WIP: Securing PoS Blockchains from Long-Range Attacks by Anchoring into Bitcoin using Taproot

Sarah Azouvi and Marko Vukolić
Protocol Labs

Abstract

Blockchain systems based on a reusable resource, such as proof-of-stake (PoS), provide weaker security guarantees than those based on proof-of-work. Specifically, they are vulnerable to long-range attacks, where an adversary can corrupt prior participants in order to rewrite the full history of the chain. To prevent this attack on a PoS chain, we propose a protocol that checkpoints the state of the PoS chain to a proof-of-work blockchain such as Bitcoin. We start by presenting a scheme that works for a PoS chain with a relatively small number of validators. We then present some WIP solutions that can potentially scale to tens of thousands of nodes. Since our work uses Schnorr threshold signatures, it could be of independent interest for other applications that require a large set of parties to produce a threshold signature using Bitcoin's Taproot.

1 Introduction

Long-range attacks (LRA) — also called posterior corruption attacks [12] — are one of the major security issues affecting permissionless proof-of-stake (PoS) blockchains. These attacks rely on the inability of a user who disconnects from the system at time t_1 and reconnects at a later time to tell that validators who were legitimate at time t_1 and left the system (by e.g., transferring their stake) are not to be trusted anymore. In a PoS system, where the creation of blocks is costless and timeless, these validators could create a fork that starts from the past, i.e., at time t_1, and runs until the present. This is unlike, for example, proof-of-work (PoW) systems, where creating blocks requires time and money (e.g., performing actual computation) and not just using cryptographic keys. A user would be unable to recognize the attack as they are presented with a "valid" chain fork. Because the past keys do not hold value in the present, previous validators can easily be bribed by an adversary intending on performing this attack.

Recently, Steinhoff et al. [29] proposed an approach to deal with LRA by anchoring the PoS membership into Ethereum's proof-of-work (PoW) blockchain (Eth 1.0), which is not vulnerable to this type of attack. The main idea of their work is to have a smart contract on the Ethereum blockchain that keeps track

of the state of the membership of the underlying PoS system. In a typical Byzantine Fault-Tolerant (BFT) protocol underlying PoS, the smart contract on Ethereum would only be updated if two thirds of the current staking power (or blockchain members in case of uniform voting rights) instruct the smart contract to do so. In the approach of Steinhoff et al. each validator will send a transaction to the smart contract that indicates a vote for a new set of validators. As soon as two thirds of the votes for the same set have been received, the smart contract automatically updates its state to the new set. From this moment on, the members of the new set are in charge of voting for the next set and so forth. Every user that needs to verify the set of validators can do so by simply checking the smart contract. An adversary cannot change the state of the smart contract, even with the keys of former validators. A LRA will never succeed, as any user can resort to the Ethereum smart contract to verify the correct state of the PoS chain. Unfortunately, as Ethereum is abandoning PoW, this approach is no longer viable. PoS of Eth 2.0 cannot be used instead of PoW for anchoring as it too suffers from the LRA vulnerability.

In this paper, we aim to design a solution to LRA, inspired by Steinhoff et al. [29], using Bitcoin's PoW, assuming this protocol will never change (there is no indication that PoS will be considered for Bitcoin in the foreseeable future). However, the implementation and design of such a scheme on Bitcoin is more challenging, compared to the implementation of Steinhoff et al. on Eth 1.0, because Bitcoin's expressivity is considerably more limited.

Besides, the approach designed by Steinhoff et al. leverages multi-signatures for anchoring, which can quickly bloat the transaction size, making it at worst impossible to anchor PoS networks with large number of validators, or, at best, very costly to do so. To address this constraint, our approach is to use the capabilities enabled by the recent Taproot upgrade to Bitcoin, which allows for more efficient Schnorr threshold signatures. Briefly, the protocol works as follows. As Bitcoin does not allow for stateful smart contracts, we will instead use an aggregated public key to represent the set of validators in the PoS system. When the set changes, the aggregated key must be updated in the Bitcoin blockchain. This is done by having a transaction transferring the funds associated with the aggregated key of the previous validators to the new aggregated key. Instead of having each validator send a transaction to the Bitcoin network, this transaction is signed interactively, off-chain, and all the signatures are aggregated into one constant-size signature. We note that since our work is based on Schnorr threshold signatures and uses Bitcoin's Taproot, it could be of independent interest to any project looking to implement large-scale threshold signing transactions on Bitcoin (for example, sidechains [3]).

In the rest of this paper, we present our high-level design in Section 3. Section 4 presents an informal security argument. In Section 6 we explain the limitations of this high-level design, notably when it comes to scalability (to large number of validators) and our work-in-progress (WIP) on improving it. Section 8 presents the implementation of the protocol (WIP). We discuss related work in Section 9.

2 Background

We use elliptic curve notation for the discrete logarithm problem. Suppose q is a large prime and G, J are generators of a subgroup of order q of an elliptic curve $\mathbb{E}$. We assume that $\mathbb{E}$ 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 $G = dJ$.

Let H, H_1, H_2 be cryptographic hash functions mapping to $\mathbb{Z}_q^*$. We denote by $x \xleftarrow{\$} S$ that x is uniformly randomly selected from S.

2.1 Schnorr signature

The Schnorr signing scheme works as follows. Let (s, Y) be a user key pair (such that $Y = sG$) and m a message to be signed. The signer performs the following steps.

1. $k \xleftarrow{\$} \mathbb{Z}_q$
2. $R \leftarrow kG$
3. $z \leftarrow k + H(m||R||Y) \cdot s \mod q$

The signature is then (z, R) and is verified by checking that $zG = R + H(m||R||Y)Y$.

2.2 Secret sharing schemes

A secret sharing scheme allows one participant (a dealer) to share a secret with n other participants, such that any t of them can recover the secret but any set of $t-1$ or less of them cannot. Furthermore, a desirable property of a secret sharing scheme is to be publicly verifiable, i.e., anyone should be able to verify that the dealer computed the correct shares and did not cheat. In this paper, we will use Feldman's verifiable secret sharing scheme [14] (VSS), which we describe in steps **1-3** of Figure 3.

2.2.1 Generating a secret

Unlike Feldman's VSS scheme, in which only one participant generates a secret and shares it with their peers, we consider a protocol where everyone contributes equally to generate a common secret, such that no set of participants of size strictly smaller than t can recover the secret on their own. We will use the scheme designed by Gennaro et al. [19] that we define in Figure 3, and we adopt the following notation:

$$(s_1, \cdots, s_n) \xleftarrow{(t,n)} (r|Y, a_k G, H_0),\ k \in \{1, \cdots, t-1\}$$

to mean that s_j is player j's share of the secret r for each $j \in H_0$. The values $a_k G$ are the public commitments used to verify the correctness of the shares

and (r, Y) forms a key pair where r is a private key and Y is the corresponding public key. The set H_0 denotes the set of players that have not been detected to be cheating during the execution of the protocol. This protocol is secure for any $t > 0.5n$.

2.3 Threshold signing

A t-of-n threshold signing scheme allows any combination of t participants to sign a message while preventing any coalition of $t-1$ participants or less to create a valid signature, i.e., at least t participants must agree to sign the message for the signature to be valid. We use the threshold signing protocol FROST [24], that we define in Figure 5. This interactive protocol will either output a Schnorr signature (z, R) on a message m or a message abort, together with a set of misbehaving participants such that the protocol can be rerun without those misbehaving participants in the next step. The protocol relies on a signature aggregator (SA). Note that if the SA is malicious, the protocol can abort without any output. We tackle aborts due to a malicious SA by implementing a timeout such that the SA is changed in case no output is produced. Furthermore, the SA could falsely report misbehavior by participants; however, this does not impact the overall security of the signing protocol. In other words, a malicious SA cannot forge a signature, but they can delay it.

We chose to use FROST because it is more efficient than alternative protocols, such as Stinson and Strobl [30] protocol, even though it is not robust, i.e., the protocol cannot complete if one participant aborts or misbehaves. However, misbehaving participants are detectable in FROST, so the protocol can simply be restarted from scratch without those malicious participants. Other Schnorr signing protocols [27, 13] were not considered as they are not compatible with threshold signing.

Note that we will not implement the key generation algorithm presented by Komlo and Goldberg [24], used originally in FROST, as it does not allow to detect misbehaving participants, therefore losing the ability to re-start the protocol without the misbehaving participants. Instead, we will use the scheme by Gennaro et al. [18] and borrow only the signing scheme presented in the FROST paper [24], as per the authors' suggestion. The distributed key generation (DKG) algorithm by Gennaro et al. is also used by Stinson and Strobl [30] and has the advantage of being robust (it will complete despite misbehaving participants). We follow the suggestion in Gennaro et al. [19] and use the simpler variant of the DKG, JF-DKG, as this is sufficient for our application of threshold signing.

The main reason for preferring an efficient but non-robust signing algorithm is that our protocol will eventually be incentivized (financial rewards will be given out to participants who perform the signature). Therefore, it is reasonable to expect participants to cooperate, especially when malicious behavior is detectable and can only delay — not prevent — the signing. Because both the DKG and the signing part of our protocol are modular, other threshold signing protocols can be used interchangeably for different threat models (e.g., including

the robust signing protocol in [30]).

2.4 PoS chain

We assume an underlying blockchain based on a reusable resource such as PoS that is secure, i.e., that satisfies the usual security properties of consistency, chain growth, and chain quality [16], as long as the adversary controls a bounded number of players at any time. Let f be the maximum fraction of power that the adversary can control while the protocol maintains its security (e.g., $f = 1/3$). For simplicity, we assume that this blockchain provides instant finality, i.e., that there are no forks. This can be achieved using some variant of a BFT-protocol [20, 7] or relaxed by using a "lookback" parameter. For example, if a block is final after k confirmations, then we will use the state of the chain k blocks in the past instead of the latest state to ensure consistent views across participants.

2.5 Taproot

Taproot is a recent Bitcoin network upgrade that allows for transactions to be signed using Schnorr signatures and that introduces a new data-structure, Merkelized Abstract Syntax Trees (MAST), for more advanced scripting in a privacy-preserving way. The main advantage of Schnorr signatures over the ECDSA multi-signature is that they enable signature aggregation, saving space in Bitcoin blocks while also providing more privacy as it is not possible to distinguish between a "regular" transaction, i.e., sending bitcoins from one person to another, and a more complex one, e.g., using a threshold signature. This could help hide identities in the blockchain and thwart clustering deanonymization.

A Taproot address has two components: a single public key (the internal key) and a script tree, identified by its Merkle root. Each component can be spent independently. In the case of threshold or multi-signatures, the internal key can be the aggregated public key of all the signers. The script tree can contain an arbitrary number of different scripts, each of which specify a condition that must be satisfied in order for the coins to be spendable. For example, one condition can be to give the pre-image of a hash. As the name suggests, in the script tree, the scripts are organized in a tree (see Figure 1). The transaction can be spent either by using the internal secret key (key path) or by satisfying one of the conditions in the tree (script path). In this paper, we are interested in spending a Taproot output using the key path, i.e., the internal key. It should be noted that it is possible to use the script tree to define a threshold signature scheme [26], though less efficient as the size of the tree would grow exponentially with the number of participants [3].

We now detail how to spend a Taproot output using the key path.

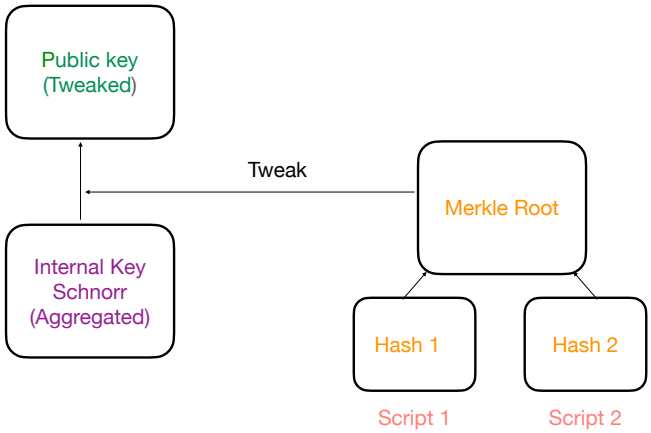

Figure 1: Taproot Output Composition

2.5.1 Key path spending

To prevent a potential vulnerability in which one user of a threshold or multi-signature could steal all the funds [5], the output key should commit to a (potentially unspendable) script path even if the spending condition does not require a script path (i.e., if only the key path is going to be used). There are multiple ways to achieve this with Taproot. The most natural way is to simply include the internal public key in the "tweak." The tweaked public key (i.e., outer key) is then computed as follows:

$$Q = P + \text{int}(H_{TapTweak}(bytes(P)))G$$

where P is the internal public key and $H_{TapTweak}$ is a hash function. The associated tweaked private key is then: $q = p + \text{int}(H_{TapTweak}(bytes(P)))$ where p is the private key associated with P. In order to spend the output using the key path, one must then sign the transaction with the tweaked private key.

Adding a commitment Alternatively, the script path could be used to add a commitment. For example in our case this commitment could be the hash of the underlying PoS chain at regular intervals. Let c denote this commitment. In this case, the tweaked public key becomes: $Q = P + H_{TapTweak}(P||c)G$ and the tweaked private key $q = p + H_{TapTweak}(P||c)$. The script path is still unspendable, and the output is spent by signing using the tweaked private key.

2.5.2 Transaction notation

For any Bitcoin transaction, we use the following notation: $\mathsf{input}_1, \ldots, \mathsf{input}_i \rightarrow ((\mathsf{amount}_1, \mathsf{output}_1), \ldots, (\mathsf{amount}_j, \mathsf{output}_j))$ to say that all the coins associated

with $\mathsf{input}_1 \ldots, \mathsf{input}_i$ are transferred to $\mathsf{output}_1, \ldots, \mathsf{output}_j$ with, respectively, $\mathsf{amount}_1, \ldots, \mathsf{amount}_j$. As a reminder, since Bitcoin is UTXO based, all the coins from an input must be transferred during the transaction, although to potentially multiple addresses. Additionally, it must be the case that $\mathsf{amount}_1 + \cdots + \mathsf{amount}_j \leq \mathsf{input}_1.\mathsf{amount} + \cdots + \mathsf{input}_i.\mathsf{amount}$ where $\mathsf{input}_k.\mathsf{amount}$ represents the total amount associated with input_k. The remaining amount (in the case of a strict inequality) is used as a transaction fee for the miner mining the block.

3 Protocol

We start by detailing our assumptions in Section 3.1 before presenting the high-level protocol for the main algorithm (Figure 2) as well as the subroutines for the DKG (Figure 3) and threshold signing (Figure 5). A pseudo-code version of the algorithm and its subroutines is provided in Algorithms 1, 2 and 3 in the appendix.

3.1 Assumptions

Each state of the PoS blockchain is associated with a set of participants, called the configuration and denoted by C, and their corresponding *power* (e.g., number of coins staked in the case of the PoS).We call the set of weighted participants in a configuration the *power table*. For simplicity, we consider a flat model, i.e., one participant accounts for one unit of power in the PoS blockchain. The flat model could be generalized by considering that one participant with x units of power possesses x public keys, one for each of their units of power. We will discuss how this assumption impacts the scalability of our protocol in Section 6. Furthermore, we assume that there is some similarity between successive configurations of the system, i.e., the set of participants does not change completely from one configuration to another. Formally, we define the difference between two configurations C_j and C_i as their symmetric difference ($C_i \triangle C_j$), which corresponds to the number of reconfiguration requests that need to be applied to C_i in order to obtain C_j. We assume that for two consecutive configurations C_i and C_{i+1}, their symmetric difference is bounded by some parameter b. We consider an adversary $\mathcal{A}$ that, for each state i of the PoS system, controls a fraction of at most f participants in configuration C_i *as well as all the keys from previous configurations* $(C_j)_{j<i-L}$ where $L \gg 1$ is a parameter. Under this assumption, the adversary is able to mount a LRA as follows. The adversary start a fork of the PoS chain at height $j < i - L$, using the keys from configuration C_j and that runs until the current height i. Since the adversary does not hold the keys from configuration $i - L$ and above, this means that from this height, the configurations on the adversarial fork and on the honest chain must differ. Note that under this attack, any online validator is able to differentiate the correct chain from a chain created as part of a LRA (since they are not part of the configurations in the adversarial fork). Our protocol will ensure that any user

is also able to distinguish each chain even if they have been offline, by looking at the Bitcoin blockchain. We discuss the security properties that the protocol should achieve in Section 4. Briefly, an adversary should not be able to create a signature on an illegitimate transaction or prevent the rest of the participants from signing a transaction.

3.2 Overview

The intuition behind the protocol is as follows: each configuration C_i is associated with a Taproot public key Q_i that consists of an internal key, in this case an aggregate public key pk_i, that participants computed with an interactive DKG protocol (step **1** of the main algorithm protocol in Figure 2) and a tweaked part as defined in Section 2.5.1. We chose to tweak the internal key using a commitment to the PoS chain (i.e., the hash of the state of the PoS blockchain). Each player j in the configuration then knows a share of the secret key associated with pk_i, $s_{i,j}$, such that t_i of the shares are enough to compute a valid signature on any message, but fewer than t_i participants cannot compute a signature. Configuration C_i is responsible for anchoring the state of the PoS chain at this point in time in the Bitcoin blockchain, which also includes updating the new configuration. In order to do so, the new configuration C_{i+1} must first compute their aggregated public key pk_{i+1} using the DKG algorithm. This key is then tweaked using a commitment ckpt to the PoS chain (i.e., the hash of the PoS chain at that time). The tweaked key becomes $Q_{i+1} = pk_{i+1} + H_{TapTweak}(pk_{i+1}||\mathsf{ckpt})G$. Note that only the tweaked key will appear on the blockchain so the hash ckpt will not be visible by anyone looking at the blockchain without external knowledge. However, anyone who has access to pk_{i+1} and ckpt can easily reconstruct Q_{i+1} to verify that their view of the PoS chain is correct.

To update the configuration from C_i to C_{i+1}, a transaction from Q_i to Q_{i+1} must be included in the Bitcoin blockchain (steps **2** and **3** in Figure 2). Leveraging the recent Bitcoin Taproot upgrade (that allows for Schnorr signatures), the transaction needs to be signed by t_i participants from configuration C_i where t_i is chosen to be strictly more than $f|C_i|$ as this ensures that at least one honest participant signs, preventing an adversary from signing an illegitimate transaction. As discussed previously, we will use the FROST algorithm for signing. Note that, the DKG requires that $t_i > 0.5|C_i|$ to ensure security so our final constraint on t_i is $t_i > \max(0.5|C_i|, f|C_i|)$. Since we assume that online validators can distinguish a LRA chain, it is enough to have the transaction signed by t_i participants as no honest validators can be fooled into signing an illegitimate transaction. If forks were allowed in the case of an adversary with only f fraction of the power (i.e., outside of LRA forks), this would be more problematic, as two conflicting transactions could then be signed, and we would require at least two thirds of the participants to sign the transaction, for $f = 1/3$ (as previously mentioned, this can also be fixed by considering a block in the past, i.e., one that has been finalized).

In addition to the transfer of coins from Q_i to Q_{i+1}, the transaction spent

by configuration C_i will have a second output that does not receive any bitcoins and that is unspendable, but that contains an identifier *cid* used to retrieve the full details of the configuration. This is done using the OP_{RETURN} opcode of Bitcoin [6] that allows storing of extra information in the chain. This identifier will be useful in the case where a user does not have access to the right PoS chain (i.e., does not have the correct value for pk_{i+1} and c due to a LRA). In this case, the content identifier *cid* can be used, together with a content-addressable decentralized storage, for example IPFS [28] (or content-addressable storage implemented on PoS network validators) to retrieve the identities of the nodes in the correct configuration. The transaction updating the configuration will look as follows: $\mathsf{tx}_i : Q_i \rightarrow ((\mathsf{amount}, Q_{i+1}), (0, OP_{RETURN} = cid))$, meaning that amount is transferred to Q_{i+1} and 0 is transferred to $OP_{RETURN} = cid$ (unspendable output). This information is then publicly available. We discuss in Section 3.2.1 how any user can then use it to get the latest PoS configuration.

The high-level description of the protocol is presented in Figure 2 and the pseudocode in Algorithm 1, in the appendix. In the pseudocode, the notation $\langle msg \rangle_i$ means that message msg was sent by participant i. We use $\text{PM}(\langle msg \rangle, i)$ to denote that a private message msg was sent to participant i.

3.2.1 Verification

Once the protocol described above has been run by the participants, users of the PoS system who went offline for an extended period of time can use the Bitcoin blockchain to determine the correct configuration and state of the chain. Informally, the verification protocol works as follows: users, who are aware of the initial aggregated public key Q_0, which serves as an identifier of the PoS blockchain on the Bitcoin blockchain, can follow the chain of transactions from Q_0 to the newest public key Q_i. The latest transaction in the chain (i.e., from Q_{i-1} to Q_i) contains an additional output that corresponds to the content identifier of the configuration C_i. The user can then use this identifier to retrieve the configuration using IPFS. The high-level protocol is described below and the pseudocode is given in Algorithm 4 in the appendix.

1. Synchronize with the Bitcoin blockchain, e.g., running a Bitcoin full node[3].
2. Look for Q_0 and follow the chain of transactions to get tx_i and cid_i.
3. Use cid_i to get the list of current nodes from IPFS.
4. Request the PoS blockchain state from these nodes.
5. Verify that the aggregated public key on the PoS blockchain pk and the hash of the block ckpt are in accordance with the Bitcoin Taproot address Q.
6. If the checkpoint and aggregated key do not match the Bitcoin checkpoint, roll back the PoS chain until the previous checkpoint and go back to step 5.

[3]Bitcoin full nodes can be run on relatively cheap devices, e.g., Raspberry Pi.

We assume that the initial aggregated public key of participants (at genesis) pk_0 as well as their tweaked key Q_0 are trusted and known by everyone and that there are some coins associated with it (enough to pay for the transaction fees of several transactions). For each round $i > 0$:

1 The protocol starts after a threshold of new registrations and unregistrations has been monitored (e.g., since the last configuration, $i-1$, there has been u new registrations or unregistrations). We call this event U_i. We note X_i the height, in the PoS blockchain, corresponding to this event. As soon as the parties notice event U_i, they start the distributed key generation algorithm defined in Figure 3. This algorithm is performed by members of the **new configuration** in order to compute the new aggregated key pk_i. We denote $H_{i,0}$ the set of members in the new reconfiguration (i.e., reconfiguration i). (Every member knows who is part of the new configuration by property of the underlying PoS, using the power table). At the end of the algorithm, the aggregated public key pk_i is known by everyone and a message can be signed by t_i out of n_i of the participants using their secret share $s_{i,j}$.

$$(s_{i,1}, \cdots, s_{i,n}) \xleftrightarrow{(t_i,n_i)} (sk_i|pk_i, a_{i,k}G, H_{i,1}),\ k \in \{1, \cdots, t_i - 1\}$$

here $H_{i,1} = H_{i,0} \setminus \{\text{misbehaving participants from the protocol}\}$.

We assume that the DKG is finished by block $X_i + Y$ where Y is chosen conservatively. The tweaked public key of the taproot address is then defined to be $Q_i = pk_i + H_{TapTweak}(pk_i||\mathsf{ckpt})G$, where c is the hash of the block corresponding to U_i.

2 Signing protocol:

(a) Leader election: we note L the leader of the block in the PoS blockchain corresponding to event U_i (i.e., the block where the u^{th} registration event happened). Sarah: here is the leader is elected for efficiency only (i.e., we do not need all 3000 miners pushing the data to ipfs etc, however this is in theory be done by every miner

(b) If $i > 1$: L checks that the previous reconfiguration transaction tx_{i-1} (according to the PoS blockchain) is included in the bitcoin blockchain. If not, they submit it before forming the new transaction.

(c) L first publishes the list of members in the new configuration to IPFS and retrieves the corresponding content identifiers cid_i.

(d) Every participant computes the transaction tx_i as follows. They locally[1] compute cid_i and verify that the configuration associated with cid_i is on IPFS. If not, they publish it. (The data will not be duplicated.[2]) All of the coins associated with Q_{i-1} are transferred to Q_i and another output that receives no coins but contains an OP_{RETURN} that contains cid_i is added: $\mathsf{tx}_i : Q_{i-1} \to ((\mathsf{amt}, Q_i), (0, OP_{RETURN} = cid_i))$ where amount is the amount associated with Q_i minus transaction fees.

(e) The members of the **current configuration** (i.e. associated with pk_{i-1}) perform the interactive signing algorithm.

i. Set $j \leftarrow 1$.

ii. $(o, H_{i,j+1}) \leftarrow \mathsf{SchnorrThresholdSign}(H_{i,j}, \mathsf{tx}_i, pk_i, Q_i)$ defined in Figure 5 where $H_{i,j+1}$ is the set of non-misbehaving parties during the execution of the protocol.

iii. If $o = (z, R)$, i.e., a signature has been successfully produced, continue to step **3**.

iv. Else (i.e., $o = \mathsf{abort}$) set $j = j + 1$ and go to step **2**(e)ii.

3 The taproot signature is then computed as $(z', R) \leftarrow (z + H(\mathsf{tx}_i||R||Q_i)H(pk_i||\mathsf{ckpt}), R)$, where c is the hash of the PoS blockchain at height X_i.

4 L sends tx_i to the blockchain to update the configuration.

5 If by block $X_i + Z$ (where Z is chosen conservatively), no transaction tx_i has been signed, a new leader, leader of block $X_i + 1$ takes on the role of leader and goes back to step **2**. If the transaction has been signed, participants set $i \leftarrow i+1$ and go back to step **1**.

Figure 2: Main Algorithm

Each participant P_i performs the following steps, where t is a parameter and n is the total number of participants:

1. Choose $r_i \xleftarrow{\$} \mathbb{Z}_q$. Let the sharing polynomial be $f_i(u) = \sum_{k=0}^{t-1} a_{ik} u^k$ where $a_{i0} = r_i$. Compute $s_i^j = f_i(j) \mod q$ for each $j \in \{1, \dots n\}$ and send s_i^j privately to P_j.
2. Expose $Y_i = r_i G$ as follows. Broadcast $A_{ik} = a_{ik} G$ for $k \in \{0, \cdots, t-1\}$.
3. Verify the values broadcast by other players: $f_j(i)G \stackrel{?}{=} \sum_{k=0}^{t-1} i^k A_{jk}$. If the check fails for an index j, complain against P_j.
4. Answer each complaint from party P_j by broadcasting s_i^j.
5. If any of the revealed shares fails this equation, remove that participant from the set of players H_0.
6. Extract $Y = \sum_{j \in H_0} r_j G$, of which each player's share of the secret is $s_i = \sum_{j \in H_0} s_j^i$. The secret $r = \sum_{j \in H_0} r_j \mod q$ is never computed.

The corresponding aggregated private and public keys are (r, Y), denoted by

$$(s_1, \cdots, s_n) \xleftrightarrow{(t,n)} (r|Y, a_k G, H_0),\ k \in \{1, \cdots, t-1\}$$

Figure 3: Distributed Key Generation Algorithm (JF-DKG by Gennaro et al. [19])

4 Security argument (WIP)

In this section we present the informal argument for why our protocol is secure. We need to prove two things: (1) that any checkpoint pushed onto the Bitcoin blockchain is *correct*, i.e., that it corresponds to the valid state of the PoS; (2) that checkpoints will be pushed regularly. These two properties correspond, loosely, to the safety and liveness properties of our scheme.

4.1 Safety

The intuitive reason why any checkpoint pushed to the Bitcoin blockchain is correct is as follows. Note that, in this case, controlling the keys of past configurations does not help an attack as, when an output has been spent in Bitcoin, it cannot be spent again. Past keys are therefore useless in the Bitcoin blockchain. Since an adversary can control at most $t-1$ participants in configuration C; by the security properties of the DKG and FROST, an adversary cannot create a signature on its own, so at least one honest participant must sign the transaction. The security property of the underlying PoS ensures that all honest and online validators see the same state and, thus, agree on the checkpoint to be signed (LRA affects only users who have been disconnected for an extended period of time). Hence, the checkpoint is correct.

SchnorrThresholdSign(H, m, Y, Q)
Input: H is the set of players, m the message.
Y is the aggregated public key. Each participant holds a share of the associated secret key. We note Q the tweaked key as defined in step **1** of Figure 2.
PreProcess: Each participant P_i performs the following steps. π is the number of signing operations that can be performed before doing another pre-process step.

1. Create an empty list L_i. For $1 \leq j \leq \pi$ do:
   (a) Sample single-use nonces $(d_{ij}, e_{ij}) \xleftarrow{\$} \mathbb{Z}_q^* \times \mathbb{Z}_q^*$.
   (b) Derive commitment shares $(D_{ij}, E_{ij}) = (d_{ij}G, e_{ij}G)$.
   (c) Append (D_{ij}, E_{ij}) to L_i. Store $((d_{ij}, D_{ij}), (e_{ij}, E_{ij}))$ for later use in signing operations.
2. Publish (i, L_i) to the PoS blockchain.

Sign(m)
Each participant P_i does the following:

1. Compute S, the set of t participants for signing using RB_i (e.g., compute $H(id||\mathrm{RB}_i)$ and the smallest t hashes are the id selected). Sarah: For the implementation we can start by taking the first t participants ordered by indexes.
2. Fetch the next available commitment for each participant $P_i \in S$ from L_i and construct $B = \langle (i, D_i, E_i) \rangle_{i \in S}$.
3. Compute the set of binding values $\rho_l = H_1(l, m, B), l \in S$ and derives the group commitment $R = \sum_{l \in S}(D_l + \rho_l E_l)$ and the challenge $c = H_2(m||R||Q)$.
4. Each $P_i \in S$ computes their response using their secret share s_i by computing $z_i = d_i + (e_i \cdot \rho_i) + \lambda_i \cdot s_i \cdot c$ using S to determine the i^{th} Lagrange coefficient λ_i as follows: if $S = \{P_1 \cdots, P_\alpha\}$ represents the participants identifiers then $\lambda_i = \prod_{j=1, j \neq i}^{\alpha} \frac{p_j}{p_j - p_i}$.
5. Each P_i securely deletes (d_i, e_i) from their local storage and then post z_i to the PoS chain.
6. After all the shares from participants in S are included in the PoS chain, each participant performs the following steps:
   (a) Derive $R = \sum_{i \in S} R_i$ and $c = H_2(m||R||Q)$.
   (b) Verify that $z_i G \stackrel{?}{=} R_i + (c \cdot \lambda_i) \cdot Y_i$ for each signing share $z_i, i \in S$. If it fails, report the misbehaving participant(s) by publishing a message on the PoS blockchain with the proof of misbehaviour(s) (i.e., $z_i G$ and $R_i + (c \cdot \lambda_i) Y_i$ for each cheating player) and abort.
   (c) If no participants was misbehaving, compute $z = \sum_{i \in S} z_i$.
   (d) Compute $\sigma = (z, R)$ to the PoS blockchain.

Output: (σ, H) if the protocol completed, (abort, S') else, where S' is the set of players who have not misbehaved during the execution of the protocol.

Figure 4: Signing Algorithm

SchnorrThresholdSign(H, m, Y, Q)
Input: H is the set of players, m the message.
Y is the aggregated public key. Each participant holds a share of the associated secret key. We note Q the tweaked key as defined in step **1** of Figure 2.
PreProcess: Each participant P_i performs the following steps. π is the number of signing operations that can be performed before doing another pre-process step.

1. Create an empty list L_i. For $1 \leq j \leq \pi$ do:
   (a) Sample single-use nonces $(d_{ij}, e_{ij}) \xleftarrow{\$} \mathbb{Z}_q^* \times \mathbb{Z}_q^*$.
   (b) Derive commitment shares $(D_{ij}, E_{ij}) = (d_{ij}G, e_{ij}G)$.
   (c) Append (D_{ij}, E_{ij}) to L_i. Store $((d_{ij}, D_{ij}), (e_{ij}, E_{ij}))$ for later use in signing operations.
2. Publish (i, L_i) to the PoS blockchain.

Sign(m)
Each participant P_i does the following:

1. Compute S, the set of t participants for signing using RB_i (e.g., compute $H(id||\mathrm{RB}_i)$ and the smallest t hashes are the id selected). Sarah: For the implementation we can start by taking the first t participants ordered by indexes.
2. Fetch the next available commitment for each participant $P_i \in S$ from L_i and construct $B = \langle (i, D_i, E_i) \rangle_{i \in S}$.
3. Compute the set of binding values $\rho_l = H_1(l, m, B), l \in S$ and derives the group commitment $R = \sum_{l \in S}(D_l + \rho_l E_l)$ and the challenge $c = H_2(m||R||Q)$.
4. Each $P_i \in S$ computes their response using their secret share s_i by computing $z_i = d_i + (e_i \cdot \rho_i) + \lambda_i \cdot s_i \cdot c$ using S to determine the i^{th} Lagrange coefficient λ_i as follows: if $S = \{P_1 \cdots, P_\alpha\}$ represents the participants identifiers then $\lambda_i = \prod_{j=1, j \neq i}^{\alpha} \frac{p_j}{p_j - p_i}$.
5. Each P_i securely deletes (d_i, e_i) from their local storage and then post z_i to the PoS chain.
6. After all the shares from participants in S are included in the PoS chain, each participant performs the following steps:
   (a) Derive $R = \sum_{i \in S} R_i$ and $c = H_2(m||R||Q)$.
   (b) Verify that $z_i G \stackrel{?}{=} R_i + (c \cdot \lambda_i) \cdot Y_i$ for each signing share $z_i, i \in S$. If it fails, report the misbehaving participant(s) by publishing a message on the PoS blockchain with the proof of misbehaviour(s) (i.e., $z_i G$ and $R_i + (c \cdot \lambda_i) Y_i$ for each cheating player) and abort.
   (c) If no participants was misbehaving, compute $z = \sum_{i \in S} z_i$.
   (d) Compute $\sigma = (z, R)$ to the PoS blockchain.

Output: (σ, H) if the protocol completed, (abort, S') else, where S' is the set of players who have not misbehaved during the execution of the protocol.

Figure 5: Signing Algorithm

4.2 Liveness

The intuition behind why an adversary cannot stop the signing from going ahead and the checkpoints from happening is as follows. (1) The robustness of the DKG ensures that an adversary cannot stop the rest of the players from computing an aggregated public key. (2) The adversary could delay the signing process by aborting; however, in the case where the adversary is not the SA, aborting or misbehaving players will be detected and excluded from the signing in the next iteration. In the case where the adversary is the SA, thanks to the time-out we implemented, a new leader should be elected shortly after, and the signing can resume. (3) The assumption about the stability across configurations ensures that enough honest participants will be able to perform the signing, i.e., we assume that enough participants from each configuration will remain available in the system long enough to sign and give the signer power to the next configuration.

Note that there is one attack that an adversary could mount against the liveness of the system: a denial-of-service against the SA to prevent them from coordinating with the rest of the players. In a later version of the paper we will automate the role of the SA to thwart this attack, so that participants post their share of the signature directly to the PoS chain without requiring any instructions or coordination.

Beside coordinating the participants, which could be automated by leveraging the underlying PoS, the only role that the SA practically has in the FROST protocol is to decide on the set of participants to partake in the signing. This, too, could be automated by choosing the set of participants in a deterministic or pseudo-random way, using some randomness coming from the chain (random numbers are often created as part of a PoS protocol as they are needed for leader election). The participants could then independently sign the message without the coordination from the SA, and liveness would not be vulnerable.

5 Dealing with a non-flat model

In Section 3, we assume that each participant holds the same amount of power (flat model). One way to deal with a non-flat model is to have every participant holds as many keys as the unit of power they possess. In some cases, such as proof-of-stake, the minimal unit of stake is easy to compute. In other cases such as proof-of-space, it is not straightforward and a minimum unit of storage may need to be arbitrarily decided, even if it means losing some precision in the number of keys held. Let's work out a concrete example. In Filecoin there is currently 13.987 EiB of storage.[4]. The biggest miner (as of December, 7th) owns 147.66 PiB or 1.03% of the total power. There exist 3606 miners in total, with the smallest miners owning 32 GiB (the smallest possible amount of storage that can be committed Sarah: not 100% sure about this). However, many of the small miners have so little relative storage that they practically never mine

[4] `https://filfox.info/en`

so discounting them from the protocol do not entail a big security risk. In order to find a small miner that has mined a block in the last 24 hours, we need to go up to number 3429, which has 16 Tib of power. Let's assume that this is the smallest unit of power that has some significance. This means that there would be roughly 1 million keys distributed across 3500 participants. It also follows that the bigger miner would, in this setting, hold more than 9000 keys. In practice there is a loss of precision that would occur as not every miner holds a power that is a multiple of 16 TiB (or whatever unit we decide), and hence some of the power will not be accounted for. Using a snapshot of the power table and a small Python script, we found that with using 16 TiB as the minimal unit associated with a keys, we will end up with 998748 keys for a power table of total power around 15.981 Eib. The expected number in an ideal world (i.e. everyone has a multiple of 16 TiB shares) would be 998766 shares. The loss of precision is thus less than 0.02%. For a bigger minimal unit (e.g., 160TiB), the loss of precision is 99798 shares vs 99883 expected shares, i.e., 1.6% loss. We discuss how theses keys could be aggregated efficiently.

Aggregating shares

To improve the scalability of our protocol in the non-flat model, we can aggregate the shares associated with one participant, such that the complexity of our protocol is quadratic in the number of players and not the number of shares (i.e. moving from a flat to non-flat model should not increase significantly the complexity of the algorithm). When aggregating shares, however, we must ensure that the aggregation does not dilute any information in a way that could be exploited by an adversary, for example in order to create the wrong shares for its secret. Futhermore, no information that allows a subset of t keys to recover a secret should be leaked in the process.

To see how shares can be aggregated, let's consider two participants, i and j that have, respectively l and m keys each. We start by considering the DKG defined in Figure 3.

Share creation

Player i should create l secret: $r_{i1}, \cdots, r_{il}$ to share with the other players. According to step 1 of the DKG, participant i should create a sharing polynomial for each of its secret: $f_{i1}(u) = \sum_{k=0}^{t-1} a_{i1,k}u^k$ where $a_{i1,0} = r_{i1}$, to share r_{i1}, $f_{i2}(u) = \sum_{k=0}^{t-1} a_{i2,k}u^k$ where $a_{i2,0} = r_{i2}$, to share r_{i2}, etc. As a first simplification, we notice that, besides the first coefficient, the other coefficients in every polynomial need not be different. Sarah: need to verify that this does not leak too much information. I think we can prove this, to do later. This means that every polynomials will have the following form: $f_{i_h}(u) = r_{i_h} + \sum_{k=1}^{t-1} a_{i,k}u^k$ for $h \in \{1, \cdots, l\}$. This reduces the number of random numbers to be generated and stored from $t \times l$ to $t - 1 + l$. Sarah: assume we have $t-1$ shares for each of the l polynomials. To recover the secret you would need to solve a system of $l \times (t-1)$ equations of degrees $t-1$ and $t-1+l$ unknown (too many solutions).

Player i must now send the shares $f_{i_1}(j_1), \cdots, f_{i_1}(j_m), \cdots, f_{i_l}(j_1), \cdots, f_{il}(jm)$ to participants j. Ultimately the share associated with the same index j_k will be added together, hence player i could simply send the sum of these shares straight away $s_i^{j_h} = (r_{i1} + \cdots + r_{il}) + l\sum_{k=1}^{t-1} a_{i,k} j_h^k$ for $1 \leq h \leq m$.

5.0.1 Commitment

Player i broadcast the following $A_{i,k} = a_{i,k}G$ for $1 \leq k \leq t-1$ and $A_{i_h,0} = r_{i_h}G$ for $1 \leq h \leq l$.

Share verification

Participant j verifies each share $s_i^{j_h}$ as follows: $s_i^{j_h}G = A_{i_1,0} + \cdots + A_{i_l,0} + l\sum_{k=1}^{t-1} A_{i,k} j_h^k$. This means they can batch the verification instead of verifying each share against l times. Sarah: check that player i cannot cheat by sending the wrong shares

Just as the original DKG, if any share does not verify correctly, participants will broadcast a complaint against that participant.

Key aggregation

The public key is computed as in Figure 3, i.e., $Y = \sum r_i G$ where for each participant i with l keys, $r_i = r_{i_1} + r_{i_l}$. Similarly, for each participant j their share of the secret are $s_{j_1} = \sum_{i \in H_0} s_{j_1}^i, \cdots, s_{j_m} = \sum_{i \in H_0} s_{j_m}^i$.

6 Scaling Schnorr threshold signing (WIP)

The main issue of the approach presented above is that it does not scale well. This is especially true if we consider a flat model where each unit of power corresponds to a different public key; we could easily end up dealing with tens of thousands of keys, even when the number of actual participants is much smaller, greatly increasing the latency of the protocol. Although some techniques such as sampling [9] or ad-hoc threshold multi-signature schemes [17] have been proposed to help scale weighted threshold signature schemes, those techniques are not currently compatible with Bitcoin's spending rules.

In the rest of this section, we discuss approaches to scale our protocol to tens of thousands of keys.

6.1 Distributed signing

To implement a threshold signature scheme with better scalability that retains compatibility with Bitcoin's spending rules, we propose parallelizing the signing. In practice, this means that each participant will be randomly assigned to a subgroup, with potentially many keys per participant. Each subgroup will then perform the DKG and threshold signing within their subgroup. Every subgroup performs these operations in parallel, greatly reducing the overhead for each

participant, as they do not need to communicate with participants who are not in their subgroup (and verify their shares or compute shares for them).

After each group has computed a signature, they will all need to be aggregated together to spend the transaction on the Bitcoin network. The question that arises then is how should the sub-signatures be aggregated together. For example, the DKG/signing schemes that we used in the previous section require interactions between the participants which would be unpractical when considering groups of participants instead of individuals and thus it is ruled out. Schemes like MuSig2 [27] require less interaction and could be viable; however, they are not compatible with threshold signing, which is problematic in our case since an adversary could potentially take control of one or more of the subgroups and prevent the signing from happening unless the subgroups are of size bigger than $f \times |C|$ (and the scheme becomes less scalable with subgroup size).

One straightforward solution is to use Bitcoin's native (i.e., pre-Taproot) multi-signature script that does not require any interaction between the participants. In this case all the public keys and necessary signatures are posted fully on-chain and are not aggregated in any efficient way. A limitation of this method is that, as space in Bitcoin's chain is scarce, only a small number of keys and signatures can be considered.

With this in mind, we propose the scheme outlined below, which requires an additional assumption: the existence of a random beacon $(\mathrm{RB}_i)_{i\in\mathbb{N}}$ that emits a new randomness for each state of the database (i.e., at each height of the underlying PoS blockchain). This is a standard assumption in PoS blockchains as a random beacon is necessary for the leader election part of the protocol. As as alternative, drand [1] is an operational external decentralized service that provides fresh randomness every 30 seconds. This service is used, for example, in the Filecoin consensus protocol [15]. We omit the initialization step where the initial output O_0 is created on the Bitcoin blockchain, and we assume it is associated with some coins.

1. We first pseudorandomly assign keys to N subgroups as follows. Let RB denote the random beacon at the state where the protocol starts. All participants compute $H(pk||\mathrm{RB})$ for every public key pk in the set of participants (with potentially many public keys per participant). The hash values are then arranged in ascending order and each key is assigned to a group depending on its position (i.e., the first n/N keys are assigned to group 1 and so forth).

2. Each subgroup independently performs a t-out-of-m DKG as defined in Figure 3 and tweaks their key using the hash of the chain as defined in Figure 2. Let $Q_1, \ldots, Q_N$ denote the associated tweaked public keys.

3. A new multi-signature output is created where only T out of the N signatures from $Q_1, \ldots, Q_N$ are required to spend the output, using Bitcoin's native script (i.e., all the public keys are posted on-chain). Let this output be O_i. The new transaction then resembles the one in the Section 3 and is defined as follows: $\mathsf{tx}_i : O_{i-1} \to ((\mathsf{amount}, O_i), (0, OP_{RETURN} = cid_i))$.

4. Each subgroup associated with O_{i-1} computes their FROST signature on tx_i.

5. A leader is elected to combine T out of N of the signatures from O_{i-1} in a script to be sent to the Bitcoin network.

6. The protocol starts again with the next configuration.

Parameter selection

The parameters t, T, m, N should be chosen to ensure that an adversary cannot unilaterally spend an output or prevent a signature from being created. The transaction needs to be signed by at least $t \times T$ players in order to be valid, therefore we need to have $t \times T > f|C|$ to ensure that the adversary cannot sign an illegitimate transaction. We also need $t \times T \leq (1-f)|C|$ to ensure that the honest participants can sign a transaction even if the adversary aborts. As an example, for $f = 1/3$, we can choose $N = 5$, $T = 3$ and $t = 55\%$.

Although this solution presents an improvement in terms of scalability compared to the previous protocol, it is still limited due to Bitcoin's multi-signature script size, i.e., it would not be practical to increase N beyond a certain bound. In the next section, we outline another, more promising, preliminary idea.

6.2 Non-interactive DKG

The main bottleneck of the protocol we described in Section 3 is the DKG, which requires many more interactions than the signing protocol (as already mentioned, the signing could be automated and shares posted on the PoS chain).

One solution to scale the protocol could, thus, be to construct a non-interactive DKG. This is, in theory, possible using SNARKs. Such a scheme exists, for example, in the case of BLS threshold signatures [21]. Although it would be non-trivial to extend this protocol to Schnorr signatures, another scheme could be designed by taking inspiration from works recently published in this area [4, 31].

7 Other approaches to scalable threshold signatures and their limitations

We present a few ideas that were investigated for scaling our protocol but that either did not work out or would require a significant investment of time (long-term research problems).

7.1 Mithril

Mithril [9] proposes a new scalable weighted threshold signature scheme. In Mithril, a signature is produced only if a sufficient number of *weighted* participants agree to sign a message. To do so, they define a new signing primitive and hence their scheme is not compatible with any signature scheme that already

exist. The protocol, similarly as multisignatures scheme such as MuSig2 [27], is non-interactive, each signer can independently produce a share of the signature. If enough shares are produced (i.e., above the threshold), they are aggregated and can be verified against the global key. Briefly, their protocol works as follows. The keys are organized in a Merkle tree and are associated with each participant's weight (in the case of PoS, this weight is the stake of that participant). Each participant samples their eligibility over m indices, where m is a parameter of the system. For each index, users are made eligible in proportion to their stake and independently of each other, i.e., each participants runs m lottery, each of which they can win with a probability proportional to their stake. This is done using a lottery such as the ones presented in Ouroboros Praos [10] or Algorand [20], that ensure that a participants cannot win more lottery if they spread their stake across multiple identities or keep it under one identity (keeping in mind that winning multiple lottery for the same index does not give any advantage as no more than one share per index is included in the final signature). Practically, the participants create m different signature shares, one for each index, and each share *wins* the lottery with probability proportional to the participant's stake. Producing an aggregate signature requires individual shares over k different indices. Once this happens, the shares can be aggregated efficiently using Bulletproofs [8]. Although this scheme achieves the goal of scalable weighted threshold signature that we pursue, it is not compatible with Bitcoin's spending rule, i.e., one could not spend a Bitcoin UTXO using this signature scheme and is hence not usable in our case (designing a similar scheme compatible with Bitcoin's spending rule is an open problem).

7.2 Sampling

The high-level idea behind Mithril is to elect a representative sample of the weighted set of participants and include their shares only. A similar idea could be adapted to our specific case of Schnorr signature. On a high-level, the protocol will work as in Section 3, except that only a subset of the miners in each configuration will be performing the DKG and the signing as explained below. This subset will be elected randomly, ensuring that an adversary cannot control more than some fraction of the elected subset.

1. Event U_i triggers the protocol.
2. There is a window X_1 for participants to check and publish their proof of eligibility (if eligible). Eligibility is determined using drand (or any other beacon used by the underlying PoS chain).
3. At the end of the window, the DKG starts. Only eligible participants partake in the DKG.
4. When the DKG completes, the participants associated with the previous aggregated key (i.e., eligible participants from the previous configuration) start the signing.

The issue with this approach is that the subset elected to perform the signing must be publicly known in advance (before performing the DKG) and hence a dynamic adversary could simply choose to corrupt the elected participants. This is unlike the protocol from Section 7.1 where the participants find out if they are elected in a lottery at the same time as they compute their signature share, at which point it is too late for an adversary to corrupt them. This solution would hence only work for a static adversary who can choose only once (at the beginning of the protocol) which players to corrupt. The only way to have a solution where only a subset of players are sampled to perform the signing and that is not vulnerable against a dynamic adversary is if the results of the election are only revealed at the time where each participants create their share of the signature (at which point it would be too late for an adversary to corrupt any of the elected players).

7.2.1 Private Sampling

One solution that was discussed[5] in order to implement some sort of privacy to the sampling protocol is to use ring signature. Ring signatures could help ensure that the set of participants elected is not publicly know. Each participant will know for themselves (and themselves only) whether they are eligible or not and hence they will participate in the DKG only if they are eligible, without revealing who they are. Intuitively this provides better guarantees than the public sampling protocol, as an adversary would not know who to corrupt. However, since the participants know their eligibility for themselves this solution is not fully satisfying. An adversary could, for example, bribe eligible participants using a smart contract or other means as was discussed by Deb et al [11].

7.3 BLS threshold or ECDSA threshold signing (Dfinity work)

7.3.1 BLS

Threshold signing is used in the core consensus algorithm of Dfinity's Internet Computer (ICP) [22, 21]. ICP uses a non-interactive DKG that allows participants to independently create their share of the public keys and publish it such that anyone is able to aggregate them and no interaction is needed between participants. This protocol is, however, based on BLS signatures which are not currently compatible with Bitcoin and it would be non-trivial to design a similar scheme for Schnorr signatures.

7.3.2 ECDSA

Dfinity is also working on creating an ECDSA threshold signature scheme.[6] ECDSA, unlike BLS, is compatible with Bitcoin and could thus be used to

[5] https://www.notion.so/protocollabs/Anonymized-DKG-3f2bd4949b9b4eaa926cbd04761295fa

[6] https://forum.dfinity.org/t/threshold-ecdsa-signatures/6152/30
https://forum.dfinity.org/t/threshold-ecdsa-signatures/6152/101

replace our threshold signing, if efficient enough.

7.4 Hierarchical signing

The solution presented in Section 6 where miners are divided into subgroups that each perform their own DKG and signing could potentially be extended further to two levels (or more) of subdivision. This means that the participants inside a subgroup could themselves be subdivided into smaller groups that would perform their own DKG and signing. The difficulty with this, as discussed in Section 6 is that the DKG algorithm for threshold signing requires much interaction and it is not realistic to have subgroups interact with subgroups (this would induce an important overhead).

Other ideas involving dividing the participants in subgroups were discussed but it is not clear whether they could work with an adversary with a significant portion of the power (e.g., 33%). We detail one idea below.

7.4.1 Subgroup + sampling

We propose the following protocol:

1. Miners are pseudorandomly assigned to different subgroup of same "weight" (with potentially more than one key per participant)
2. Each subgroup performs their own threshold DKG. We end up with N keys $pk_1, \cdots, pk_N$.
3. The output from the previous transaction can be spent by one (or more) of the aggregated keys as long as the signature is below a target, i.e., the output must be spend by $(pk_1\ OR\ pk_2\ OR \ldots pk_N)\ AND\ hash(sig) < target)$
4. Some proof-of-work-like mechanism could be implemented. If no signature is below the threshold, then we add a nonce to it: $hash(sig + nonce) < target$.
5. We don't know which subgroup wins before the signing is done (it's ok for two subgroups to win) so the adversary does not know who to corrupt in advance.
6. We could potentially ask for multiple signatures (i.e. more than one) to ensure that adversary cannot spend the output on its own.

We assume that $N > 3$ (otherwise the protocol would not improve scalability significantly) and that the adversary can control up to 1/3 of the power. Given that the subgroups are known in advance of signing, the adversary could easily decide to corrupt one full subgroup, or more. Then the adversary has a probability of at least $1/N$ of having its signature "winning" (i.e., being below the threshold) and hence could sign a illegitmate message. This seems like a weak security guarantee. NB: this is why in Section 6 we propose a protocol where

$N = 5$ and 3 signatures are needed: this ensures that an adversary cannot sign a transaction on its own, even when it can choose who to corrupt.

7.5 Adapt multisignature scheme to threshold signing

There exist multiple Schnorr multi-signature schemes that require a minimum level of interaction (e.g., MuSig2 [27]). In these schemes, there is no interactive DKG, the aggregated key is simply the sum of the keys of the participants. The only issue with multi-signature schemes is that, unlike, threshold signing, they require every participant associated with an aggregated key to sign, an adversary can thus stall the protocol for ever. One could wonder whether it is possible to construct a threshold scheme based on these multi-signature schemes by, for example, restarting the signing without the aborting/misbehaving participants. The issue would then be to ensure that misbehaviour is detectable. Another issue would be to compute the aggregated key. In this case, it would only be computable after the signing is done, i.e., once the set of signers is known, which is problematic for our use-case as the aggregated key must be known well ahead of the signing.

7.6 Vector commitments

Another idea which was discussed was to use vector commitments. Vector commitments can be used as a commitment to a set of public keys such that it is efficient to verify the membership of one keys to the set. This is unlike, for example, hash functions where one would need the entire set of keys to verify the membership of one keys to the committed set. The protocol would work as follows:

1. Previous configuration, C_{i-1} uses a multi-signature scheme to sign a transaction that includes a vector commitment VC_i that commits to the public keys of configuration C_i (using Bitcoin's OP_{RETURN}).
2. The transaction is signed using multi-signature and in the case where some participants do not participate, the public key is updated accordingly. With multi-signature, the aggregated key is just the sum of the keys of the participants and can thus be computed non-interactively and after the signing has happened (i.e., suming only the keys of the participants who did submit a correct share of the signature).
3. Anyone can verify that the keys associated with the multi-signature are indeed the correct keys of configuration C_i by checking their inclusion in the previous vector commitment VC_{i-1} and that there were more signers than the required threshold.
4. It is not clear how the aggregated public key can be funded (since we do not know it before the signing happened).
5. It is not clear how to link the different transactions together since each output is unspendable (due to OP_{RETURN}).

8 Implementation and Evaluation

The protocol from Section 3 is currently being implemented; some preliminary results will be ready to be presented by the end of the year.

9 Related Work

LRA have long been studied in the field of PoS and other types of checkpointing have been proposed that either rely on some sort of central authority [23] or on additional assumptions [2]. Like the solution from Steinhoff et al. [29], this paper offers a fully decentralized solution without additional security assumptions other than the ones needed for the security of the underlying PoS.

Kuznetsov and Tolkih propose an alternative solution to addressing long-range attacks in BFT/PoS [25], using forward-secure digital signatures. However, this solution is inapplicable in the rational adversary model, in which rational nodes might simply not follow the assumptions of forward-secure digital signatures, retaining their old private keys to mount attacks in the future.

Acknowledgment

The authors would like to thank Nicolas Gailly and Rosario Gennaro for helpful discussions on this project and Jorge Soares for helpful feedback.

References

[1] Drand documentation. `https://drand.love/`, Nov 2021.

[2] S. Azouvi, G. Danezis, and V. Nikolaenko. Winkle: Foiling long-range attacks in proof-of-stake systems. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 189–201, 2020.

[3] M. Bell. Proof-of-stake bitcoin sidechains. `https://gist.github.com/mappum/da11e37f4e90891642a52621594d03f6`, June 2021.

[4] E. Ben-Sasson, D. Carmon, S. Kopparty, and D. Levit. Elliptic Curve Fast Fourier Transform (ECFFT) Part I: Fast polynomial algorithms over all finite fields. *arXiv preprint arXiv:2107.08473*, 2021.

[5] Bitcoin. Bips/bip-0341.mediawiki at master · bitcoin/bips, Jul 2021.

[6] Bitcoin Wiki. OP_RETURN. `https://en.bitcoin.it/wiki/OP_RETURN`, June 2020.

[7] E. Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[8] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.

[9] P. Chaidos and A. Kiayias. Mithril: Stake-based threshold multisignatures. *Cryptology ePrint Archive*, 2021.

[10] B. David, P. Gaži, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 66–98. Springer, 2018.

[11] S. Deb, S. Kannan, and D. Tse. Posat: Proof-of-work availability and unpredictability, without the work. In *International Conference on Financial Cryptography and Data Security*, pages 104–128. Springer, 2021.

[12] E. Deirmentzoglou, G. Papakyriakopoulos, and C. Patsakis. A survey on long-range attacks for proof of stake protocols. *IEEE Access*, 7:28712–28725, 2019.

[13] M. Drijvers, K. Edalatnejad, B. Ford, E. Kiltz, J. Loss, G. Neven, and I. Stepanovs. On the security of two-round multi-signatures. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1084–1101. IEEE, 2019.

[14] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438. IEEE, 1987.

[15] Filecoin. Filecoin library drand. `https://spec.filecoin.io/#section-libraries.drand`, November 2021.

[16] J. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015.

[17] P. Gaži, A. Kiayias, and D. Zindros. Proof-of-stake sidechains. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 139–156. IEEE, 2019.

[18] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *Proceedings of the 17th International Conference on the Theory and Applications of Cryptographic Techniques*, pages 295–310. Springer, 1999.

[19] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.

[20] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

[21] J. Groth. Non-interactive distributed key generation and key resharing. *IACR Cryptol. ePrint Arch.*, 2021:339, 2021.

[22] T. Hanke, M. Movahedi, and D. Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

[23] S. King and S. Nadal. PPCoin: Peer-to-peer crypto-currency with proof-of-stake. `https://www.peercoin.net/whitepapers/peercoin-paper.pdf`, 2012.

[24] C. Komlo and I. Goldberg. FROST: Flexible Round-Optimized Schnorr Threshold Signatures. *IACR Cryptology ePrint Archive*, 2020:852, 2020.

[25] P. Kuznetsov and A. Tonkikh. Asynchronous reconfiguration with byzantine failures. In H. Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 27:1–27:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[26] Murch. 2-of-3 multisig inputs using Pay-to-Taproot. `https://murchandamus.medium.com/2-of-3-multisig-inputs-using-pay-to-taproot-d5faf2312ba3`, December 2020.

[27] J. Nick, T. Ruffing, and Y. Seurin. MuSig2: Simple two-round Schnorr multi-signatures. *Cryptology ePrint Archive*, 2020:1261, 2020.

[28] Protocol Labs. IPFS powers the distributed web. `https://ipfs.io/`.

[29] S. Steinhoff, C. Stathakopoulou, M. Pavlovic, and M. Vukolić. BMS: Secure decentralized reconfiguration for blockchain and BFT systems. *arXiv preprint arXiv:2109.03913*, 2021.

[30] D. R. Stinson and R. Strobl. Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates. In *6th Australasian Conference on Information Security and Privacy*, pages 417–434. Springer, 2001.

[31] The Electric Coin Company. Halo 2. `https://zcash.github.io/halo2/`, 2021.

A Pseudocode

Algorithm 1 Main algorithm

import PoS
import PoS.PowerTable as PT
import BTC
import PrivateMessage as PM
import IPFS
import Signing Algorithm (Algorithm 3), Distributed Key Generation Algorithm(Algorithm 2)
Parameters:
    id ▷ The node id
    u ▷ Tolerated difference between local configuration and current configuration in BMS
    f ▷ Fault tolerance of the current configuration
    Z ▷ Number of blocks after which a leader is re-elected to be the SA
    Y ▷ Number of blocks to wait for complaints to be broadcast
Init:
    $C_{cur} \leftarrow C_0$ ▷ Current configuration
    $C_{last} \leftarrow C_0$ ▷ Last configuration
    $pk_{cur} \leftarrow pk_0$ ▷ Initial public key
    $PendingRequests \leftarrow$ empty queue ▷ Queue of pending valid join and leave requests
    CurrentShares $\leftarrow$ empty dictionary ▷ Share of the aggregated public key
    $H_0 \leftarrow C_{cur}$ ▷ non-misbehaving participants
upon event receiving $PT.update(req) \wedge C_{last} \triangle C_{cur} < u$ **do**
    if $req = \langle p, \text{“}join\text{”} \rangle$ **then**
        $C_{cur}.members \leftarrow C_{cur}.members \cup \{p\}$
    if $req = \langle p, \text{“}leave\text{”} \rangle$ **then**
        $C_{cur}.members \leftarrow C_{cur}.members \setminus \{p\}$
upon event $C_{last} \triangle C_{cur} \geq u$ ▷ After u (un)registrations **do**
    $X \leftarrow$ PoS.CurrentBlock()
    Do Algorithm 2 (DKG)
upon event PoS.CurrentHeight == PoS.Height(X)+Y **do** ▷ Give enough time to receive and answer to complaints
    $H_0 \leftarrow C_{cur}.getIndexes() \setminus misbehavingPlayers$ ▷ set of indexes of non-cheating players
    $pk_{new} \leftarrow \sum_{j \in H_0} CurrentShares[j]$ ▷ Compute the aggregated key
    $c \leftarrow PoS.Blockhash(X)$
    $q \leftarrow pk_{new} + H_{TapTweak}(pk_{new}||c)G$ ▷ Taproot address
    $o \leftarrow 0$ ▷ Counter for the pre-process step
    $L \leftarrow PoS.BlockLeader(X)$ ▷ Check who is elected leader
    if $id == L$ **then**
        if BTC.latestCheckpoint.Output $\neq pk_{cur}$ **then** ▷ Check the Bitcoin blockchain for the previous transaction
            BTC.Broadcast(PoS.Read($\langle Checkpoint, tx \rangle$)) ▷ send latest checkpoint
        IPFS.push(C_{cur})
        $cid \leftarrow IPFS.getCid(C_{cur})$
        tx $\leftarrow BTC.TX(pk_{cur} \rightarrow (all, q), (0, OP_{RETURN} = cid))$ ▷ Compute the transaction
        PoS.Broadcats($\langle$Checkpoint, tx$\rangle$)
upon event PoS.Receive($\langle$Checkpoint, tx$\rangle$) **do**
    if $id \in C_{last}$ **then** ▷ Members associated with pk_{cur} vote
        do Algorithm 3 ▷ Signing protocol with other members
upon event PoS.currentBlockHeight()== $X + Z$ **do**
    $lastTx \leftarrow$PoS.Read($\langle$Checkpoint, tx$\rangle$) ▷ Get last transaction
    if $lastTx.Output \neq q$ **then** ▷ no new transaction received
        $X \leftarrow X + Z$
        go to line 34
    else
        $C_{last} \leftarrow C_{cur}$
        $pk_{cur} \leftarrow q$
        CurrentShares $\leftarrow \emptyset$

Algorithm 2 Distributed Key Generation

import MainAlgorithm
Parameters:
    $t \leftarrow 0.5|C_{cur}| + 1$ ▷ Number of parties controlled by the adversary
    $misbeahvingPlayers \leftarrow \emptyset$ ▷ Set of misbehaving participants
if $id \in C_{cur}$ **then** ▷ only member of the new configuration perform the DKG
    ▷ Assume participants are ordered in lexicographical order in C_{cur} and indexed in that order
    $i \leftarrow C_{cur}.members[id].getIndex()$ ▷ Nodes index
    $r_i \xleftarrow{\$} \mathbb{Z}_q$
    $a_{i0} \leftarrow r_i$
    for $k \in \{1, \ldots, t-1\}$ **do**
        $a_{ik} \xleftarrow{\$} \mathbb{Z}_q$
    $f_i(u) \leftarrow \sum_{k=0}^{t-1} a_{ik} u^k$
    for $j \in \{1, \ldots, |C_{cur}.members|\}$ **do**
        PM($\langle$SHARE, $s_j^i = f_i(j)\rangle, C_{cur}.members.index[j]$) ▷ send share of secret to each player
if id$\in C_{cur}$ **then**
    for $k \in \{0, \ldots, t-1\}$ **do**
        $A_{ik} \leftarrow a_{ik}G$
    PoS.Broadcast($\langle$secretCommitments, $A_{i0}, \ldots, A_{i(t-1)}\rangle_i$)
upon event PoS.Receive($\langle$secretCommitments, $A_{j0}, \ldots, A_{j(t-1)}\rangle_j$) **do**
    if id$\in C_{cur}$ and $s_i^j \neq \sum_{k=0}^{t-1} i^k A_{jk}$ **then**
        PoS.Broadcast($\langle$complaintSecret, j,$f_j(i)\rangle$)
    else
        CurrentShares.append(j, A_{j0})
upon event PoS.Receive($\langle$complaint, $j\rangle_k$) **do**
    $misbehavingPlayers.append(j)$ ▷ Keep count of complaints against j
upon event PoS.Receive($\langle$complaintAnswer, proof, $j\rangle_l$) **do** ▷ j can answer a complaint from l
    $s_l^j \leftarrow$Parse(proof)
    $(A_{jk})_{k=0}^{t-1} \leftarrow PoS.Read(\langle\text{FirstCommitments}\rangle_j)$ ▷ Get j commitments
    if $s_l^j G == \sum_{k=0}^{t-1} l^k A_{jk}$ **then**
        $misbehavingPlayers.remove(j)$
upon event PoS.Receive($\langle$complaint, $i\rangle_j$) **do** ▷ reply to complaints
    PoS.Broadcast($\langle$complaintAnswer, $s_j^i, i\rangle$)

Algorithm 3 Signing algorithm

import MainAlgorithm
Parameters: π ▷ Number of pre-process steps
$L_i \leftarrow \emptyset$
for $j \in \{0, \ldots, \pi\}$ **do**
    $(d_{ij}, e_{ij}) \xleftarrow{\$} \mathbb{Z}_q^* \times \mathbb{Z}_q^*$
    $(D_{ij}, E_{ij}) = (d_{ij}G, e_{ij}G)$
    $L_i.append(D_{ij}, E_{ij})$
PoS.Broadcast($\langle$PreProcess, $i, L_i\rangle$)
$B \leftarrow \emptyset$
$S \leftarrow C_{cur}.getIndexes()[: f|C_{last}| + 1]$ ▷ Choose t+1 of the participants for signing (for now choose the "first" $t + 1$ players). Will be randomized eventually.
for $i \in S$ **do**
    (D_{io}, E_{io}) ←PoS.Read($\langle$PreProcess, $i, L_i[o]\rangle$)
    $B.append((i, D_{io}, E_{io}))$
$cid \leftarrow IPFS.getCid(C_{cur})$
if $cid \notin IPFS.data$ **then** ▷ If the current configuration has not been sent to IPFS, the participant do it themselves
    $IPFS.Push(C_{cur})$
for $l \in S$ **do**
    $\mathsf{tx} \leftarrow BTC.TX(pk_{cur} \rightarrow (all, q), (0, OP_{RETURN} = cid))$ ▷ Compute the transaction
    $\rho_l \leftarrow H_1(l, m, B)$
    $\lambda_i \leftarrow \prod_{j \in S, j \neq i} \frac{p_j}{p_j - p_i}$ where p_j is the identifier of participant j
    $R \leftarrow \sum_{l \in S} D_l + \rho_l E_l$, $c \leftarrow H_2(R, Y, m)$
    $z_i \leftarrow d_i + (e_i \cdot \rho_i) + \lambda_i \cdot s_i \cdot c$
    delete $((d_{ij}, D_{ij}), (e_{ij}, E_{ij}))$ from local storage
    PoS.Broadcast($\langle$SHARE, $z_i\rangle$)
if $id \in S$ **then**
    upon event PoS.Receive($\langle$SHARE, $z_k\rangle$) from all $k \in S$ **do**
        CheatingPlayers← $\emptyset$ ▷ a new verification rule should be included in the PoS chain such that only valid complaint are included in the chain
        for $k \in S$ **do**
            $\rho_k \leftarrow H_1(k, m, B)$, $R_k \leftarrow D_{kj} + \rho_k E_{kj}$, $R \leftarrow \sum_{k \in S} R_k$, $c \leftarrow H_2(R, Y, m)$
            if $g^{z_k} \neq R_k + c \cdot \lambda_k \cdot Y_k$ **then**
                PoS.Broadcast($\langle k, g^{z_k}, R_k + c \cdot \lambda_k \cdot A_{k0}$, misbehaving $\rangle$)
                CheatingPlayers.append(k)
    if CheatingPlayers$\neq \emptyset$ **then**
        PoS.Broadcast($\langle$RESTART SIGNING, CheatingPlayers$\rangle$)
        restofplayers $\leftarrow C_{cur}.getIndexes \setminus S$
        $S \leftarrow S \backslash$CheatingPlayers
        S.append(restofplayers[:|CheatingPlayers|]) ▷ add as many players as were removed
        $o \leftarrow o + 1$
        go to 11
    else
        $z \leftarrow \sum_{i \in S} z_i$
        PoS.broadcast($\sigma = (R, z), S$)
        $c \leftarrow PoS.Blockhash(X)$ ▷ Commitment to the blockchain
        $\sigma' \leftarrow \sigma + H(\mathsf{tx}||R||q)H(pk_{cur}||c)$ ▷ compute taproot signature
        BTC.Broadcast(tx, σ')
        PoS.Broadcast(tx,σ')

Algorithm 4 Verification

import BTC
import IPFS
import PoS
Parameters: pk_0 ▷ Initial public key
$\mathsf{tx}_0 \leftarrow$ BTC.output(pk_0)
$i \leftarrow 0$
while output is unspent **do**
    output←BTC.getOutput(tx_i) ▷ Get chain of transactions
    $i \leftarrow i + 1$
cid ← output.OP_{RETURN}
$Q \leftarrow$ output.TaprootAddress
$members \leftarrow$ IPFS.getData(cid) ▷ Get the configuration from IPFS
for m in $members$ **do**
    PoS←query(m,PoS) ▷ get the latest PoS state from the current members
$c \leftarrow$ PoS.getLatetsCheckpoint ▷ verify checkpoint
$pk \leftarrow$ PoS.getLatestAggregatedKey
if $Q == pk + H_{Taproot}(pk||c)G$ **then** ▷ Verify that the state of the database is consistent with the Bitcoin checkpoint
    return 1
else
    PoS ← PoS.RemoveBlocks(after c) ▷ Roll back the PoS chain to the previous checkpoint
    $c \leftarrow$PoS.getLatetsCheckpoint
    $pk \leftarrow$ PoS.getLatestAggregatedKey
    Go to step 17