. Parse the components $\{S_i, C_i\}_{i=0}^{N-1}$ from InCoins.
4. Compute the serial commitment offset:

$$S' = \mathsf{Comm}(s, 0, -\mathcal{H}_{\text{ser}''}(s, D)) + D$$

5. Compute the value commitment offset:

$$C' = \mathsf{Com}(v_{\text{stake}} + f, \mathcal{H}_{\text{val}''}(s, D))$$

6. Generate a parallel one-of-many proof:

$$\Pi_{\text{par}} = \mathsf{ParProve}(pp_{\text{par}}, \{S_i, C_i\}_{i=0}^{N-1}, S', C'; \\ (l, \mathcal{H}_{\text{ser}''}(s, D), \mathcal{H}_{\text{val}}(k) - \mathcal{H}_{\text{val}''}(s, D)))$$

7. Generate a representation proof of the collateral value:

$$\Pi_{\text{val}} = \mathsf{RepProve}(pp_{\text{rep}}, H, C' - \mathsf{Com}(v_{\text{stake}} + f, 0); \mathcal{H}_{\text{val}''}(s, D))$$

8. Generate a tag proof that additionally binds the context m into the initial transcript:

$$\Pi_{\text{tag}} = \mathsf{TagProve}((pp_{\text{tag}}, \mathcal{H}_{\text{stake}}(m)), S', T; (s, r, -\mathcal{H}_{\text{ser}''}(s, D)))$$

9. Output the tuple $(\mathsf{InCoins}, S', C', T, m, \Pi_{\text{par}}, \Pi_{\text{val}}, \Pi_{\text{tag}})$.

The context m may include data on the masternode that is required for network participants to process its registration. This may include the node's network address, signing keys, and payout Spark address. Any network participant can watch future spend transactions; if it sees a valid spend transaction revealing the tag T, then it knows the staked collateral has been spent and can deregister the masternode.

3.2 StakeVerify

This function determines the validity of the output of a staking transaction provided for masternode registration.

The inputs to StakeVerify are:

- Public parameters pp
- Staking transaction: $(\mathsf{InCoins}, S', C', T, m, \Pi_{\text{par}}, \Pi_{\text{val}}, \Pi_{\text{tag}})$

The algorithm outputs 1 if the staking transaction is valid, and 0 if it is not.

The algorithm proceeds as follows:

1. If the tag T appears in any valid Spark spend transaction, output 0.
2. Verify that the context m is valid under implementation-specific rules, and output 0 otherwise.
3. Parse the components $\{S_i, C_i\}_{i=0}^{N-1}$ from InCoins.
4. Check that $\mathsf{ParVerify}(pp_{\text{par}}, \{S_i, C_i\}_{i=0}^{N-1}, S', C'; \Pi_{\text{par}})$, and output 0 if this verification fails.

5. Check that $\mathsf{RepVerify}(pp_{\text{rep}}, H, C' - \mathsf{Com}(v_{\text{stake}} + f, 0); \Pi_{\text{val}})$, and output 0 if this verification fails.

6. Check that $\mathsf{TagVerify}((pp_{\text{tag}}, \mathcal{H}_{\text{stake}}(m)), S', T; \Pi_{\text{tag}})$, and output 0 if this verification fails.

7. Output 1.

Remark. Because collateral staking requires specification of an input ambiguity set of possible coins being staked, it is important to consider the selection of this set. In particular, a later transaction that spends the collateral and reveals the tag generated in the collateral staking will also include an input ambiguity set. The presence of these two sets with the same tag means that observers can infer that the spent collateral coin must be contained in the intersection of the two sets. For this reason, the two sets should overlap as much as possible, consistent with consensus-specific size parameters and any other input set selection rules. We note that input set selection in general is complex, and methods used should be carefully analyzed.

4 Masternode payouts

Firo coinbase transactions select a masternode (using implementation-specific rules outside the scope of this technical note) and construct a coin directed to the masternode owner's address. Currently, this is done using transparent outputs. We introduce a method for performing payouts that is consistent with Spark outputs and our staking transaction design. For this design, we assume that staking transactions include each registered masternode's Spark payout address, and that the node selected for payout has not been deregistered.

We describe a new function Payout that generates a new type of transaction, a payout transaction. Unlike mint and spend transactions in the Spark protocol, both the recipient address and payout value are public; this is required so network participants can assert the validity of the payout with respect to implementation-specific rules; this verification is done via a new PayoutVerify function.

4.1 Payout

This function generates a payout transaction directed to a masternode's Spark payout address, and asserts the validity of the payout. It assumes the existence of a block-specific identifier that is unique to the payout and can be inferred by all network participants; this is used to deterministically generate the payout and ensure unique coins in the event the node receives multiple payouts over time to the same address.

The inputs to Payout are:

- Public parameters pp

- Payout public address $\mathsf{addr}_{\mathrm{pk}}$
- Payout value v_{payout}
- Unique block-specific identifier j

The algorithm outputs a modified coin structure and auxiliary information as a payout transaction.

The algorithm proceeds as follows:

1. Parse the address values (d, Q_1, Q_2) from $\mathsf{addr}_{\mathrm{pk}}$.
2. Set $k = \mathcal{H}_{\mathrm{payout}}(j, d, Q_1, Q_2)$.
3. Compute the recovery key $K = \mathcal{H}_k(k)\mathcal{H}_{\mathrm{div}}(d)$.
4. Compute the serial commitment $S = \mathsf{Comm}(\mathcal{H}_{\mathrm{ser}}(k), 0, 0) + Q_2$.
5. Compute the value commitment $C = \mathsf{Com}(v_{\mathrm{payout}}, \mathcal{H}_{\mathrm{val}}(k))$.
6. Let $\mathsf{Coin} = (S, K, C)$, and output the tuple $(\mathsf{addr}_{\mathrm{pk}}, \mathsf{Coin})$.

4.2 PayoutVerify

This function determines the validity of a payout transaction.

The inputs to PayoutVerify are:

- Public parameters pp
- Payout coin structure: $(\mathsf{addr}_{\mathrm{pk}}, \mathsf{Coin})$
- Payout value v_{payout}
- Unique block-specific identifier j

The algorithm outputs 1 if the payout is valid, and 0 if it is not.

The algorithm proceeds as follows:

1. If $\mathsf{addr}_{\mathrm{pk}}$ does not meet implementation-specific rules as the correct payout address, output 0.
2. If v_{payout} does not meet implementation-specific rules as the correct payout value, or if $v_{\mathrm{payout}} \notin [0, v_{\max})$, output 0.
3. Run $\mathsf{Payout}(pp, \mathsf{addr}_{\mathrm{pk}}, j) = (\mathsf{addr}_{\mathrm{pk}}, \mathsf{Coin}')$.
4. If $\mathsf{Coin}' = \mathsf{Coin}$, output 1; otherwise, output 0.

Remark. The existing protocol in [1] can be easily modified to account for payout transactions. Even though we produce a modified coin structure in payout transactions, coin identification and recovery proceed mostly as in [1]. At this point, the recipient of the payout can use the coin in a standard spend transaction. Verification of payout transactions is done via PayoutVerify introduced above.

References

[1] Aram Jivanyan and Aaron Feickert. Lelantus Spark: Secure and flexible private transactions. Cryptology ePrint Archive, Report 2021/1173, 2021. `https://ia.cr/2021/1173`.

[2] Bram Stoker. *Dracula*. Doubleday, Page & Co., Garden City, N.Y., 1920.