Flow: Specialized Proof of Confidential Knowledge (SPoCK)

– Technical Note –

Tarak Ben Youssef
tarak.benyoussef@dapperlabs.com

Riad S. Wahby
rsw@cs.stanford.edu

September 2020

Abstract

Flow is a high-throughput blockchain with a dedicated step for executing the transactions in a block and a subsequent verification step performed by Verification Nodes [1, 2, 3]. To enforce integrity of the blockchain, the protocol requires a component that prevents Verification Nodes from approving execution results without checking. In our preceding work, we have sketched out an approach called Specialized Proof of Confidential Knowledge (SPoCK). Using SPoCK, nodes can provide evidence to a third party that they both executed the same transaction sequence without revealing the resulting execution trace. The previous white paper [3] presented a basic implementation of such scheme.

In this note, we introduce a new SPoCK implementation that is more concise and more efficient than the previous proposal. We first provide a formal generic description of a SPoCK scheme as well as its security definition. Then we propose a new construction of SPoCK based on the BLS signature scheme [4]. We support the new scheme with its proof of security under the appropriate computation assumptions.

1 Introduction

Flow is a blockchain that achieves high-throughput based on a pipelined architecture, where nodes with dedicated roles perform different sets of tasks. Specifically, Execution Nodes perform the heavy computation of all transactions in a block. The Flow protocol introduces a mechanism to detect and attribute faulty results by assigning Verification Nodes to re-compute the block transactions. The block computation is broken up into chunks for a parallelized and distributed verification. The Security Nodes (consensus nodes) make sure the computation was checked by a sufficient number of Verification Nodes and then commit the execution result into the chain. To enforce integrity of the blockchain, the protocol requires a mechanism that prevents "lazy" Verification Nodes from approving execution results without checking.

A Specialized Proof of Confidential Knowledge (SPoCK) is a personalized commitment to a secret. While the mechanism can be based on any type of secret, the secret in Flow is generated from the execution trace of a chunk. We assume the execution trace cannot be derived more cheaply than by executing the entire chunk. An Execution Node provides a SPoCK as a commitment to their execution traces for each chunk in a block. A Verification Node provides a SPoCK as a commitment to their own execution traces for each chunk they verify. The role of the Security Nodes is to arbitrate by verifying that the SPoCKs of each chunk are consistently generated from the same trace. Although the protocol prevents "lazy" Verification Nodes from approving chunks without re-computing them, it does not prevent or detect collusion with any party that has already computed the trace. To prevent such a collusion, a minority of honest Verification Nodes checking a wrong chunk result is sufficient to prevent it from being committed into the chain with high probability. It also leads to slashing the Execution Node(s) that generated the wrong result as well as all the Verification Nodes that approved it.

In this note, we take a more formal approach to define an abstract SPoCK scheme. Based on Flow's execution–verification use-case, we define the properties that a SPoCK scheme is expected to achieve. The definitions and properties are considered without any assumption on the possible constructions of the scheme. At a high level, these properties are, first that proofs generated by a honest Execution and verification Nodes from the same secret message have to be accepted by a honest Security Node, and second that proofs are unforgeable by any malicious node in two different game contexts.

Then we propose a new construction of SPoCK based on the BLS signature scheme [4]. The proof generation is straightforward and is as simple as a BLS signature generation of the secret message. The verification, although simple, is less standard and benefits from the same pairing properties used in the BLS signature verification. We note that all the private and public keys used are the staking keys of the nodes participating in the Flow protocol.

The BLS-based SPoCK is basically a scheme that allows a party to check whether two or more signatures have signed the same secret message without learning more about the secret itself. To the best of our knowledge, no previous work has described such a scheme. We analyze the security of the new BLS-based implementation based on the generic SPoCK unforgeability definitions and prove its security under standard computational assumptions. The BLS-based SPoCK construction in particular also offers some elegant properties beyond the ones required by a SPoCK scheme in Flow.

Relation to the Verifier's Dilemma

The Verifier's Dilemma can arise in distributed systems where some participating nodes are supposed to verify the work of other node(s) [5]. Consider a system with the following properties.

- Compensation for a verifier increases (statistically) with its speed.
- The results, which the verifiers are checking, are correct with a probability sufficiently close to 1.

Most blockchains have built-in incentives for nodes to run as fast as possible and to deliver correct results. Even if the verifiers are compensated for their work, blindly approving all results can still be the most profitable strategy in such an environment, because this strategy not only saves time but also expenditures for hardware and energy for verification work. Nodes adopting such strategy undermine the network's resilience to malicious actors in the long run.

Flow mitigates the Verifier's Dilemma through its architecture in combination with SPoCK. Specifically, Verification Nodes have to prove they know the execution trace of the chunks they were assigned to in the form of SPoCK; they are slashed for approving wrong results; and a minority of $\frac{1}{3}$ honest Verification Nodes checking a wrong result is sufficient to slash the Execution Node(s) that generated the wrong result as well as all the Verification Nodes that approved it.

There are other techniques for addressing the Verifier's Dilemma, such as truebit's proposals [6] or Arbitrum [7, 8]. For brevity of this technical note, we refrain from a detailed discussion of the broader academic context and related work.

2 Preliminaries

2.1 co-CDH and related assumptions

Let G_1 and G_2 be two cyclic groups of prime order, and let g_1 and g_2 be generators of G_1 and G_2 respectively. The Computational co-Diffie-Hellman (co-CDH) problem [4] is to compute $g_1^{x \cdot y}$ given $(g_1, g_1^x, g_1^y, g_2, g_2^y)$. The co-CDH assumption states that no probabilistic polynomial-time algorithm solves the co-CDH problem with a non-negligible probability.

A related problem is the Divisible Computational co-Diffie Hellman problem (co-DCDH) [9]: given $(g_1, g_1^{x \cdot y}, g_1^y, g_2, g_2^y)$, compute g_1^x. The co-DCDH assumption is analogous to the co-CDH assumption.

Similarly to the work of Bao et al. [10], we show that these two assumptions are equivalent.

Lemma 1. *The co-DCDH and co-CDH assumptions in (G_1, G_2) are equivalent.*

Proof. We show that an adversary $\mathcal{A}$ that solves the co-DCDH problem also solves the co-CDH problem and vice versa.

- co-CDH $\Rightarrow$ co-DCDH:
  $\mathcal{A}$ is given a co-DCDH challenge $(g_1, g_1^{x.y}, g_1^y, g_2, g_2^y)$ and has access to a co-CDH solver algorithm $\mathcal{A}_{\text{co-CDH}}$ such that $\mathcal{A}_{\text{co-CDH}}(r_1, r_1^a, r_1^b, r_2, r_2^b) = r_1^{a.b}$ for any random (r_1, r_2) in $G_1 \times G_2$.
  $\mathcal{A}$ computes $\mathcal{A}_{\text{co-CDH}}(g_1^y, g_1^{x.y}, g_1, g_2^y, g_2) = \mathcal{A}_{\text{co-CDH}}(g_1^y, (g_1^y)^x, (g_1^y)^{y^{-1}}, g_2^y, (g_2^y)^{y^{-1}})$ which outputs $(g_1^y)^{x.y^{-1}} = g_1^x$ and solves the co-DCDH challenge.
- co-CDH $\Leftarrow$ co-DCDH:
  $\mathcal{A}$ is given a co-CDH challenge $(g_1, g_1^x, g_1^y, g_2, g_2^y)$ and has access to a co-DCDH solver algorithm $\mathcal{A}_{\text{co-DCDH}}$ such that $\mathcal{A}_{\text{co-DCDH}}(r_1, r_1^{a.b}, r_1^b, r_2, r_2^b) = r_1^{a.b.b^{-1}} = r_1^a$ for any random (r_1, r_2) in $G_1 \times G_2$.
  $\mathcal{A}$ computes $\mathcal{A}_{\text{co-DCDH}}(g_1^y, g_1^x, g_1, g_2^y, g_2) = \mathcal{A}_{\text{co-DCDH}}(g_1^y, (g_1^y)^{x.y^{-1}}, (g_1^y)^{y^{-1}}, g_2^y, (g_2^y)^{y^{-1}})$ which outputs $(g_1^y)^{x.y^{-1}.(y^{-1})^{-1}} = g_1^{x.y}$ and solves the co-CDH challenge.

The two problems are equivalent and therefore the two assumptions are also equivalent. □

2.2 BLS signatures

We first briefly review the BLS signature scheme [4]. Let G_1, G_2 and G_T be three cyclic groups of prime order p where (G_1, G_2) is a bilinear group pair [4, Def. 2.2]. Let e be an efficiently computable non-degenerate pairing $e : G_1 \times G_2 \to G_T$, and H be a hash function $H : \{0,1\}^\star \to G_1$ (modeled as a random oracle). The multiplicative notation is used for the three groups. The signature scheme is defined as follows:

- $\mathsf{BLS\text{-}KeyGen}() \xrightarrow{\text{R}} (sk, pk)$, where $sk \xleftarrow{\text{R}} \mathbb{Z}_p$ and $pk \leftarrow g_2^{sk} \in G_2$.
- $\mathsf{BLS\text{-}Sign}(sk, m) \to \sigma$, where $\sigma \leftarrow H(m)^{sk} \in G_1$.
- $\mathsf{BLS\text{-}Verify}(pk, m, \sigma) \to v \in \{\mathsf{OK}, \mathsf{FAIL}\}$, where
  v is OK if $e(\sigma, g_2) = e(H(m), pk)$ and FAIL otherwise.

Boneh et al. proved the signature scheme is secure against existential forgery under the chosen message attack, in the random oracle and under the co-CDH assumption in (G_1, G_2).

2.3 Registered key and knowledge of secret key models

Ristenpart et al. [11] define the *registered key* model, which is a protocol $\mathcal{R} = (\mathsf{Reg\text{-}Prove}, \mathsf{Reg\text{-}Verify})$:

- $\mathsf{Reg\text{-}Prove}(sk, pk) \to \pi$ generates a registration proof.
- $\mathsf{Reg\text{-}Verify}(pk, \pi) \to \{\mathsf{OK}, \mathsf{FAIL}\}$ outputs OK if the proof is valid for pk and FAIL otherwise.

If no key registration is required, $(\mathsf{Reg\text{-}Prove}, \mathsf{Reg\text{-}Verify})$ can be replaced with vacuous functions as follows: $\mathsf{Reg\text{-}Prove}$ outputs the empty string, and $\mathsf{Reg\text{-}Verify}$ outputs OK on any input.

In Section 4, we consider a class of key registration protocols in which parties prove knowledge of the secret corresponding to their public key; this is called the *knowledge of secret key* (KOSK) model [12]. In this model, all parties have access to the functions $\mathcal{R}_{KOSK} = (\mathsf{KOSK\text{-}Prove}, \mathsf{KOSK\text{-}Verify})$ which generate and verify proofs, respectively.

To instantiate this model, we require parties to use a zero-knowledge proof of knowledge (ZKPK) of their secret. Key registration protocols based on ZKPKs provide two additional algorithms, $\mathsf{KOSK\text{-}Simulate}$ and $\mathsf{KOSK\text{-}Extract}$, which they inherit from the underlying ZKPK:

- $\mathsf{KOSK\text{-}Simulate}(pk) \to \pi$ is a *simulator* that, given access to the randomness used for proof verification, outputs a proof that is computationally indistinguishable from a real proof.
- $\mathsf{KOSK\text{-}Extract}(pk, \pi) \to sk$ is an *extractor* that interacts with (in particular, rewinds) the party who generated π to output sk from a convincing proof.

Simulation and extraction are standard notions for ZKPKs, so we do not define them more formally; interested readers should consult [13, Ch. 19–20].

3 SPoCK scheme

In Flow, honest Execution and Verification Nodes generate a SPoCK proof using their staking private keys and a secret that we call *confidential knowledge*. A Security Node verifies the SPoCK proofs generated by execution and verification nodes using the corresponding public keys. The verification process should ensure that all proofs were generated based upon the same confidential knowledge. Based on this use-case, we give a generic definition of a SPoCK scheme.

3.1 Scheme definition

Definition 1. A SPoCK scheme comprises four algorithms:

- $\mathsf{SP\text{-}Setup}(1^\lambda) \xrightarrow{\text{R}} pp$. Sample random public parameters pp with security parameter λ. pp is an implicit input to the remaining algorithms.
- $\mathsf{SP\text{-}KeyGen}() \xrightarrow{\text{R}} (sk, pk)$. Sample a random private and public key pair.
- $\mathsf{SP\text{-}Prove}(sk, m) \to \sigma$. Construct a SPoCK proof σ for message m under secret key sk. (Note that SP-Prove is a deterministic algorithm.)
- $\mathsf{SP\text{-}Verify}(pk_a, \sigma_a, pk_b, \sigma_b) \to v \in \{\mathsf{OK}, \mathsf{FAIL}\}$. For sk_a the secret corresponding to pk_a, and likewise sk_b to pk_b, return OK if

$$\exists m \ : \ \mathsf{SP\text{-}Prove}(sk_a, m) = \sigma_a \ \wedge \ \mathsf{SP\text{-}Prove}(sk_b, m) = \sigma_b$$

  and FAIL otherwise.

If two proofs are generated honestly from the same confidential knowledge, SP-Verify is required to output OK, which defines the *correctness* of the scheme.

Definition 2. Correctness. A SPoCK scheme is *correct* if

$$\Pr\left[\begin{array}{l|rl} & (sk_a, pk_a) \leftarrow \mathsf{SP\text{-}KeyGen}() \\ & (sk_b, pk_b) \leftarrow \mathsf{SP\text{-}KeyGen}() \\ \mathsf{SP\text{-}Verify}(pk_a, \sigma_a, pk_b, \sigma_b) = \mathsf{OK} & : & m \xleftarrow{\text{R}} \{0,1\}^l \\ & \sigma_a \leftarrow \mathsf{SP\text{-}Prove}(sk_a, m) \\ & \sigma_b \leftarrow \mathsf{SP\text{-}Prove}(sk_b, m) \end{array}\right] = 1$$

where l is an upper bound on the message length.

We also define SPoCK in the Registered Key Model (§2.3). In this case, the correctness condition holds only with respect to registered keys.

The SP-Verify definition only requires the existence of a message m consistent with σ_a and σ_b. While this definition suffices for two parties, extending it to three or more parties is more subtle. To see why, consider the case that the owners of sk_a and sk_b share a secret m_1, while the owners of sk_a and sk_c share a secret m_2. By the definition of SP-Verify, $\mathsf{SP\text{-}Verify}(pk_a, \sigma_a, pk_b, \sigma_b)$ and $\mathsf{SP\text{-}Verify}(pk_a, \sigma_a, pk_c, \sigma_c)$ might both output OK even when $m_1 \neq m_2$. As a result, these two checks are not sufficient to guarantee that the owners of sk_b and sk_c share *any* secret.

Recall, however, that in Flow we require the verification process to ensure that the holders of sk_a, sk_b, and sk_c all share the same secret knowledge m (and likewise for groups larger than three). To reflect this property, we define *strong transitivity* for a SPoCK scheme:

Definition 3. Strong transitivity. A SPoCK scheme satisfies strong transitivity if for all integer $n \geq 3$, for all valid key-pairs $(sk_1, pk_1), ..., (sk_n, pk_n)$, and for all proofs $\sigma_1, ..., \sigma_n$,

$$\left(\begin{array}{l} \forall i \in \{2, \ldots, n\}, \\ \mathsf{SP\text{-}Verify}(pk_1, \sigma_1, pk_i, \sigma_i) = \mathsf{OK} \end{array} \right) \Rightarrow \left(\begin{array}{l} \exists m : \\ \forall i \in \{1, \ldots, n\}, \mathsf{SP\text{-}Prove}(sk_i, m) = \sigma_i \end{array} \right)$$

Notice that if there exists a message m such that all σ_i are consistent with m (i.e., if the right side of the above implication holds), then $\forall i, j, 1 \leq i, j \leq n$, $\mathsf{SP\text{-}Verify}(pk_i, \sigma_i, pk_j, \sigma_j) = \mathsf{OK}$.

The strong transitivity definition states that if multiple proofs verify against a same reference proof σ_1, then all proofs verify against each other, and there exists a single message m from which all these proofs (including the reference proof) could be generated.

In Flow, an Execution Node generates a reference proof, while multiple proofs are generated by the verification nodes. For a SPoCK scheme satisfying strong transitivity, it suffices to verify all proofs against the execution node's proof to ensure all nodes share the same secret knowledge.

3.2 Scheme security

A Security Node does not require any information about the confidential knowledge, other than two SPoCK proofs and the corresponding keys, to run the SP-Verify algorithm. Intuitively, the confidential knowledge should remain secret to all parties in the network except those who know the execution trace of a block. This means that a SPoCK proof should not allow recovering the confidential knowledge.

More specifically, the SPoCK scheme should be resistant against recovering the secret knowledge from a SPoCK proof or forging a SPoCK proof without access to that knowledge. In Flow, such attacks might be mounted by "lazy" verification nodes that aim to skip costly block execution while claiming they know the secret. Attackers may also attempt to forge a proof on behalf of another node, for example, to accumulate more approvals on a faulty result.

Intuitively, generating a SPoCK proof requires knowing two secrets, a key sk and a message m. We formalize this intuition via two security games, each between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. The first game, *knowledge-forgery*, models the ability of a "lazy" node to forge a proof under its own key without knowing the confidential knowledge m. The second game, *key-forgery*, models the ability of a malicious node to create a proof for some chosen m under another node's public key without knowing the corresponding secret key. We define these games assuming a key registration scheme (Reg-Prove, Reg-Verify), which can be the vacuous scheme (§2.3) if no registration is required.

Definition 4. The knowledge-forgery game.

- **Setup**. $\mathcal{C}$ samples a random message $m \xleftarrow{R} \{0,1\}^l$ and initializes an empty list L_k.
- **Query**. $\mathcal{A}$ makes any number of queries to $\mathcal{C}$. On each such query, $\mathcal{C}$ samples a fresh key $(sk_i, pk_i) \xleftarrow{R} \mathsf{SP\text{-}KeyGen}()$, computes a registration proof $\pi_i \leftarrow \mathsf{Reg\text{-}Prove}(sk_i, pk_i)$ and a SPoCK proof $\sigma_i \leftarrow \mathsf{SP\text{-}Prove}(sk_i, m)$, and sends (pk_i, π_i, σ_i) to $\mathcal{A}$. Finally, $\mathcal{C}$ adds pk_i to L_k.
- **Output**. $\mathcal{A}$ outputs (pk_a, π_a, σ_a) and wins if

$$\begin{array}{lll} & pk_a \notin L_k & \textit{(pk_a is fresh)} \\ \wedge & \mathsf{Reg\text{-}Verify}(pk_a, \pi_a) = \mathsf{OK} & \textit{(π_a is valid for pk_a)} \\ \wedge & \exists i : \mathsf{SP\text{-}Verify}(pk_a, \sigma_a, pk_i, \sigma_i) = \mathsf{OK} & \textit{(σ_a is consistent with m and some pk_i)} \end{array}$$

Definition 4 assumes that the message m is sampled at random from a large message space, and that the adversary knows nothing about m. In particular, it does not apply if m is structured or if the adversary knows, say, half the bits of m. We note, however, that this definition suffices if the SPoCK scheme is instead applied to $H(m)$ for a random oracle $H(\cdot)$. In addition, it must be infeasible to simply guess m or $H(m)$. Both of these requirements are satisfied in Flow.

Definition 5. The key-forgery game.

- **Setup**. $\mathcal{C}$ samples $(sk_c, pk_c) \xleftarrow{R} \mathsf{SP\text{-}KeyGen}()$, computes $\pi_c \leftarrow \mathsf{Reg\text{-}Prove}(sk_c, pk_c)$, and sends (pk_c, π_c) to $\mathcal{A}$. $\mathcal{C}$ also initializes an empty list L_m.
- **Query**. $\mathcal{A}$ makes any number of queries to $\mathcal{C}$. On each such query, $\mathcal{A}$ sends m_i to $\mathcal{C}$, who computes $\sigma_i \leftarrow \mathsf{SP\text{-}Prove}(sk_c, m_i)$ and sends σ_i to $\mathcal{A}$. Finally, $\mathcal{C}$ adds m_i to L_m.
- **Output**. $\mathcal{A}$ outputs (m_a, σ_a) and wins if

$$\begin{aligned} & m_a \notin L_m && \textit{(m_a is fresh)} \\ \wedge \quad & \sigma_a = \mathsf{SP\text{-}Prove}(sk_c, m_a) && \textit{(σ_a is the proof for m_a under sk_c)} \end{aligned}$$

  (Note that this winning condition uses the fact that SP-Prove is deterministic.)

One could define a third game to capture the case where an adversary does not have either the confidential knowledge or the secret key. This case corresponds to forging a SPoCK proof in Flow to claim a target node has access to some secret when the attacker does not know either the secret or the target's key. This forgery is clearly harder than the two other games: an adversary with an algorithm that succeeds at such a forgery could easily win either of the other two games. We therefore do not consider this game.

Definition 6. Unforgeability. We say that a SPoCK scheme is:
- *secure against knowledge-forgery* if no probabilistic polynomial-time adversary $\mathcal{A}$ wins knowledge-forgery, except with at most negligible probability.
- *secure against key-forgery* if no probabilistic polynomial-time adversary $\mathcal{A}$ wins key-forgery, except with at most negligible probability.
- *unforgeable* if it is secure against both knowledge-forgery and key-forgery.

We define one further property of a SPoCK scheme, *non-malleability*. Intuitively, for a proof σ_b and two public keys pk_a and pk_b, given a proof σ_a that verifies against (σ_b, pk_b, pk_a) it is infeasible to produce a distinct proof σ'_a that also verifies against (σ_b, pk_b, pk_a). Flow does not require SPoCK to have this property, but in practice non-malleability can eliminate subtle attacks.

Definition 7. Non-malleability. A SPoCK scheme is *non-malleable* if for all probabilistic polynomial-time adversaries $\mathcal{A}$,

$$\Pr\left[\begin{array}{r} \sigma'_a \neq \sigma_a \quad \wedge \\ \mathsf{SP\text{-}Verify}(pk_a, \sigma'_a, pk_b, \sigma_b) = \mathsf{OK} \end{array} \;:\; \begin{array}{r} (sk_a, pk_a) \leftarrow \mathsf{SP\text{-}KeyGen}() \\ (sk_b, pk_b) \leftarrow \mathsf{SP\text{-}KeyGen}() \\ m \xleftarrow{R} \{0,1\}^l \\ \sigma_a \leftarrow \mathsf{SP\text{-}Prove}(sk_a, m) \\ \sigma_b \leftarrow \mathsf{SP\text{-}Prove}(sk_b, m) \\ \sigma'_a \leftarrow \mathcal{A}(m, pk_a, \sigma_a, pk_b, \sigma_b) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

where $\mathsf{negl}(\lambda)$ is a negligible function in the security parameter λ.

A slightly stronger notion related to non-malleability is *uniqueness*:

Definition 8. Uniqueness. A SPoCK scheme satisfies uniqueness if for all proofs σ_a and for all public keys pk_a and pk_b, there exists a unique proof σ_b such that $\mathsf{SP\text{-}Verify}(pk_a, \sigma_a, pk_b, \sigma_b) = \mathsf{OK}$.

Corollary 1. *A SPoCK scheme satisfying uniqueness is also non-malleable.*

4 SPoCK based on BLS signatures

In this section, we define a SPoCK scheme based on BLS signatures (§2.2), and prove that this scheme is unforgeable (Def. 6) when instantiated in the KOSK model (§2.3).

4.1 BLS-SPoCK Scheme

Definition 9. BLS-SPoCK comprises four algorithms: BSP-Setup, BSP-KeyGen, BSP-Prove and BSP-Verify.

- $\mathsf{BSP\text{-}Setup}(1^\lambda) \xrightarrow{R} pp_{BLS}$: Output public parameters comprising:
  - A bilinear group pair (G_1, G_2) of order p with generators g_1 and g_2, respectively.
  - A target group G_T of order p.
  - A non-degenerate pairing $e : G_1 \times G_2 \to G_T$.
  - A hash function $H : \{0,1\}^\star \to G_1$ modeled as a random oracle.
- $\mathsf{BSP\text{-}KeyGen}() \xrightarrow{R} (sk, pk)$: Output $(sk, pk) \xleftarrow{R} \mathsf{BLS\text{-}KeyGen} \in (\mathbb{Z}_p \times G_2)$.
- $\mathsf{BSP\text{-}Prove}(sk, m) \to \sigma$: Output $\sigma \leftarrow \mathsf{BLS\text{-}Sign}(sk, m) \in G_1$.
  In other words, σ is a BLS signature of the message m under the private key sk.
- $\mathsf{BSP\text{-}Verify}(pk_a, \sigma_a, pk_b, \sigma_b) \to v \in \{\mathsf{OK}, \mathsf{FAIL}\}$: Output OK if

$$e(\sigma_a, pk_b) = e(\sigma_b, pk_a) \tag{1}$$

  Otherwise, output FAIL.

Lemma 2. *The BLS-SPoCK scheme is a correct SPoCK scheme.*

Proof. For any message m, and key pairs (sk_a, pk_a), (sk_b, pk_b), by the definition of e we have $e(\mathsf{BSP\text{-}Prove}(sk_a, m), pk_b) = e(H(m)^{sk_a}, g_2^{sk_b}) = e(H(m), g_2)^{sk_a \cdot sk_b} = e(H(m)^{sk_b}, g_2^{sk_a})$. This means that $e(\mathsf{BSP\text{-}Prove}(sk_a, m), pk_b) = e(\mathsf{BSP\text{-}Prove}(sk_b, m), pk_a)$, satisfying Definition 2. □

Lemma 3. *The BLS-SPoCK scheme satisfies strong transitivity.*

Proof. Let an integer n be larger than 3, and let a set of n valid key-pairs be $(sk_1, pk_1), ..., (sk_n, pk_n)$, and a set of n proofs be $\sigma_1, ..., \sigma_n$ such that:

$$\forall i \in \{2, \ldots, n\} : \mathsf{BSP\text{-}Verify}(pk_1, \sigma_1, pk_i, \sigma_i) = \mathsf{OK}$$

By the bilinearity of the pairing e, and since G_T is a cyclic group of a prime order, we deduce that ${\sigma_1}^{sk_i} = {\sigma_i}^{sk_1}$ and thus ${\sigma_1}^{{sk_1}^{-1}} = {\sigma_i}^{{sk_i}^{-1}}$ for all $1 \leq i \leq n$. Then $h = {\sigma_1}^{{sk_1}^{-1}} = \ldots = {\sigma_n}^{{sk_n}^{-1}}$ is an element in G_1 for which some message m exists satisfying $H(m) = h$. The message m clearly satisfies $\sigma_i = H(m)^{sk_i} = \mathsf{BSP\text{-}Prove}(sk_i, m)$ for all $1 \leq i \leq n$, which establishes strong transitivity. □

Corollary 2. *The BLS-SPoCK scheme satisfies uniqueness (Def. 8).*

Proof. Let $pk_a, pk_b \in G_2$ and $\sigma_a \in G_1$. Notice that an element of G_1 that verifies against σ_a, pk_a and pk_b can be written as $\sigma = \sigma_a^{{sk_b \cdot sk_a}^{-1}}$. σ is therefore the unique element of G_1 that verifies against σ_a, pk_a and pk_b. □

Corollary 3. *The BLS-SPoCK scheme is non-malleable.*

4.2 Security proof

In this section, we prove BLS-SPoCK is secure against forgery as in Definition 6. We assume that BLS-SPoCK is instantiated in the KOSK model (§2.3) using a Schnorr zero-knowledge proof of knowledge of discrete log [14]. It may be possible to prove security when BLS-SPoCK is instantiated with *proofs of possession* [11] rather than proofs of knowledge; this is future work.

Remark 1. We stress that key registration is crucial to the scheme's security: in its absence, an adversary can easily win the knowledge-forgery game (Def. 4). For example, for any key (sk, pk) and any proof σ under sk, $\mathsf{BSP\text{-}Verify}(pk, \sigma, pk^2, \sigma^2) = \mathsf{OK}$. Informally, requiring key registration ensures that an adversary attempting to produce such a forgery must first break pk.

Theorem 1. *The BLS-SPoCK scheme is secure against knowledge-forgery under the co-CDH assumption in (G_1, G_2) in the KOSK model.*

Proof. We show that an adversary $\mathcal{A}_{know}$ that breaks knowledge-forgery can be used as a black box to break co-DCDH (and thus co-CDH; Lem 1, §2.1). To do so, we construct an algorithm $\mathcal{C}_{know}$ that acts as the challenger for the knowledge-forgery game and the adversary for the co-DCDH game.

To begin, $\mathcal{C}_{know}$ requests a co-DCDH challenge $(g_1, g_1^{x \cdot y}, g_1^y, g_2, g_2^y)$. On each of $\mathcal{A}_{know}$'s queries, $\mathcal{C}_{know}$ samples $r_i \xleftarrow{R} \mathbb{Z}_n$; sets $pk_i = (g_2^y)^{r_i}$, $\sigma_i = \left(g_1^{x \cdot y}\right)^{r_i}$, and $\pi_i \leftarrow \mathsf{KOSK\text{-}Simulate}(pk_i)$; and sends (pk_i, π_i, σ_i) to $\mathcal{A}_{know}$. Finally, $\mathcal{A}_{know}$ responds with (pk_a, π_a, σ_a) and $\mathcal{C}_{know}$ aborts if $\mathsf{KOSK\text{-}Verify}(pk_a, \pi_a) = \mathsf{FAIL}$ or if $\nexists i : \mathsf{BSP\text{-}Verify}(pk_a, \sigma_a, pk_i, \sigma_i) = \mathsf{OK}$.

Assume that $\mathcal{C}_{know}$ does not abort, which happens with non-negligible probability by the assumption that $\mathcal{A}_{know}$ breaks knowledge-forgery. Then $\mathcal{C}_{know}$ wins the co-DCDH game by first computing $sk_a \leftarrow \mathsf{KOSK\text{-}Extract}(\pi_a)$, then answering $\sigma_a^{sk_a^{-1}}$ for the co-DCDH game.

To see why this works, notice that since $\mathsf{BSP\text{-}Verify}$ returned OK for some i, we have that $e(\sigma_a, pk_i) = e(\sigma_i, pk_a)$, meaning that $\sigma_a = g_1^{sk_a \cdot x}$ and thus $\sigma_a^{sk_a^{-1}} = g_1^x$, the correct co-DCDH answer. Further, all pk_i, π_i, and σ_i are distributed as in the real knowledge-forgery game: $sk_i = y \cdot r_i$ is a uniformly random secret key corresponding to pk_i, $\sigma_i = (g_1^x)^{sk_i}$, and π_i is indistinguishable from a KOSK proof by the definition of $\mathsf{KOSK\text{-}Simulate}$.

Thus, $\mathcal{C}_{know}$ wins the co-DCDH game just when $\mathcal{A}_{know}$ wins, $\mathsf{KOSK\text{-}Simulate}$ outputs a convincing π_i, and $\mathsf{KOSK\text{-}Extract}$ outputs sk_a. $\mathsf{KOSK\text{-}Simulate}$ and $\mathsf{KOSK\text{-}Extract}$ succeed with overwhelming probability by definition, so $\mathcal{C}_{know}$ wins the co-DCDH game with non-negligible probability. By Lemma 1 this contradicts the assumption that co-CDH is hard, so $\mathcal{A}_{know}$ cannot win knowledge-forgery with non-negligible probability. □

Theorem 2. *The BLS-SPoCK scheme is secure against key-forgery in the random oracle model, under the co-CDH assumption in (G_1, G_2).*

Proof. To win the key-forgery game, $\mathcal{A}$ must forge a BLS-SPoCK proof $\sigma_a = \mathsf{BSP\text{-}Prove}(sk_c, m_a))$, which is the BLS signature for message m_a with respect to the honestly-generated key pair (sk_c, pk_c). In other words, winning key-forgery implies an existential forgery on m_a under sk_c. Boneh et al. [4] prove security of BLS signatures against existential forgery for a chosen message (i.e., EUF-CMA security) in the random oracle model and under the co-CDH assumption for (G_1, G_2), which proves the theorem. □

Corollary 4. *The BLS-SPoCK scheme is unforgeable in the KOSK and random oracle model under the co-CDH assumption in (G_1, G_2).*

5 Conclusion

Starting from the Flow use-case, we have defined a generic SPoCK scheme and outlined the different required security properties. The scheme allows any party to verify two proofs have been generated using the same secret message, without having access to the message itself. We have presented both a generic scheme and an instantiation based on BLS signatures.

We have presented an alternative to the current construction of SPoCK in Flow. The new scheme is based on the BLS signature scheme and inherits many of its nice properties. A proof in the new construction is as simple as a BLS signature. Therefore the scheme can be used in more generic use-cases beyond Flow to prove two BLS signatures were generated from a same secret message. In a standard signature scheme, only the private key is treated as a secret data while the message is considered a public data. The novelty of the BLS based SPoCK is that both the private key and the message are treated as secret data. The generated proofs do not reveal the message, but allow verifying whether multiple signatures have signed the same message.

5.1 Future work: Proof of Possession

In this work, unforgeability of the BLS-SPoCK scheme is proved under the strong KOSK assumption, in particular a zero-knowledge proof of knowledge of discrete log. The same unforgeability security should hold under a more relaxed registration model, namely a proof of possession (POP) [11]. POP is a simpler and more practical assumption than KOSK: rather than generating a proof of knowledge, each node simply computes a BLS signature of its own public key with respect to a random oracle that is independent of the one output by BSP-Setup (§4).

5.2 Future work: Aggregation

The BLS signature scheme takes advantage of the elegant bilinear group pair structure to provide the nice features of signature aggregations. The BLS-SPoCK scheme inherits similar properties and allows aggregating multiple proofs in a single one. Aggregating multiple proofs is equivalent to multiplying a set of elements in G_1, while aggregating public keys is equivalent to multiplying a set of elements in G_2. The aggregation optimizes the size of the overall proofs linearly, and allows a faster batch verification of all the proofs. A batch verification of all the proofs against a single reference proof should be equivalent to calling BSP-Verify on the reference proof and public key along with the aggregated signature and aggregated public key. The security proofs and assumptions will be in the scope of a future work.

In Flow, a Security Node makes sure the proofs of multiple verification nodes are consistent with a reference proof of an Execution node. The Security Nodes would take advantage of the batching technique to verify all the Verification Nodes' proofs at once.

Acknowledgments

We thank Dan Boneh for the insightful comments and suggestions on the draft of this paper.

References

[1] Alexander Hentschel, Dieter Shirley, and Layne Lafrance. Flow: Separating Consensus and Compute, 2019. https://arxiv.org/abs/1909.05821.

[2] Alexander Hentschel, Yahya Hassanzadeh-Nazarabadi, Ramtin Seraj, Dieter Shirley, and Layne Lafrance. Flow: Separating Consensus and Compute – Block Formation and Execution. 2020. https://arxiv.org/abs/2002.07403.

[3] Alexander Hentschel, Dieter Shirley, Layne Lafrance, and Maor Zamski. Flow: Separating Consensus and Compute – Execution Verification. 2019. https://arxiv.org/abs/1909.05832.

[4] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In *ASIACRYPT*, December 2001.

[5] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719, New York, NY, USA, 2015. ACM.

[6] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains, March 2017. Accessed:2017-10-06.

[7] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1353–1370, Baltimore, MD, August 2018. USENIX Association.

[8] Edward W. Felten. Cheater Checking: How attention challenges solve the verifier's dilemma, 2019. https://medium.com/offchainlabs/cheater-checking-how-attention-challenges-solve-the-verifiers-dilemma-681a92d9948e.

[9] Sanjit Chatterjee and Alfred Menezes. On cryptographic protocols employing asymmetric pairings — the role of ψ revisited. *Discrete Applied Mathematics*, 159(13):1311 – 1322, 2011.

[10] Feng Bao, Robert H. Deng, and Huafei Zhu. Variations of diffie-hellman problem. In *ICICS*, October 2003.

[11] Thomas Ristenpart and Scott Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In *EUROCRYPT*, May 2007.

[12] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *PKC*, January 2003.

[13] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. 2020. https://cryptobook.us.

[14] C. P. Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York.