Conclave: A Collective Stake Pool Protocol

Dimitris Karakostas[1,3], Aggelos Kiayias[1,3], and Mario Larangeira[1,2]

[1] University of Edinburgh
[2] Tokyo Institute of Technology
[3] IOHK
dimitris.karakostas@ed.ac.uk,akiayias@inf.ed.ac.uk,mario@c.titech.ac.jp

Abstract. Proof-of-Stake (PoS) distributed ledgers are the most common alternative to Bitcoin's Proof-of-Work (PoW) paradigm, replacing the hardware dependency with stake, i.e., assets that a party controls. Similar to PoW's mining pools, PoS's stake pools, i.e., collaborative entities comprising of multiple stakeholders, allow a party to earn rewards more regularly, compared to participating on an individual basis. However, stake pools tend to increase centralization, since they are typically managed by a single party that acts on behalf of the pool's members. In this work we propose *Conclave*, a formal design of a *Collective Stake Pool*, i.e., a decentralized pool with no single point of authority. We formalize Conclave as an ideal functionality and implement it as a distributed protocol, based on standard cryptographic primitives. Among Conclave's building blocks is a weighted threshold signature scheme (WTSS); to that end, we define a WTSS ideal functionality — which might be of independent interest — and propose two constructions based on threshold ECDSA, which enable (1) fast trustless setup and (2) identifiable aborts.

1 Introduction

A major innovation of Bitcoin [39] was combining Proof-of-Work (PoW), to prevent sybil attacks, with financial rewards, to incentivize participation.

Regarding sybil resilience, Bitcoin's PoW depends on the collective network's ability to compute hashes. Thus, PoW limits each party's power and also determines how the distributed ledger is updated, i.e., which blocks can extend its blockchain. However, PoW's deficiencies, particularly its egregious environmental cost,[4] have driven research on alternative designs, most prominently Proof-of-Stake (PoS). PoS removes hardware requirements altogether and internalizes sybil resilience by relying on parties' *stake*, i.e., the assets that they own. These assets are managed by the distributed ledger and serve as both the system's internal currency and consensus participation tokens. PoS systems are almost energy-free, but often rely on complex cryptographic primitives, e.g., secure Multiparty Computation [33], Byzantine Agreement [12,23,34], or Verifiable Random Functions (VRFs) [13,23].

[4] The carbon footprint of: i) a single Bitcoin transaction is equivalent to $1,202,422$ VISA transactions; ii) the total Bitcoin network is comparable to Sweden. (`https://digiconomist.net/bitcoin-energy-consumption`; May 2021)

Regarding rewards, blockchain-based financial systems, like Bitcoin, aim to incentivize participation in the consensus mechanism. The rewards comprise of newly-issued assets and of transaction fees, i.e., assets paid by parties for using the system. Interestingly, both PoW and PoS ledgers are economies of scale, who favor parties with large amounts of participating power. One reason is poorly-designed incentives, resulting in disproportionate power accumulation [29,17]. Another is temporal discounting, i.e., the tendency to disfavor rare or delayed rewards [44]. Specifically, in Bitcoin, a party is rewarded for every block it produces, so parties with insignificant amounts of power are rarely rewarded. In contrast, accumulating the power of multiple small parties in "pools" yields a steadier reward. As a result, PoW systems see the formation of mining pools,[5] while PoS systems usually favor delegation to stake pools [10,28] over "pure" PoS, where parties act independently. Finally, the ledger's performance and security is often better under fewer participants. For instance, PoS systems require participants to be constantly online, since abstaining is a security hazard; this requirement is more easily guaranteed within a small set of dedicated delegates.

A major drawback of existing stake pools is that they are typically managed by a single party, the *operator*. This party participates in consensus, claims the rewards offered by the system, and then distributes them among the pool's members (after subtracting a fee). However, the operator is a single point of failure. In this work, we explore a more desirable design, which allows players to jointly form a *collective pool*, i.e., a conclave. This design assumes no single operator, minimizing excess fees, and trust and security concerns, altogether. Collective stake pools also promote a more fair and decentralized environment. In existing incentive schemes [4], operators who can pledge large amounts of stake to the pool are preferred. Consequently, the system favors a few major pool operators and, in the long run, its wealth is concentrated around them, resulting in a "rich get richer" situation. Although this problem is inherent in all decentralized financial systems [29], a well-designed collective pool may offset the stakeholder imbalance and slightly decelerate this tendency. Especially in PoS systems, a well-designed pool mechanism can prevent attacks observed on PoW [27,46,36].

Desiderata. Our design assumes a group of stakeholders who jointly create a stake pool without a single operator. Since large stakeholders typically form pools on their own, our protocol concerns smaller stakeholders, who could otherwise not participate directly. Therefore, our design could e.g., be appealing to a group of friends or colleagues, who aim for a more steady reward ratio without relying on a third party. Importantly, it should operate in a trustless environment as, unfortunately, even in these scenarios, trust is not a given. Notably, our targeted audience is parties who wish to actively participate, i.e., always be online to perform the required consensus actions; parties who wish to remain offline may instead opt for delegation schemes [10,28].

[5] 86% of Bitcoin's hashing power and 83% of Ethereum's hashing power are controlled by 5 entities each. (`https://miningpools.com`; May 2021)

In the absence of a central party, the responsibility of running the pool is shared among all pool's members, requiring some level of coordination which may be cumbersome. For instance, if the protocol requires unanimous actions, a single member could halt the pool's operation. To ensure good performance, the pool should allow a subset (of a carefully chosen size) to act on behalf of the whole group. The choice of such subsets depends on each party's "weight", which is in proportion to their stake. In summary, we have the following initial assumptions, which form the basis for outlining our work's desiderata:

- **small number of parties**: a collective pool is operated by a small group of players;
- **small stake disparity**: the profiles of the collective pool's members are similar, i.e., they contribute a similar amount of stake to the pool;
- **stake proportion as "weight"**: each party is assigned a weight for participating in the pool's actions, relative to their part of the pool's total stake.

Next, we provide an exhaustive list of basic requirements of a collective stake pool. We note that an *admissible party set* is a set of parties with enough stake, i.e., above a threshold of the total pool's stake which is agreed upon during the pool's initialization. To the extent that some desiderata are conflicting, our design will aim to satify as many requirements as possible:

- **Proportional Rewards**: the claim of each member on the entire pool's protocol rewards should be proportional to their individual contribution.
- **Joint Control of Rewards**: the members of a pool should jointly control the access to its funds.
- **Unilateral Reward Withdrawal**: at any point in time, a stakeholder should be able to claim their reward, accumulated up to that point, without necessarily interacting with other members of the pool.
- **Permissioned Access**: new users can join the pool following agreement by an admissible set of pool members.
- **Robustness against Aborting**: the pool should not fail to participate in consensus, unless an admissible set of members aborts or is corrupted.
- **Public Verifiability**: stake pool formation and operation should be publicly verifiable (s.t. consensus could take into account the aggregate pool's stake).
- **Stake Reallocation**: users should freely change their personal stake allocated to the pool, without interacting with other members of the pool.
- **Parameter Updates**: an admissible set of parties should be able to update the stake pool's parameters.
- **Force Removal**: an admissible set of parties should be able to remove a member from the pool.
- **Pool Closing**: an admissible set of parties should be able to permanently close the stake pool.
- **Prevention of Double Stake Allocation**: a party should not simultaneously commit the same stake to two different stake pools.

Our Contributions and Roadmap. We propose *Conclave*, a collectively managed stake pool protocol that aims to satisfy the listed desiderata. Our first

contribution is the ideal functionality $\mathcal{F}_{pool}$, a simulation-based security definition of collective stake pools, which captures the core security properties that our collective pool scheme should possess. We then describe π_{pool}, a distributed protocol executed by a set of n parties $\mathbb{P}$ which realizes $\mathcal{F}_{pool}$. π_{pool} employs certificates, which are published on the ledger, to announce its formation and closing. A major consideration and performance enhancement of our design is load balancing of transaction verification. Each transaction is verified by a (deterministically elected) committee of parties, whose size is a tradeoff between balancing workload, i.e., not requiring each party to verify every transaction, and reducing trust on the chosen validator(s). We thus construct a *distributed mempool*, i.e., a collectively managed set of unpublished transactions, s.t. if a majority of the committee's members are honest, transaction verification is secure. Our scheme uses a weighted threshold signature scheme (WTSS), to share the pool's key among its members, and a smart contract to manage the rewards. To that end, we provide a WTSS Universally Composable ideal functionality Section 3), which may be of independent interest, and construct an ECDSA WTSS, based on [21,22], s.t. each party has as many shares as "units" of weight. The scheme's complexity, which depends on the relative differences between each party's weight, is discussed in detail in Section 5.4.

Related Work. In the past years a multitude of PoS protocols have been proposed. The Ouroboros family [2,13,32,33] offers, like Bitcoin, eventual guarantees of liveness and persistence. Subselection has been employed in systems like Algorand [23], which employs Byzantine Agreement to achieve transaction finality in (expected) constant time, and Snow White [12,41], which uses the notion of "robustly reconfigurable consensus" to address potential lack of participation. Our work is complementary to these protocols and can be composed with them, as it is agnostic to the underlying PoS ledger's consensus mechanism.

Real-world PoS implementations often opt for stake representation and delegation. Systems like Cardano[6], EOS [10], and (to some extent) Tezos [25], employ different consensus protocols, but all enforce that a (relatively small) subset of representatives is elected to participate. Decred [14] takes a somewhat different approach, where stakeholders buy a ticket for participation, akin to PoS with optional participation. However, these systems typically assume single parties that act as delegates, either individually or as pool operators; our design directly aims at relaxing this restriction without requiring changes to the consensus protocol.

In cryptographic literature, stake pools are mostly treated from an engineering perspective. Ouroboros [33] offers a brief description of how delegation can be used within the protocol. This idea is expanded in [28], which provides a formal definition of PoS wallets and includes stake pool formation method via certificates. However, the pool's management is again centralized around the operator; our work extends this line of work by enabling the formation of a collective pool. Another work, orthogonal to ours, by Brünjes *et al.* [4] considers the incentives of distributing rewards among stake pools and aims to incentivize the creation

[6] `https://cardano.org`

of a (pre-defined) number of pools. However, it assumes that the pool operator commits part of their stake to make the pool more appealing, thus favoring larger pool operators. Our work eases such wealth concentration tendencies by enabling a collective pool to be equally competitive to a centralized one.

2 Preliminaries

2.1 The Execution Model of the Protocol

Our work is based on Canetti's Universal Composability (UC) formulation of the "hybrid world" [6], i.e., our protocol is defined with respect to auxiliary functionalities later to be described in Section 4.1. In this setting, briefly, the environment $\mathcal{Z}$ spawns an instance of an Interactive Turing Machine (ITM) which runs the defined protocol. Our system is locally polynomial-bounded, ensuring polynomial-time execution. The adversary $\mathcal{A}$ is also an ITM which may corrupt a number of parties on the environment's instructions.

We assume that the execution is organized in *time slots*. Each time slot corresponds to a single round, during which the parties are sequentially activated and exchange messages. The adversary is *adaptive*, i.e., can corrupt parties dynamically, and *rushing*, i.e., can decide its strategy after receiving, and possibly delaying, the honest parties' messages. At each round a party is elected to produce a block, and thus extend the distributed ledger, with probability proportional to its stake. Each block is a collection of valid transactions chosen by the elected party; a transaction is valid as long as it adheres to the ledger's rules as defined in its validation predicate.

In our setting, the ledger's maintainers are the stake pools. An adversary can break the ledger's properties by corrupting a set of stake pools which are allocated a majority of the total stake. Consequently, since we assume that the majority of stake is owned by honest stakeholders, the adversary needs to corrupt at least one pool to which honest players delegate. Especially in the collective setting, the adversary may control a subset of the pool's members, although not a majority of them. Therefore, our work considers adversaries who attempt to violate the security of a collectively-owned stake pool while controlling a minority of its members. Security violations include producing invalid blocks or transactions, as well as abstaining from join the consensus protocol.

2.2 Transactions, Blocks, and the Global Ledger

Our protocol utilizes two features of the Kachina [31] framework: first, the formalization of the ledger and, second, smart contracts.

The Simple Ledger Functionality. Our collective pool protocol interacts with a ledger functionality in a hybrid execution. A secure ledger, in the literature, is described as follows.

Definition 1. *A secure distributed ledger [19] satisfies the following properties:*

- ***Persistence****: A transaction which is part of a block at least k blocks away from the ledger's head, i.e., a block which is part of the chain which results from removing the last k blocks of the current chain, is stable, i.e., every honest party reports it in the same position in the ledger.*
- ***Liveness****: A transaction which is provided continuously as input to the parties is stable after u rounds.*

One option is Bitcoin's formalization from [3]. However, this functionality is local, while we would prefer a global functionality, following the Global UC Framework [8], hence we will use the $\overline{\mathcal{G}}_{simpleLedger}$ functionality from Kachina [31]. The functionality is available in Figure 1, where $\prec$ defines the prefix operation, i.e., $\Omega \prec \Omega'$ means the state Ω is included in Ω', and, for readability and consistency purposes, we rename *transaction* (τ) to *block* (b).

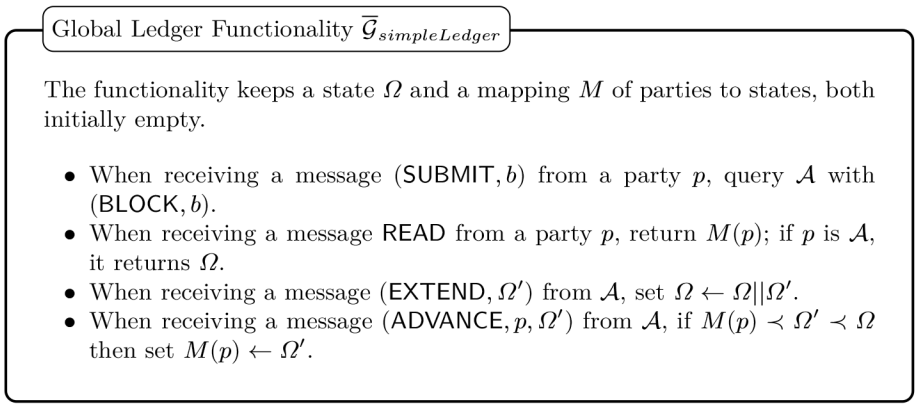

Fig. 1. The Simple Global Ledger Ideal Functionality.

The $\overline{\mathcal{G}}_{simpleLedger}$ Functionality is generic enough to abstract transactions and blocks, focusing on the ledger's properties. However, in our setting we need to define these objects, in order to better formulate a real-world blockchain. A transaction is the following object: $\tau = \langle \alpha_s, \alpha_r, v, f \rangle$, where i) $\alpha_s, \alpha_r \in \{0,1\}^*$ are the sender's and receiver's addresses respectively, ii) $v \in \mathbb{R}$ is the value transferred from α_s to α_r, and iii) $f \in \mathbb{R}$ is the fees of the transaction. A block consists of an ordered list of transactions. In order to organize transaction in blocks, we assume a function $\mathsf{blockify}$ which, given a set of transactions and a chain, returns a block which can extend the chain, i.e., satisfies the validity requirements of the system.

Reward Management via Smart Contracts. We also employ the formal model of smart contracts from [31]. This model considers smart contracts from a privacy-preserving perspective, which is out of the scope of this paper. However, it also provides a UC definition of standard smart contracts, consisting of the

universal machine $\mathcal{U}$, which acts as the oracle over the ledger's state, and the *core* contract Γ, as illustrated by Figure 3 adapted to our stake pool design. In our setting, the latter relates directly to the management of the rewards for the members of the pool, and therefore it is presented as an auxiliary (reward) functionality Γ_{reward} in Section 4.2.

2.3 Delegation and Stake Pools

The UC Model for delegated PoS systems was proposed in [28]. This framework formalizes the core of a PoS wallet, including operations like payment, delegation, and the formation of (centralized) stake pools. Specifically, each address corresponds to two keys, responsible for payments and staking. To perform an action, such as delegation or stake pool formation, a user creates a *certificate*, signed by the staking key, and publishes it on the ledger. Hence, any staking-related actions are public and parties can act accordingly when executing the consensus protocol. A user can re-delegate the stake of an address α in two ways: i) via a new delegation certificate signed by the staking key of α; ii) via a payment which moves the funds to an address α' with a different staking key, which issues a new delegation certificate. A user can also delegate its stake to multiple pools, by splitting its funds to multiple addresses, each with a different staking key. When a user moves funds from an address, its stake is automatically un-delegated, providing an easy way to withdraw from a stake pool.

To form a stake pool, a set of stakeholders pledge their stake to the pool. Following, the pool's *registration certificate* is signed by the stakeholders and the pool's managing key, which is managed solely by the pool's operator, and is published on the ledger. Among other parameters, the certificate defines the address which receives the rewards awarded for the pool's participation in consensus. Finally, the *revocation certificate*, also signed by the pool's managing key, is published on the ledger to halt the pool's operation.

This framework partially fulfills our earlier desiderata. In particular, *Prevention of Double Stake Allocation* and *Public Verifiabilitty* are addressed by the certificate-based registration and revocation mechanisms. However, the remaining items do not seem immediately solved without further assumptions. For instance, if the members have the same proportion of shares, a standard threshold signature scheme could address more of our desiderata, e.g., *Offline and Online Participation*, *Pool Proportional Rewards*, *Joint Control of Rewards*, and *Robustness against Aborting*. Following, reconfiguration of the pool is accomplished by regenerating the registration certificate and the pool's threshold key. Other desiderata can be approached in a similar fashion. Thus, our idea is to generalize the access structure of an efficient threshold signature scheme to add "weight" capabilities, such that the weights capture the pledged stake distribution among the pool's members.

2.4 Weighted Threshold Digital Signatures

Each pool member is given a share of the pool's private key, weighted by its stake. Every message produced by the pool, e.g., certificates or new blocks, is signed using a weighted threshold scheme. In a weighted threshold digital signature scheme [38,45], each player p is associated with a (integer) weight $\omega[p] \geq 0$, where ω is a mapping of players to weights. A signature can be produced by any set of keys, the aggregate weight of which is above the defined threshold. The weighted threshold signature scheme (Definition 4) is constructed by combining a digital signature scheme (Definition 2) with a weighted threshold secret sharing scheme (Definition 3). Additionally, standard threshold signatures is a special case of the weighted variant, with $\omega[p] = 1$ for every party p.

Definition 2 (Digital Signature). *For a security parameter λ, a digital signature scheme Σ is a tuple* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$*:*

- $\mathsf{Gen}(1^\lambda) \rightarrow (\mathsf{vk}, \mathsf{sk})$*: a randomized algorithm that, given the security parameter λ, outputs a pair of keys, the verification key* vk *and the λ-bit long private key* sk*;*
- $\mathsf{Sign}(\mathsf{m}, \mathsf{sk}) \rightarrow \sigma$*: (possibly) randomized algorithm that, given a message* m *and the private key* sk*, outputs a signature σ;*
- $\mathsf{Verify}(\mathsf{m}, \mathsf{vk}, \sigma) \rightarrow \{0, 1\}$*: a deterministic algorithm that, given a message* m*, a public key* vk*, and a signature σ outputs 1 if a signature is valid w.r.t. message m and verification key* vk *(respectively 0 if the signature is invalid).*

A digital signature scheme Σ is Existentially Unforgeable under Adaptive Chosen Message Attacks (EUF-CMA) if it presents the following properties:

Completeness: *For any message* m*, it holds:*

$$\Pr[(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda), \sigma \leftarrow \mathsf{Sign}(\mathsf{m}, \mathsf{sk}) : 0 \leftarrow \mathsf{Verify}(\mathsf{m}, \mathsf{vk}, \sigma)] \leq negl(\lambda)$$

where all the probabilities are computed over the random coins of the generation and sign algorithms.

Consistency: *For any message* m*, the probability that two independent executions of* $\mathsf{Verify}(\mathsf{m}, \mathsf{vk}, \sigma)$ *for a key pair* $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$*, output two different outcomes is smaller than $negl(\lambda)$.*

Unforgeability: *For any PPT algorithm $\mathcal{A}_{forger}$, which can query the signature oracle* $\mathsf{Sign}(\cdot, \mathsf{sk})$ *for signatures on a polynomial number of messages* m_i*, it holds:*

$$\begin{aligned}\Pr[(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda) : (\mathsf{m}, \sigma) \leftarrow \mathcal{A}_{forge}^{\mathsf{Sign}(\cdot, \mathsf{sk})} \wedge \mathsf{m} \neq \mathsf{m}_i \\ \wedge \mathsf{Verify}(\mathsf{m}, \mathsf{vk}, \sigma) = 1] < negl(\lambda)\end{aligned}$$

where all the probabilities are computed over the random coins of the generation algorithm and the adversary.

Definition 3 (Weighted Threshold Secret Sharing). *A (T, n, ω)-threshold secret sharing of a secret x consists of n shares $x_1, \ldots, x_n$, each associated with*

a weight $w_1, \ldots, w_n$, such that an efficient algorithm exists, that takes as input a set of shares B, with $\sum_{i \in B} w_i > T$, and outputs the secret value x. Any set of shares B with $\sum_{i \in B} w_i \leq T$ cannot obtain any information about the secret x.

Definition 4 (Weighted Threshold Signature). *Let Σ be a signature scheme. A (T, n, ω)-threshold signature scheme Σ_{thresh} for Σ is a triple of algorithms* $(\mathsf{Thresh\text{-}Key\text{-}Gen}, \mathsf{Thresh\text{-}Sign}, \mathsf{Thresh\text{-}Verify})$ *for the players $p_1, \ldots, p_n \in \mathcal{P}$ s.t.:*

- $\mathsf{Thresh\text{-}Key\text{-}Gen}(1^\lambda, \omega) \rightarrow (\mathsf{vk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n)$*: given the security parameter λ, outputs a public key* vk *and a list of private keys $\mathbb{K} = [\mathsf{sk}_1, \ldots, \mathsf{sk}_n]$ which form a (T, n, ω)-threshold secret sharing of* sk*; the pair* $(\mathsf{vk}, \mathsf{sk})$ *has the same distribution as the keys output by* Gen *of Definition 2;*
- $\mathsf{Thresh\text{-}Sign}(\mathsf{m}, B) \rightarrow \sigma$*: given a message* m *and a set of private keys B, $B \subseteq \mathbb{K}$, outputs a signature σ;*
- $\mathsf{Thresh\text{-}Verify}(\mathsf{m}, \mathsf{vk}, \sigma) \rightarrow \{0, 1\}$*: a deterministic algorithm that, given a message* m*, a public key* vk*, and a signature σ outputs 1 if a signature is valid w.r.t. message m and verification key* vk *(resp. 0 if the signature is invalid).*

A (T, n, ω)-threshold signature scheme Σ_{thresh} is ***EUF-CMA*** *if it presents the properties of Definition 2 and the following:*

Threshold Completeness: *For any message* m*, it holds:*

$$\Pr[(\mathsf{vk}, \mathbb{K}) \leftarrow \mathsf{Thresh\text{-}Key\text{-}Gen}(1^\lambda, \omega), \sigma \leftarrow \mathsf{Thresh\text{-}Sign}(\mathsf{m}, B), \sum_{k \in B} w_k > T : 0 \leftarrow \mathsf{Verify}(\mathsf{m}, \sigma, \mathsf{vk})] \leq negl(\lambda)$$

and

$$\Pr[(\mathsf{vk}, \mathbb{K}) \leftarrow \mathsf{Thresh\text{-}Key\text{-}Gen}(1^\lambda, \omega), \sigma \leftarrow \mathsf{Thresh\text{-}Sign}(\mathsf{m}, B), \sum_{k \in B} w_k \leq T : 1 \leftarrow \mathsf{Verify}(\mathsf{m}, \sigma, \mathsf{vk})] \leq negl(\lambda)$$

where all the probabilities are computed over the random coins of the key generation and sign algorithms.

2.5 Standalone Consensus

Apart from distributed ledgers, we consider standalone consensus where a set of parties need to reach agreement. To reach agreement on a transaction's validity, the committee members run a consensus sub-protocol. With hindsight, in Section 4.3 we use a consensus sub-protocol to increase performance by assuming a committee of pool members that verify each transaction.

Definition 5 (Consensus). *A consensus protocol $\pi_{consensus}$, which is performed among n processes p_i each with input v_i, satisfies [11]:*

- ***Termination:*** *eventually each correct process outputs a single value*
- ***Agreement:*** *all correct processes output the same value*
- ***Validity:*** *if all correct processes start $\pi_{consensus}$ with the same input v, then every correct process outputs v*

2.6 WTSS Building Blocks

The threshold signature scheme of [21] relies on ECDSA [18], as well as a Homomorphic Encryption Scheme, a Non-Malleable Equivocal Commitment Scheme, and a Multiplier to Addition (MtA) Share conversion Protocol, presented next.

Non-Malleable Equivocal Commitments A non-interactive trapdoor commitment scheme consists of four algorithms gen, com, ver, and equiv:

- $\mathsf{gen}(\lambda) \rightarrow (\mathsf{pk}, \mathsf{tk})$: On the input of the security parameter λ, it outputs the public key associated with the commitment scheme and the trapdoor.
- $\mathsf{com}(\mathsf{pk}, \mathsf{m}, r) \rightarrow [C(\mathsf{m}), D(\mathsf{m})]$: on input of the message, commitment key, and coin tosses r, it outputs the commitment string $C(\mathsf{m})$ and the secret decommitment string $D(\mathsf{m})$ for later opening.
- $\mathsf{ver}(C, D, \mathsf{pk}) \rightarrow \{\mathsf{m}, \perp\}$: On the input of the key and the commitment information, it outputs the original message or it fails to verify.
- $\mathsf{equiv}(\mathsf{pk}, \mathsf{m}, r, \mathsf{m}', T) \rightarrow D'$: On input of the values, respectively, for key, message and coin tosses, and a message $\mathsf{m} \neq \mathsf{m}'$ and a string T, if $T = \mathsf{tk}$, then $cver(C(\mathsf{m}), D', \mathsf{pk}) \rightarrow \mathsf{m}'$.

In practice, as pointed in [21], a hash function is used for a more efficient construction of the outlined commitment scheme; a thorough discussion of concurrent uses and non-malleability properties of commitments is provided in [21].

The Additively Homomorphic Encryption Let the encryption scheme $\mathcal{E}$ be is additively homomorphic module a large integer N. Therefore $E_{PK}(\cdot)$ is the encryption algorithm for public key PK. Given ciphertexts $c_1 = E_{PK}(a)$ and $c_2 = E_{PK}(b)$, there is an efficiently computable function $+_{\mathcal{E}}$ such that $c_1 +_{\mathcal{E}} c_2 = E_{PK}(a + b \mod N)$. This operation implies that an escalar as the following, assume $k \in \mathbb{N}$, $k \times_{\mathcal{E}} c = E(km \mod \mathbb{N})$. A concrete instantiation of such protocol is Pailllier's [40].

The Multiplier to Addition Conversion In [21], the authors introduced an adapted version for a protocol to convert multiplier shares to additive shares. Denote it the MtA protocol. More concretely, if two players respectively hold secrets $a, b \in \mathbb{Z}_q$, such that $x = ab \mod q$. They can convert the values a and b to multiplicative shares for x. That is, compute α and β, such that $x = \alpha + \beta$, such way that each party holds α and β.

The MtA protocol assumes that one of the parties have a public key for an additive homomorphic encryption scheme. Furthermore, regarding the share b, the value g^b may be public, therefore in some cases a checking should be done in order to assure the use of the correct value for b. The version of MtA with checks is denoted by MtAwc. We refer the reader to [21] for the full description of both MtA and MtAwc protocols.

3 UC Weighted Threshold Signature

In this section we present the weighted threshold signature ideal functionality $\mathcal{F}_{wtss}$ (Figure 2). This functionality is used in the Collective Pool Protocol π_{pool}, which employs weighted threshold signatures for collectively signing certificates and new blocks. The functionality $\mathcal{F}_{wtss}$ is inspired by Almansa *et al.* [1], which is in turn inspired by Canetti [7]. However, unlike Almansa *et al.* and similar to Canetti, during signature verification we consider the case of a corrupted signer, i.e., a set of parties such that the majority (of weights) is corrupted.

$\mathcal{F}_{wtss}$ interacts with a set of n parties. Each party P_i is associated with an integer w_i, i.e., its weight. $\mathcal{F}_{wtss}$ also keeps the following, initially empty, tables: i) pubkeys: tuples $\langle sid, vk \rangle$ of sid and a public key vk; ii) sigs: tuples (m, σ, vk, f) of message m, a signature σ, a public key vk, and a verification bit f. The mapping $\omega[p] \rightarrow w_p$ denotes the weight of a party p, while the term ω also denotes the set of keys the participating parties.

Weighted Threshold Signature Functionality $\mathcal{F}_{wtss}$

Each message is associated with $sid = \langle \mathcal{P}, \omega, T, sid' \rangle$, where $\mathcal{P}$ is the set of parties, ω is a mapping of parties to weights, T is the collective signature weight threshold, and sid' is a unique identifier.

Key Generation: Upon receiving (KeyGen, sid) from every honest party $P \in \mathcal{P}$, send (KeyGen, sid, P) to $\mathcal{S}$. Upon receiving a response $(\text{KeyGen}, sid, \mathsf{vk})$ from $\mathcal{S}$, record $\langle sid, \mathsf{vk} \rangle$ to pubkeys and send $(\text{KeyGen}, sid, \mathsf{vk})$ to every party in $\mathcal{P}$. Following, all messages that do not contain the established sid are ignored.

Signature Generation: Upon receiving (Sign, sid, m) from a party p, forward it to $\mathcal{S}$. After a subset of parties $\mathcal{P}' \subseteq \mathcal{P}$ has submitted a Sign message for the same m, and upon receiving $(\text{Sign}, sid, m, \sigma)$ from $\mathcal{S}$, check that $\sum_{p \in \mathcal{P}'} \omega[p] > T$ (*Note: This condition guarantees threshold completeness.*) Next, if $(m, \sigma, \mathsf{vk}, 0) \notin$ sigs (for the key vk that corresponds to sid in pubkeys), record $(m, \sigma, \mathsf{vk}, 1)$ to sigs and reply with $(\text{Sign}, sid, m, \sigma)$.

Signature Verification: Upon receiving $(\text{Verify}, sid, m, \sigma, \mathsf{vk}')$ from P, forward it to $\mathcal{S}$. Upon receiving $(\text{Verified}, sid, m, \sigma, \phi)$ from $\mathcal{S}$, set f as next:

1. If $\mathsf{vk}' = \mathsf{vk}$ and $(m, \sigma, \mathsf{vk}, 1) \in$ sigs, $f = 1$. (*This guarantees completeness.*)
2. Else, if $\mathsf{vk}' = \mathsf{vk}$, the aggregate weight of the corrupted parties in $\mathcal{P}$ is strictly less than T, and $(m, \sigma, \mathsf{vk}, 1) \notin$ sigs, $f = 0$ and record $(m, \sigma, \mathsf{vk}, 0)$ to sigs. (*This guarantees unforgeability, if the aggregate weight of the corrupted parties is below the threshold.*)
3. Else, if $(m, \sigma, \mathsf{vk}', b) \in$ sigs, $f = b$. (*This guarantees consistency.*)
4. Else, $f = \phi$ and record $(m, \sigma, \mathsf{vk}', f)$ to sigs.

Finally, send $(\text{Verified}, sid, m, \sigma, \mathsf{vk}', f)$ to P.

Fig. 2. Weighted Threshold Signature Ideal Functionality

As highlighted in the definition, *completeness, consistency*, and *unforgeability* are enforced upon verification, whereas *threshold completeness* is enforced upon signature generation. Hence, it should be infeasible to issue a signature unless using keys with enough weight, i.e., above the threshold, say, a value T.

4 The Collective Stake Pool

Our analysis is based on the UC Framework, following Canetti's formulation of the "real world" [5]. Specifically, we define the collective pool ideal functionality $\mathcal{F}_{pool}$, which distills the required (operational and security) properties; for readability, $\mathcal{F}_{pool}$ is divided in two parts, *management* and *consensus participation*. The ideal functionality is realized – in the "real world" – by the distributed protocol π_{pool}, which employs various established cryptographic primitives, and, therefore, π_{pool} can described with auxiliary functionalities. Before proceeding with the functionality's definition, we first describe the hybrid execution of π_{pool} and its building blocks.

4.1 Hybrid Protocol Execution

The protocol π_{pool} is performed by n parties, where each party p_i holds two pairs of keys: $(\mathsf{vk}_{p_i}, \mathsf{sk}_{p_i})$ for issuing transactions, and $(\mathsf{vk}_{s_i}, \mathsf{sk}_{s_i})$ for staking operations, e.g., issuing delegation certificates (cf. [28]). The public key vk_i is also used to generate an address α_i. Each pool member p_i pledges the funds of an address α_i (which it owns) to the pool. These funds are the player's stake in the pool and form the player's weight in the weight distribution mapping ω.

We assume the members' stake, i.e., their weight w_i in the pool, is public. Therefore, the weight distribution mapping ω is also public. Furthermore, each member of the pool has its own signature key, and can issue standard signatures through a standard signature scheme. A weighted version for a threshold signature scheme follows by having each party holding as many shares, of the original threshold scheme, as its weight. This approach has the extra advantage that security guarantees of the original scheme are carried straightforwardly into the weighted version. The full description of the WTSS Σ_{thresh} based on ECDSA is presented in Section 5.

Additionally, our construction relies on the consensus sub-protocol $\pi_{consensus}$ to validate a transaction by the elected committee. Specifically, the collective stake pool protocol is parameterized by: i) the validation predicate $\mathsf{Validate}$, ii) the permutation algorithm π_{perm}, and iii) a consensus sub-protocol $\pi_{consensus}$.

Finally, our (modular) protocol is described in a hybrid world whith auxiliary functionalities for established primitives. The functionality $\mathcal{F}_{BC}$ [26] provides a broadcast channel to all parties; $\mathcal{F}_{corewallet}$ [28] enables delegation to the pool; $\mathcal{F}_{wtss}$ (cf. Section 3) is used for weighted threshold signature operations; the Smart Contract Functionality Γ_{reward} realizes the reward distribution mechanism; $\overline{\mathcal{G}}_{simpleLedger}$ is a global Ledger Functionality [31]. Let $\mathsf{HYBRID}^{pool}_{\pi_{pool},\mathcal{A},\mathcal{Z}}$ denote the $\{\overline{\mathcal{G}}_{simpleLedger}, \mathcal{F}_{BC}, \mathcal{F}_{corewallet}, \mathcal{F}_{wtss}, \Gamma_{reward}\}$-hybrid execution of π_{pool} in the (global) UC Framework.

4.2 Part 1: Stake Pool Management

The functionality's first part (Figure 4) includes all operations that are not consensus-oriented. First, establishing a stake pool consists of two parts, defined as corresponding interfaces in the ideal functionality. The pool's members *gather* and jointly decide to create a staking pool; they contact each other, e.g., via off-chain direct channels, agree on the pool's parameters, and generate its key. Importantly, the participants are aware of the total number of participants in the pool, as well as their weights. Then, the members of the pool perform a setup protocol and *register* the new pool via a *registration certificate*, which is signed by the pool's key and published on the ledger. Following, the pool receives rewards for participating in the consensus protocol. The rewards are managed by a smart contract and, at any point, each each party can withdraw their part, which is proportional to the internal stake distribution. Finally, to close the pool, the members sign and publish a revocation certificate.

In more detail, the functionality $\mathcal{F}_{pool}$ interacts with n parties $p_1, \ldots, p_n$ and is parameterized by:

- the validation predicate $\mathsf{Validate}(\cdot, \cdot)$ which, given a transaction τ and a chain $\mathcal{C}$, defines whether τ can be appended to $\mathcal{C}$ (as part of a block);
- the algorithm $\mathsf{blockify}$ which, given a set of transactions, serializes them (deterministically) in a block;
- the probability $\Pi^{\theta,t,n}$ that the elected committee, responsible for a transaction's verification, is corrupted, dependent on the subselection parameter θ and the number of corrupted parties t out of n total parties.

It also keeps the (initially empty) variables: i) the signature threshold T; ii) the public key vk_{pool}; iii) the reward address α_{reward}; iv) the set of valid and unpublished transactions $\mathsf{mempool}$; v) a mapping of parties to weights W; vi) a table of signatures sigs.

Gathering and Registration. The first step in creating a pool is the gathering of parties, in order to collectively create the pool's public key vk_{pool}. Following, the parties create and publish on the ledger the registration certificate cert_{reg}, which contains the following:

- ω: a mapping identifying each member's weight;
- α_{reward}: the address which accumulates the pool's rewards;
- vk_{pool}: the pool's threshold public key;
- σ_{pool}: the signature of $\langle \omega, \alpha_{reward} \rangle$ created by vk_{pool}.

Reward Withdrawal. During the life cycle of the pool, a member may want to withdraw the rewards received up to that point. As per the desiderata of Section 1, any party should be able to do so, without the explicit permission of the other pool's members. Additionally, the rewards that each party receives should be proportional to its stake, i.e., its weight within the collective pool. Reward withdrawal is implemented as the smart contract functionality Γ_{reward}.

The contract is initialized with the weight distribution of the pool's members and each member's public key. We assume that the contract is associated with an address and can receive funds, similar to real-world smart contract systems like Ethereum [47]. The state transition functionality Γ_{reward} is defined in Figure 3(following the ledger formalization of Section 2.2).

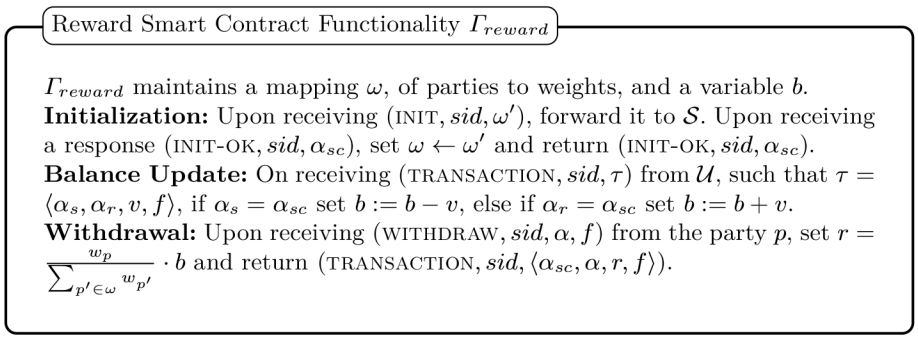

Fig. 3. The pool's Reward Smart Contract Functionality.

Closing. Eventually, the members halt the operation of the pool. In order to do so, they revoke the pool's registration by jointly producing a revocation certificate cert_{rev}. The certificate is relatively simple, containing a timestamp x announcing the end of the pool and signed by the pool's public key vk_{pool}.

The first part of our functionality definition is given by Figure 4, whereas the management routines, i.e., the first part of the description, of our protocol construction is given by Figure 5.

4.3 Part 2: Participation in Consensus

After a pool is set up, the functionality's second part (Figure 7) considers participation in the system, i.e., *validating transactions* and *issuing blocks*. The pool members continuously monitor the network for new transactions, which they collect, validate, and organize in a *mempool*. As mentioned in the introduction, the pool members *remain online* for the entirety of the execution to perform the pool's operations. Specifically, when the pool is elected to participate, the mempool's transactions are serialized and published in a block. Under PoS, the pool participates proportionally to its aggregated member and delegated stake.

To improve performance, we define a distributed mechanism for transaction verification, i.e., a *distributed mempool*. Such load balancing mechanism increases efficiency by requiring only a subset of the pool's members to verify each transaction. Notably, this is in contrast to the standard practice of Bitcoin mining pools, where the pool's operator decides the transactions to be mined by its members; instead, our approach further reduces these trust requirements.

Collective Pool Functionality $\mathcal{F}_{pool}^{T,\omega}$ (first part)

Gathering: Upon receiving (GATHER, sid) from p, forward it to $\mathcal{S}$. After every party $p_i, i \in [1,n]$ has submitted gather, upon receiving from $\mathcal{S}$ $(\text{GATHER-OK}, sid, \mathsf{vk}_{pool})$, store T and vk_{pool}, add all party-weight pairs (p_i, ω_i) to W, and reply with $(\text{GATHER-OK}, sid, \mathsf{vk}_{pool})$ to all parties.
Pool Registration: Upon receiving $(\text{REGISTER}, sid, W)$ from p, forward it to $\mathcal{S}$. After all parties $p_i, i \in [1,n]$ have submitted $\mathsf{register}$, upon receiving from $\mathcal{S}$ $(\text{REGISTER-OK}, sid, \alpha_{reward}, \sigma_{pool})$, set $\mathsf{cert}_{reg} = \langle (W, \alpha_{reward}, \mathsf{vk}_{pool}, \sigma_{pool}) \rangle$. Then check if $\forall (m, \sigma, b') \in \mathsf{sigs} : \sigma \neq \sigma_{pool}, (\mathsf{cert}_{reg}, \sigma_{pool}, 0) \notin \mathsf{sigs}$; if the checks hold, insert $(\mathsf{cert}_{reg}, \sigma_{pool}, 1)$ to sigs. Finally, store α_{reward} and reply with $(\text{REGISTER-OK}, sid, \mathsf{cert}_{reg})$.
Reward Withdrawal: Upon receiving the message $(\text{WITHDRAW}, sid, \alpha, f)$ from p_i, forward it to $\mathcal{S}$. Then, compute $r = \frac{w_{p_i}}{\sum_{j=1}^{n} w_{p_j}} \cdot r_{pool}$, where r_{pool} is the funds of address α_{sc} as defined in $\overline{\mathcal{G}}_{simpleLedger}$. Finally, return $(\text{TRANSACTION}, sid, \langle \alpha_{sc}, \alpha, r, f \rangle)$.
Closing: Upon receiving (CLOSE, sid, x) from p, forward it to $\mathcal{S}$. After a set of parties B has submitted close for the same x, if $\sum_{p \in B} w_p > T$, upon receiving $(\text{CLOSE-OK}, sid, \sigma_{pool})$ from $\mathcal{S}$, check if $\forall (m, \sigma, b') \in \mathsf{sigs} : \sigma \neq \sigma_{pool}, (x, \sigma_{pool}, 0) \notin \mathsf{sigs}$; if the checks hold, insert $(x, \sigma_{pool}, 1)$ to sigs. Finally, return to all parties $(\text{CLOSE-OK}, sid, \mathsf{cert}_{rev})$, with $\mathsf{cert}_{rev} = \langle x, \sigma_{pool} \rangle$.

Fig. 4. The first part of the Collective Pool Functionality, parameterized with threshold T and weight mapping ω, refers to the creation and management of the pool (the second part is given by Figure 7).

Collective Pool Protocol $\pi_{pool}^{T,\omega}$ (first part)

Gathering: Upon receiving (GATHER, sid), send (KeyGen, sid) to $\mathcal{F}_{wtss}$, with sid containing the weight mapping ω and the threshold T. Upon receiving the reply $(\text{KeyGen}, sid, \mathsf{vk}_{pool})$, return $(\text{GATHER-OK}, sid, \mathsf{vk}_{pool})$.
Pool Registration: Upon receiving $(\text{REGISTER}, sid, W)$, send (INIT, sid, W) to Γ_{reward} and wait for the reply $(\text{INIT-OK}, sid, \alpha_{reward})$. Then, set $m = (W, \alpha_{reward})$ and send (Sign, sid, m) to $\mathcal{F}_{wtss}$. Upon receiving a reply $(\text{Sign}, sid, m, \sigma_{pool})$, return $(\text{REGISTER-OK}, sid, \mathsf{cert}_{reg})$, where $\mathsf{cert}_{reg} = \langle (W, \alpha_{reward}, \mathsf{vk}_{pool}, \sigma_{pool}) \rangle$.
Reward Withdrawal: Upon receiving $(\text{WITHDRAW}, sid, \alpha, f)$, forward it to Γ_{reward}. Upon receiving a response $(\text{TRANSACTION}, sid, \langle \alpha_{sc}, \alpha, r, f \rangle)$ return it.
Closing: Upon receiving (CLOSE, sid, x), send (Sign, sid, x) to $\mathcal{F}_{wtss}$. Upon receiving a reply $(\text{Sign}, sid, x, \sigma_{pool})$, return $(\text{CLOSE-OK}, sid, \mathsf{cert}_{rev})$ with $\mathsf{cert}_{rev} = \langle x, \sigma_{pool} \rangle$.

Fig. 5. The first part of the Collective Pool Protocol, which describes the set of management operations (the second part is given by Figure 8).

To construct a distributed mempool, we consider a subselection mechanism to identify the parties that verify each transaction. This mechanism should be: a) *non-interactive* b) *deterministic*, c) *balanced*, i.e., every party should be chosen with the same probability. Subselection is secure if a majority of the elected committee is honest. However, since the adversary may corrupt some pool members, this may not always be the case. We model this uncertainty via the probability $\Pi^{\theta,t,n}$, which depends on the size of the committee and the power of the adversary among the pool's members.

A straightforward way to implement subselection is to assume that the pool's members are ordered in a well-defined manner, e.g., lexicographically. Given the ordered list $L = [p_1, p_2, \ldots, p_n]$ of the pool's members, we use a permutation algorithm $\pi_{perm}(\cdot, \cdot, \cdot)$, which takes two arguments, i) a transaction τ, ii) a chain $\mathcal{C}$, and iii) the ordered list of pool members L, and outputs a pseudorandom permuted list L_τ. For every transaction τ and a given chain $\mathcal{C}$, the committee responsible for verification consists of the θ first members in L_τ. Naturally, this proposal is rather simple, so alternative, e.g., VRF-based, mechanisms could be proposed to improve performance.

We note that using $\mathcal{C}$ during the subselection mechanism is important to avoid adaptive attacks. Specifically, the chain $\mathcal{C}$ simulates a randomness beacon, such that at least one of its last u blocks is honest, for some parameter u. If $\mathcal{C}$ was not used, the adversary could construct a malicious transaction in such way that the subselected committee would also be malicious. By using $\mathcal{C}$ as a seed to the pseudorandom permutation, the adversary's ability to construct such malicious transaction is limited. Alternatively, cryptographic sortition [23] could be employed to fully handle adaptive adversaries.

The (honest) members need to always have the same view of the distributed mempool; this is achieved via authenticated broadcast. Assuming a Public Key Infrastracture, as is our setting, it is possible to achieve deterministic authenticated broadcast in $t + 1$ rounds for t adversarial parties [35,42,16]. Each time a party adds a transaction to its mempool, it broadcasts it, such that, at any point in time, the honest members of the pool have the same view of the network w.r.t. the canonical chain and the mempool of unconfirmed transactions. We remind that, as shown by Garay *et al.* [20], $\mathcal{F}_{BC}$ can be implemented to ensure adaptive corruptions using commitments. We note that, in existing distributed ledgers, the order with which transactions are added to the mempool does not affect the choice when creating a new block; for instance, transactions of a new block are typically chosen based on a fee-per-byte score. If the order of transactions is pertinent, a stronger primitive like Atomic Broadcast [15] could be employed.

Following, the committee employs a consensus sub-protocol to agree on the transaction's validity. When a party p retrieves a new transaction τ from the network, it broadcasts it as above. Then, each party computes the permuted list L_τ. Each party, which is in the validation committee for τ, computes locally the validation predicate and submits its output to the consensus protocol. The consensus protocol should offer *strong validity*, i.e., if all honest parties should have the same input bit, they should output this bit. Finally, the output of the

consensus protocol is broadcast to the rest of the pool. To verify the committee's actions, a party may request the transcript of the consensus sub-protocol.

Finally, to compute the probability of electing an honest committee, we have a hypergeometric distribution, with population size n and $n-t$ honest parties, where a sample of parties of size θ is chosen *without replacement*. Thus, the probability of honest committee majority is: $\Pi^{\theta,t,n} = 1 - \sum_{v=\lfloor \frac{\theta+1}{2} \rfloor}^{\min(\theta,t)} \frac{\binom{t}{v} \cdot \binom{n-t}{\theta-v}}{\binom{n}{\theta}}$. Figure 6 provides further intuition on the probability w.r.t. the subselection parameter θ.

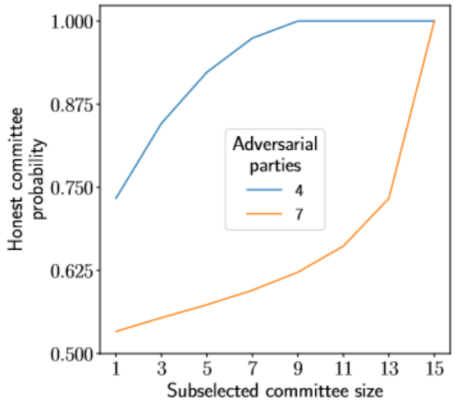

Fig. 6. The probability of subselecting an honest committee w.r.t. the committee size θ, $n = 15$ total parties and $\lfloor \frac{n-1}{3} \rfloor$ and $\lfloor \frac{n-1}{2} \rfloor$ adversarial parties.

Following, Figure 7 defines the second part of our functionality, while Figure 8 presents the second part of our protocol.

4.4 The Security of the Conclave Collective Stake Pool

Theorem 1. *The protocol π_{pool}, parameterized by a validation predicate* Validate*, a permutation algorithm π_{perm}, and a consensus protocol $\pi_{consensus}$ (cf. Definition 5) securely realizes $\mathcal{F}_{pool}$ with the hybrid execution* $\mathsf{HYBRID}^{pool}_{\pi_{pool},\mathcal{A},\mathcal{Z}}$ *in the global $\overline{\mathcal{G}}_{simpleLedger}$ model, and $\Pi^{\theta,t,n} = 1 - \sum_{v=\lfloor \frac{\theta+1}{2} \rfloor}^{\min(\theta,t)} \frac{\binom{t}{v} \cdot \binom{n-t}{\theta-v}}{\binom{n}{\theta}}$, assuming $\sum_{p \in P_{\mathcal{A}}} w_p < T$, where θ is the subselection parameter for transaction verification, $P_{\mathcal{A}}$ is the set of t corrupted parties out of n total parties, ω is the weight distribution of the n parties, and T is the signature threshold.*

Proof. The proof is constructed in the UC Framework, therefore it is simulation-based. As such, we will show that the environment $\mathcal{Z}$ cannot efficiently distinguish between two executions, the ideal and the real. The simulator $\mathcal{S}$ interacts

Collective Pool Functionality $\mathcal{F}_{pool}^{T,\omega}$ (second part)

Transaction Verification: Upon receiving (TRANSACTION, sid, τ, θ) from p_i, forward it to $\mathcal{S}$. Then send READ to $\overline{\mathcal{G}}_{simpleLedger}$ on behalf of p_i and wait for the reply $\mathcal{C}$. Following, set t as the number of corrupted parties; with probability $\Pi^{\theta,t,n}$ set $b := \mathsf{Validate}(\tau, \mathcal{C})$, otherwise (with probability $1 - \Pi^{\theta,t,n}$), send (TRANSACTION-VER, sid, τ) to $\mathcal{S}$, wait for a reply (TRANSACTION-OK, $sid, \mathcal{C}, \tau, f$), and set $b := f$. Finally, if $b = 1$, insert τ to mempool and send (TRANSACTION, $sid, \mathcal{C}, \tau, b$) to all parties.
Mempool Update: Upon receiving (TRANSACTION, $sid, \mathcal{C}', \tau, 1$) from p_i, forward it to $\mathcal{S}$. Then send READ to $\overline{\mathcal{G}}_{simpleLedger}$ on behalf of p_i and wait for the reply $\mathcal{C}$. If $\mathcal{C}' \prec \mathcal{C}$ and p_i is honest, insert τ to mempool and return (MEMPOOL-UPDATED, sid, τ).
Block issuing: Upon receiving (ISSUE-BLOCK, sid) from a party p, forward it to $\mathcal{S}$. When a set of parties $\mathbb{P}$ has submitted (ISSUE-BLOCK, sid), if $\sum_{j\in[1,m]} W[p_j] > T$, then for every party $p_i \in \mathbb{P}$, send READ to $\overline{\mathcal{G}}_{simpleLedger}$ on behalf of p_i and wait for the reply $\mathcal{C}_i$. If all received chains equal, i.e., are the same chain $\mathcal{C}$, remove every τ in mempool that also exists in $\mathcal{C}$. Then, set $b = \mathsf{blockify}(\mathsf{mempool})$, send (ISSUE-BLOCK, sid, b) to $\mathcal{S}$, and wait for the reply (ISSUE-BLOCK, sid, b, σ_{pool}). Following, check if $\forall(m, \sigma, b') \in T : \sigma \neq \sigma_{pool}, (b, \sigma_{pool}, 0) \notin T$; if the checks hold, insert $(b, \sigma_{pool}, 1)$ to T. Finally, reply with (BLOCK, sid, b, σ_{pool}).

Fig. 7. The second part of the proposed Pool Functionality, which defines the consensus participation operations.

Collective Pool Protocol $\pi_{pool}^{T,\omega}$ (second part)

Transaction Verification: Upon receiving (TRANSACTION, sid, τ, θ), send READ to $\overline{\mathcal{G}}_{simpleLedger}$ and wait for the reply $\mathcal{C}$. Then, set $b = \mathsf{Validate}(\mathcal{C}, \tau)$, compute $L' = \pi_{perm}(\tau, \mathcal{C}, L)$ and initiate protocol $\pi_{consensus}$ with the θ first parties in L' with input b. Upon computing the output of $\pi_{consensus}$, β, send (TRANSACTION, $sid, \mathcal{C}, \tau, \beta$) to $\mathcal{F}_{BC}$ and return it.
Mempool Update: Upon receiving (TRANSACTION, $sid, \mathcal{C}', \tau, 1$), p_i, send READ to $\overline{\mathcal{G}}_{simpleLedger}$ and wait for the reply $\mathcal{C}$. If $\mathcal{C}' \prec \mathcal{C}$, insert τ to mempool and return (MEMPOOL-UPDATED, sid, τ).
Block Issuing: Upon receiving (ISSUE-BLOCK, sid), send READ to $\overline{\mathcal{G}}_{simpleLedger}$ and wait for the reply $\mathcal{C}$. For every τ in mempool, if τ is also in $\mathcal{C}$, then remove τ from mempool. Next, set $b = \mathsf{blockify}(\mathsf{mempool})$ and send (SIGN, sid, b) to $\mathcal{F}_{wtss}$. Upon receiving a reply (SIGN, sid, b, σ_{pool}), return (BLOCK, sid, b, σ_{pool}).

Fig. 8. The second part of our protocol, which describes the set of operations for consensus participation.

with the ideal functionality $\mathcal{F}_{pool}$ in the ideal execution, whereas $\mathcal{A}$ interacts with π_{pool} in the real execution. We will show that, if π_{pool} does not securely realize the ideal functionality $\mathcal{F}_{pool}$, when instantiated with the parameters defined in the theorem, then at least one of the conditions is violated.

First, we provide the construction for the simulator. $\mathcal{S}$ runs internally a copy of the adversary $\mathcal{A}$. $\mathcal{S}$ forwards any inputs received from the environment $\mathcal{Z}$ to the internal copy of $\mathcal{A}$, and vice versa.

Gathering: Upon receiving the message (GATHER, sid) for all parties $p_i, i \in [1, n]$ from $\mathcal{F}_{pool}$, send (KEYGEN, sid) to $\mathcal{F}_{wtss}$ with the appropriate sid. Upon receiving the reply (KEYGEN, sid, vk_{pool}), record vk, and return (GATHER-OK, sid, vk) to $\mathcal{F}_{pool}$.

Pool Registration: Upon receiving from $\mathcal{F}_{pool}$ the n messages (REGISTER, sid, members), send (INIT, sid, members) to Γ_{reward} and wait for the reply (INIT-OK, sid, α_{reward}). Then send (SIGN, sid, m) to $\mathcal{F}_{wtss}$ with $m = (\mathsf{members}, \alpha_{reward})$. Upon receiving a reply (SIGN, sid, m, σ_{pool}), register (m, σ_{pool}) and return the message (REGISTER-OK, $sid, \alpha_{reward}, \sigma_{pool}$) to $\mathcal{F}_{pool}$.

Closing: Upon receiving (CLOSE, sid, x) from $\mathcal{F}_{pool}$, on behalf of a set of parties $\mathbb{P}$, send the message (SIGN, sid, x) to $\mathcal{F}_{wtss}$ on behalf of each party in $\mathbb{P}$. Upon receiving a reply (SIGN, sid, x, σ_{pool}), record σ_{pool} and return (CLOSE-OK, sid, σ_{pool}) to $\mathcal{F}_{pool}$.

Transaction Verification: Upon receiving (TRANSACTION-VER, sid, τ) from $\mathcal{F}_{pool}$, forward it to the internal copy of $\mathcal{A}$, wait for the output (TRANSACTION-OK, $sid, \mathcal{C}, \tau, f$) from $\mathcal{A}$ and forward it to $\mathcal{F}_{pool}$.

Block issuing: Upon receiving (ISSUE-BLOCK, sid, b) from $\mathcal{F}_{pool}$, send (SIGN, sid, b) to $\mathcal{F}_{wtss}$. Upon receiving a reply (SIGN, sid, b, σ_{pool}) and record (b, σ_{pool}). Finally, return (ISSUE-BLOCK, sid, b, σ_{pool}) to $\mathcal{F}_{pool}$.

Party corruption: When the adversary $\mathcal{A}$ corrupts a party p, $\mathcal{S}$ corrupts the same party in the ideal process and hands to $\mathcal{A}$ its internal state.

Global ledger update: When $\mathcal{A}$ sends $(\mathsf{ADVANCE}, p, \Sigma)$ to the global ledger $\overline{\mathcal{G}}_{simpleLedger}$, $\mathcal{S}$ also does so in the ideal world; similarly, when $\mathcal{A}$ sends (EXTEND, b) to $\overline{\mathcal{G}}_{simpleLedger}$, so does $\mathcal{S}$.

Signature generation: When the adversary $\mathcal{A}$ requests a signature on some message m, $\mathcal{S}$ sends (SIGN, sid, m) to $\mathcal{F}_{wtss}$; upon receiving the reply (SIGN, sid, m, σ), it returns σ to $\mathcal{A}$.

The first observation is that $\mathcal{S}$ needs to ensure that a party p has the same view of the ledger as in the real world. Therefore, it advances parties only when the real world adversary $\mathcal{A}$ does so.

To prove the theorem, we assume that π_{pool} does not realize $\mathcal{F}_{pool}$, i.e., there exists adversary $\mathcal{A}$ such that, for every simulator $\mathcal{S}$, there exists environment $\mathcal{Z}$ that can distinguish between the ideal world (of $\mathcal{F}_{pool}$ and $\mathcal{S}$) and the real world (of π_{pool} and $\mathcal{A}$). Following, we show that $\mathcal{S}$ violates the security of one of the primitives used by π_{pool}, i.e., the consensus protocol $\pi_{consensus}$ and the weighted threshold signature scheme Σ_{thresh}.

We build an algorithm $\mathcal{D}$ that breaks the security of the cryptographic primitives as follows. $\mathcal{D}$ runs a simulated copy of $\mathcal{Z}$ and simulates for $\mathcal{Z}$ the ideal environment, i.e., $\mathcal{D}$ acts both as $\mathcal{F}_{pool}$ and $\mathcal{S}$ on $\mathcal{Z}$'s messages.

Similar to $\mathcal{S}$, $\mathcal{D}$ runs a simulated copy of $\mathcal{A}$. When running **Gathering** to obtain the threshold keys, instead of running Thresh-Key-Gen, $\mathcal{D}$ hands $\mathcal{A}$ the public key vk which is obtained as the input from $\mathcal{S}$. To obtain a signature σ on a message m, $\mathcal{D}$ hands m to its oracle, instead of using Thresh-Sign. When $\mathcal{A}$ advances the state of party p in the global ledger $\overline{\mathcal{G}}_{simpleLedger}$, $\mathcal{D}$ does so as well.

Regarding the consensus subprotocol $\pi_{consensus}$, we consider the case when $\mathcal{Z}$ activates an uncorrupted party p with input a transaction τ via the interface **Transaction Verification**. At that point, $\mathcal{D}$ computes $b = \mathsf{Validate}(\mathcal{C}, \tau)$, where $\mathcal{C}$ is the state of party p in the global ledger $\overline{\mathcal{G}}_{simpleLedger}$. Next, $\mathcal{D}$ checks the output b' in the real world (where $\mathcal{A}$ operates). If the majority of the committee elected to validate τ is honest and $b \neq b'$, then $\mathcal{D}$ retrieves the transcript of $\pi_{consensus}$, run for the validation of τ by $\mathcal{A}$, and outputs it (observe that this transcript is represents an execution of $\pi_{consensus}$ where its security breaks).

To analyze the success probability of $\mathcal{D}$, we consider the event E, where $b \neq b'$, as defined above. Since $\pi_{consensus}$ is secure as long as a majority of participants is honest, the executions of the real world, i.e., the interaction of $\mathcal{Z}$ with $\mathcal{A}$ and π_{pool}, and the ideal world (resp. $\mathcal{S}$ and $\mathcal{F}_{pool}$) are statistically close. If we are guaranteed that $\mathcal{Z}$ distinguishes between the two executions, then E occurs with non-negligible probability. Finally, from the point of view of $\mathcal{A}$ and $\mathcal{Z}$, the interaction with $\mathcal{D}$ is the same as with an interaction with protocol π_{pool} in the real world.

Regarding the weighted threshold signatures, we note that the functionality $\mathcal{F}_{pool}$ performs the same checks regarding signature issuing as $\mathcal{F}_{wtss}$. In fact, only signature generation is performed by the collective pool; signature verification should be employed when advancing the ledger state, i.e., upon adopting new blocks, or when validating a certificate. Therefore, the security of $\mathcal{F}_{wtss}$ ensures that $\mathcal{Z}$ cannot distinguish the two executions (real vs. ideal world) w.r.t. the weighted threshold signatures.

Regarding block issuing we consider the event E', where a set of parties controlling a majority of the pool's stake initiate block issuing, but no signed block is output. In that case, either the signature issuing of Σ_{thresh} fails or the parties locally produce a different block b, i.e., their mempool is not synchronized. Regarding the former, the same analysis on signature issuing as above applies. Regarding the latter, if two honest parties hold a different mempool at the point when blockify is used, either their ledger state $\mathcal{C}$ is different or their mempool is

different. This implies that $\mathcal{F}_{BC}$ fails for at least one transaction τ, i.e., an honest party inserts τ in its mempool and, after τ is sent to $\mathcal{F}_{BC}$, at least one other honest party fails to also insert it to its mempool. However, this is impossible, since the simulator ensures the former and $\mathcal{F}_{BC}$ ensures the latter.

Finally, the permutation algorithm π_{perm} is executed locally by each party, therefore the adversary cannot affect its output. Additionally, the probability $\Pi^{\theta,t,n}$ is computed following the analysis of Section 4.3. □

4.5 Incentives of the Collective Pool

Although, as shown in Theorem 1, π_{pool} is secure, it is unclear whether rational users will opt for using it. In this section, we discuss the incentive compatibility of π_{pool}. We identify its shortcomings and propose a minor change, such that rational members cannot gain more rewards by deviating from it.

First, we consider the cost of each operation performed in π_{pool}. Signing operations does not incur any cost, thus pool registration and revocation are cost-free. Block production depends on the internal workings of blockify. For instance, solving the Knapsack problem can be expensive, while a greedy algorithm that prioritizes high-fee transactions is typically not. Therefore, without loss of generality, we also assume that block production is cost-free. However, both mempool update and transaction verification incur costs c_{mu} and c_{tv} respectively. A mempool consists of millions of transactions and verifying them requires an accurate view of the ledger. Thus, both objects may require significant amounts of computation complexity and storage.[7]

We focus on the *profit* of each member, i.e., the rewards subtracted by the cost of executing π_{pool}. The core observation is that a member p receives $r_p = \frac{w_p}{\sum_{p' \in \omega} w_{p'}}$ of the total pool's rewards *regardless of its performance*. For instance, if p acts only on the pool's creation, it still receives its proportional share of rewards for the blocks produced by the rest of the pool. Therefore, as long as a member believes that the other members act honestly, it is incentivized to abstain and minimize its cost, thus maximizing its profit. Naturally, if all parties follow this strategy, the pool produces no blocks and receives no rewards.

A possible solution is to the Free Rider problem above is to penalize a party for misbehaving. However, identifying misbehavior is not straightforward. For instance, a party p who inputs 0 to $\pi_{consensus}$ for a transaction τ may do so either because the transaction is invalid or because it didn't perform validation and input 0 by default. Our approach is to penalize a party when diverging from the rest of the pool. In the previous example, if the output of $\pi_{consensus}$ is 1, then p incurs a fixed penalty $\widehat{c_{tv}}$. Similarly, if a party fails to sign a new block then it incurs a (fixed) penalty $\widehat{c_{mu}}$. The penalty amount, which is withheld from p, is then distributed equally among the other pool members. To incentivize p to follow π_{pool} the penalties should be high enough; specifically, it should hold

[7] As of January 2021, the Bitcoin chain is roughly 320GB and increases linearly over time. (`https://www.blockchain.com/charts/blocks-size`)

$\widehat{c_{tv}} > c_{tv}$ and $\widehat{c_{mu}} > c_{mu}$. If all parties follow π_{pool}, diverting incurs a cost $\widehat{c_{mu}} - c_{mu} > 0$ (resp. $\widehat{c_{tv}} - c_{tv} > 0$), thus the new protocol is an equilibrium.

Finally, penalties can be automatically enforced via an interface to the smart contract Γ_{reward} which, given a proof of misbehavior, reduces the misbehaving party's rewards accordingly. For transaction verification, a proof of misbehavior is the transcript of $\pi_{consensus}$, which describes the consensus sub-protocol's execution. For block issuing, we can use a threshold signature scheme with identifiable abort [22], which allows to identify the parties that do not participate in the signing of a block. In the next section, we construct such WTSS with identifiable abort, which can be used in an incentive-compatible collective pool.

5 Weighted Threshold ECDSA

Our final contribution is a weighted threshold signature construction, which can be used in the implementation of π_{pool}. Our scheme is based on [21]; specifically, we introduce weights, with each party having as many shares as "units" of weight. The preliminaries of key generation and signing are available at Section 2.6.

Our construction is a (t, n, ω)-weighted threshold ECDSA. We assume that each player p_i has a associated a weight w_i, identified by the (publicly available) weight function ω such that $\omega[p_i] = w_i$; ω is a parameter in the following two algorithms(cf. Section 2.4). Furthermore, we assume an index function $\mathcal{I}(i, w)$ in the secret sharing scheme, which assigns a unique index to each pair (p_i, w_i).

Following, we instantiate the algorithms Thresh-Key-Gen and Thresh-Sign. We outline the changes of our constructions to obtain identifiable abort capability based on [22], to make it suitable for an incentive-compatible pool. We note that some PoS protocols employ a Verifiable Random Function (VRF) [13,23]. Thus, this section's secret sharing techniques can also be used to distribute the VRF key in a weighted manner.

5.1 Key Generation Protocol $\mathsf{Thresh\text{-}Key\text{-}Gen}_{\omega}$

Each party p_i is associated with a public key for the homomorphic encryption E_i and the weight w_i.

- Phase 1: Each party p_i picks its share proportionally to its weight, i.e., w_i shares. Then it commits to them and broadcast them together with its homomorphic encryption key E_i.
  - Pick uniformly random local values $u_i^{(1)}, \ldots, u_i^{(w_i)} \in \mathbb{Z}_p$
  - Compute $y_i^{(w)} = \mathsf{com}(g^{u_i^{(w)}}) = [C_i^{(w)}, D_i^{(w)}]$, for $\forall w = \{1, \ldots, w_i\}$
  - Broadcast $C_i^{(1)}, \ldots, C_i^{(w_i)}$
  - Broadcast E_i
- Phase 2: The confirmation of the values is done through opening of commitments, and each value for each weight is secretly shared among all the players. Therefore each player executes as many secret-sharing instance as weight "units" it has, resulting in its combined shares for the secret key $(x_i^{(1)}, \ldots, x_i^{(w_i)})$ proportionally to its weight w_i.

  - Broadcast $D_i^{(1)}, \ldots, D_i^{(w_i)}$
  - Receive the decommitments for $(y_j^{(1)}, \ldots, y_j^{(w_j)})$, $\forall j \in \{1, \ldots, n\}, j \neq i$
  - Perform secret-sharing for each share $u_i^{(1)}, \ldots, u_i^{(w_i)}$, s.t. for each value $u_i^{(w)}$ compute the shares $u_{i,\mathcal{I}(j,w')}^{(w)}$ and secretly send to p_j, with respect to weight $1 \leq w' \leq w_j$ and index $\mathcal{I}(j, w')$, receiving back the share $u_{\mathcal{I}(j,w'),i}^{(w)}$
  - Each player p_i compute its respective set of shares

$$x_i^{(1)} = \sum_{\substack{1 \leq j \leq n \\ 1 \leq w' \leq w_j}} u_{\mathcal{I}(j,w'),i}^{(1)} \ , \ \ldots \ , \ x_i^{(w_i)} = \sum_{\substack{1 \leq j \leq n \\ 1 \leq w' \leq w_j}} u_{\mathcal{I}(j,w'),i}^{(w_i)}$$

    with the values received from other parties p_j.
- Phase 3: For the public key E_i, the module $N_i = p_i \cdot q_i$ for primes p_i and q_i provide zero-knowledge proof for:
  - for p_i and q_i (Proof of knowledge for factoring [43])
  - and $x_i^{(1)}, \ldots, x_i^{(w_i)}$ (Schnorr based)

Note that the joint public-key is $\mathsf{vk} = \prod_{i=1}^{i=n} \prod_{w=1}^{w=w_i} y_i^{(w)}$, whereas the joint secret-key is $\mathsf{tsk} = \sum_{i=1}^{i=n} \sum_{w=1}^{w=w_i} x_i^{(w)}$.

5.2 Signing Protocol Thresh-Sign$_\omega$

We assume a set B of parties p_i that jointly compute a signature.

- Phase 1: Each party selects two tuples of values, each with w_i values, and broadcasts w_i commitments to one of the sets.
  - Pick random values $k_i^{(1)}, \ldots, k_i^{(w_i)} \in_R \mathbb{Z}_p$
  - Pick random values $\gamma_i^{(1)}, \ldots, \gamma_i^{(w_i)} \in_R \mathbb{Z}_p$
  - Define $k = \sum_{i \in B} \sum_{w=1}^{w=w_i} k_i^{(w)}$ and $\gamma = \sum_{i \in B} \sum_{w=1}^{w=w_i} \gamma_i^{(w)}$
  - Compute w_i commitments $\mathsf{com}(g^{\gamma_i^{(w)}}) = [C_i^{(w)}, D_i^{(w)}]$ for $\forall w = \{1, \ldots, w_i\}$
  - Broadcast $C_i^{(1)}, \ldots, C_i^{(w_I)}$
- Phase 2: Each party computes the interpolation coefficients $\lambda_i^{(w)}$ for each share it keeps, that is the shares for weights $w = \{1, \ldots, w_i\}$, taking into account its indexes $\mathcal{I}(i, w)$.
  - For $w = \{1, \ldots, w_i\}$ and $w' = \{1, \ldots, w_j\}$, compute the Lagrangian coefficients $\lambda_{i,B}^{(w)} = \prod_{j \in B, w'=1}^{w'=w_j} \frac{-\mathcal{I}(j,w')}{\mathcal{I}(i,w) - \mathcal{I}(j,w')}$
  - Compute the values

$$\mathbf{x}_i^{(1)} = (\lambda_i^{(1)}) \cdot (x_i^{(1)}) \ , \ \ldots \ , \ \mathbf{x}_i^{(w_i)} = (\lambda_i^{(w_i)}) \cdot (x_i^{(w_i)}).$$

- Phase 2A - Local Shares: The party p_i executes locally the MtA protocol with the local shares, which are $(k_i^{(1)}, \ldots, k_i^{(w_i)})$ and $(\gamma_i^{(1)}, \ldots, \gamma_i^{(w_i)})$ to compute α and β such that $k_i^{(w)} \gamma_i^{(w')} = \alpha_{i,i}^{(w)(w')} + \beta_{i,i}^{(w)(w')}$ for p_i and $1 \leq w, w' \leq w_i$. Note that both values of the pair $k_i^{(w)}$ and $\gamma_i^{(w)}$ are used which means MtA is executed twice for a given party p_i and weight w.

- Phase 2B - Online Shares: Party p_i executes MtA protocol between its local shares $(k_i^{(1)}, \ldots, k_i^{(w_i)})$ and shares of the remaining parties, other than p_i:

$p_1, \ldots, p_{(i-1)}$	$p_{(i+1)}, \ldots, p_n$
$(\gamma_1^{(1)}, \ldots, \gamma_1^{(w_1)})$	$(\gamma_{i+1}^{(1)}, \ldots, \gamma_{i+1}^{(w_{i+1})})$
$\vdots$	$\vdots$
$(\gamma_{i-1}^{(1)}, \ldots, \gamma_{i-1}^{(w_{i-1})})$	$(\gamma_n^{(1)}, \ldots, \gamma_n^{(w_n)})$

  Like Local Shares, there will be two MtA executions for each pair $k_i^{(w)}$ and $\gamma_i^{(w)}$, i.e., $k_i^{(w)}\gamma_j^{(w')} = \alpha_{i,j}^{(w)(w')} + \beta_{j,i}^{(w)(w')}$.
- Phase 2C - Compute $\delta_i^{(w)}$, for $1 \leq w \leq w_i$ and $1 \leq i \leq n$ the following values by summing the produced values from steps 2A and 2B. Second and third terms from 2A, and the remaining terms from 2B:

$$\delta_i^{(w)} = k_i^{(w)}\gamma_i^{(w)} + \sum_{\substack{w'=1 \\ w \neq w'}}^{w'=w_i} \alpha_{i,i}^{(w)(w')} + \sum_{\substack{w'=1 \\ w \neq w'}}^{w'=w_i} \beta_{i,i}^{(w)(w')}$$

$$+ \sum_{\substack{1 \leq \ell \leq i-1 \\ 1 \leq w' \leq w_\ell \\ j \in B}} \left(\alpha_{i,j}^{(w)(w')} + \beta_{j,i}^{(w)(w')}\right) + \sum_{\substack{i+1 \leq \ell \leq n \\ 1 \leq w' \leq w_\ell \\ j \in B}} \left(\alpha_{i,j}^{(w)(w')} + \beta_{j,i}^{(w)(w')}\right).$$

- Phase 2D - Local Shares: Party p_i executes locally the MtA protocol wi the local shares which are $(k_i^{(1)}, \ldots, k_i^{(w_i)})$ and $(\mathbf{x}_i^{(1)}, \ldots, \mathbf{x}_i^{(w_i)})$ to compute μ and ν such that $k_i^{(w)}\mathbf{x}_i^{(w')} = \mu_{i,i}^{(w)(w')} + \nu_{i,i}^{(w)(w')}$ for p_i and $1 \leq w, w' \leq w_i$.
- Phase 2E - Online Shares: Party p_i executes MtAwc protocol between its local shares $(k_i^{(1)}, \ldots, k_i^{(w_i)})$ and shares of the remaining parties except p_i:

$p_1, \ldots, p_{(i-1)}$	$p_{(i+1)}, \ldots, p_n$
$(\mathbf{x}_1^{(1)}, \ldots, \mathbf{x}_1^{(w_1)})$	$(\mathbf{x}_{i+1}^{(1)}, \ldots, \mathbf{x}_{i+1}^{(w_{i+1})})$
$\vdots$	$\vdots$
$(\mathbf{x}_{i-1}^{(1)}, \ldots, \mathbf{x}_{i-1}^{(w_{i-1})})$	$(\mathbf{x}_n^{(1)}, \ldots, \mathbf{x}_n^{(w_n)})$

  Likewise the Local Shares, there will be two executions of the MtAwc protocol for each pair $k_i^{(w)}$ and $\mathbf{x}_i^{(w)}$, that is $k_i^{(w)}\mathbf{x}_j^{(w')} = \mu_{i,j}^{(w)(w')} + \nu_{j,i}^{(w)(w')}$.
- Phase 2F: Compute $\sigma_i^{(w)}$, for $1 \leq w \leq w_i$ and $1 \leq i \leq n$ the following values by summing the produced values from Steps 2D and 2E. Second and third terms from 2D, and the remaining terms from 2E:

$$\sigma_i^{(w)} = k_i^{(w)}\mathbf{x}_i^{(w)} + \sum_{\substack{w'=1 \\ w \neq w'}}^{w'=w_i} \mu_{i,i}^{(w)(w')} + \sum_{\substack{w'=1 \\ w \neq w'}}^{w'=w_i} \nu_{i,i}^{(w)(w')}$$

$$+ \sum_{\substack{1\le\ell\le i-1\\ 1\le w'\le w_\ell\\ j\in B}} \left(\mu_{i,j}^{(w)(w')} + \nu_{j,i}^{(w)(w')}\right) + \sum_{\substack{i+1\le\ell\le n\\ 1\le w'\le w_\ell\\ j\in B}} \left(\mu_{i,j}^{(w)(w')} + \nu_{j,i}^{(w)(w')}\right).$$

- Phase 3: At this point each party p_i has two sets of values $(\delta_i^{(1)}, \ldots, \delta_i^{(w_i)})$ and $(\sigma_i^{(1)}, \ldots, \sigma_i^{(w_i)})$ from, respectively, Steps 2C and 2F. The party p_i broadcasts the former set, and all parties reconstruct the value $\delta = \sum_{\substack{w=1\\ i\in B}}^{w=w_i} \delta_i^{(w)} = k \cdot \gamma$ (as defined in Step 1).
- Phase 4: Release w_i commitments computed in Step 1, and use them to compute the r as the first part of the signature.
  - Broadcast the values $D_i^{(w)}$ which open the commitments for $\Gamma_i^{(w)} = g^{\gamma_i^{(w)}}$
  - p_i proves in ZK the knowledge of $\gamma_i^{(w)}$ for $1 \le w \le w_i$
  - All compute

$$R = \left(\prod_{\substack{i\in B\\ 1\le w\le w_i}} \Gamma_i^{(w)}\right)^{\delta^{-1}} = g^{\left(\sum_{\substack{i\in B\\ 1\le w\le w_i}} \gamma_i^{(w)}\right) k^{-1}\gamma^{-1}} = g^{\gamma k^{-1}\gamma^{-1}} = g^{k^{-1}}$$

  - Compute the first half of the signature as r=$R \mod p$
- Phase 5: Each player p_i computes $s_i^{(w)} = mk_i^{(w)} + r\sigma_i^{(w)}$, so each player p_i holds the set $(s_i^{(1)}, \ldots, s_i^{(w_i)})$ of shares of the second part of the signature.
- Phase 5A: To build the second half of the signature it is necessary to randomly sample and commit to two value sets:
  - Choose two sets of random values $(\ell_i^{(1)}, \ldots, \ell_i^{(w_i)})$ and $(\rho_i^{(1)}, \ldots, \rho_i^{(w_i)})$ such that $\ell_i^{(w)} \in \mathbb{Z}_p$ and $\rho_i^{(w)} \in \mathbb{Z}_p$.
  - Compute the set $(V_i^{(1)}, \ldots, V_i^{(w)})$ such that $V_i^{(w)} = r^{s_i^{(w)}} g^{\ell_i^{(w)}}$
  - Compute $(A_i^{(1)}, \ldots, A^{(w)_i})$ such that $A_i^{(w)} = g^{\rho_i^{(w)}}$
  - Compute the commitments $([\widehat{C}_i^{(1)}, \widehat{D}_i^{(1)}], \ldots, [\widehat{C}_i^{(w_i)}, \widehat{D}_i^{(w_i)}])$, such that $\mathsf{com}(V_i^{(w)}, A_i^{(w)}) = [\widehat{C}_i^{(w)}, \widehat{D}_i^{(w)}]$
  - Broadcast $(\widehat{C}_i^{(1)}, \ldots, \widehat{C}_i^{(w_i)})$
- Phase 5B: Once all committed values were received, open the commits in order to joint compute V and A:
  - Broadcast $(\widehat{D}_i^{(1)}, \ldots, \widehat{D}_i^{(w_i)})$
  - Prove in ZK, for each value w, such that $1 \le w \le w_i$, the knowledge of $\ell_i^{(w)}$, $\rho_i^{(w)}$ and $s_i^{(w)}$ such that $V_i^{(w)} = R^{s_i^{(w)}} g^{\ell_i^{(w)}}$ and $A_i^{(w)} = g^{\rho_i^{(w)}}$
  - Compute:

$$V = g^{-m} \cdot (\mathsf{vk})^{-r} \cdot \prod_{\substack{i\in B\\ 1\le w\le w_i}} V_i^{(w)}, A = \prod_{\substack{i\in B\\ 1\le w\le w_i}} V_i^{(w)}$$

- Phase 5C: Like Step 5A, compute two sets of values $U_i^{(w)}$ and $T_i^{(w)}$ and prove the knowledge of them via ZK proofs. These values are used to guarantee consistency of the shares:

- Compute the set $(U_i^{(1)}, \ldots, U_i^{(w_i)})$ such that $U_i^{(w)} = V^{\rho_i^w}$
- Compute the set $(T_i^{(1)}, \ldots, T_i^{(w_i)})$ such that $T_i^{(w)} = A^{\ell_i^w}$
- Compute the commitments $([\widetilde{C}_i^{(1)}, \widetilde{D}_i^{(1)}], \ldots, [\widetilde{C}_i^{(w_i)}, \widetilde{D}_i^{(w_i)}])$, such that $\mathsf{com}(U_i^{(w)}, T_i^{(w)}) = [\widetilde{C}_i^{(w)}, \widetilde{D}_i^{(w)}]$
- Broadcast $(\widetilde{C}_i^{(1)}, \ldots, \widetilde{C}_i^{(w_i)})$

- Phase 5D: Once the commitments are received, broadcasts their openings and verify the consistency of the shares:
  - Broadcast $(\widetilde{D}_i^{(1)}, \ldots, \widetilde{D}_i^{(w_i)})$
  - If $\prod_{\substack{i \in B \\ 1 \leq w \leq w_i}} T_i^{(w)} \neq \prod_{\substack{i \in B \\ 1 \leq w \leq w_i}} U_i^{(w)}$, then abort
- Phase 5E: Broadcast the shares of the second half of the signature, and reconstruct it:
  - Broadcast the set $(s_i^{(1)}, \ldots, s_i^{(w_i)})$
  - Compute the second signature share as $s = \sum_{\substack{i \in B \\ 1 \leq w \leq w_i}} s_i$. If (r, s) is not a valid signature, then abort.

5.3 Identifiable Abort

Here we describe the changes required to provide identifiable abort capability considering weights as it is used in our proposed construction. As mentioned earlier, weights can be also introduced in the extended version of [22]; we note that weights can be similarly applied to the scheme of [9], which extends [22]. The changes yield a similar construction as the one presented earlier, and affect only Phase 3, and the substitution of the Phases 5, 5A, 5B, 5C, 5D and 5E, to new Phases 5, 6 and 7. Identification follows similarly to [22], therefore we refer the reader to that work for a fully description of the procedure.

Concretely, for the new phases with weights below, consider $w \in \{1, \ldots, w_i\}$:

- Phase 3:
  - All parties reconstruct $\delta = \sum_{\substack{w=1 \\ i \in B}}^{w=w_i} \delta_i^{(w)} = k \cdot \gamma$ and compute $\delta^{-1} \mod p$
  - Compute $(T_i^{(1)}, \ldots, T_i^{(w_i)})$ such that $T_i^{(w)} = g^{\sigma_i^{(w)}} h^{\ell_i^{(w)}}$, and provide a ZK proof of knowledge of $(\ell_i^{(1)}, \ldots, \ell_i^{(w_i)})$ and $(\sigma_i^{(1)}, \ldots, \sigma_i^{(w_i)})$
- Phase 5: All players broadcast $\widetilde{R}_i^{(w)} = R^{k_i^{(w)}}$ and a ZK proof of range (as the ones sent in the MtA on Phase 2) between $R_i^{(w)}$ and $E_i(k_i^{(w)})$. If $g \neq \prod_{\substack{i \in B \\ 1 \leq w \leq w_i}} \widetilde{R}_i^{(w)}$, the protocol aborts.
- Phase 6: All parties broadcast $S_i^{(w)} = R^{\sigma_i^{(w)}}$ and a ZK knowledge proof (as in Phase 3) between $S_i^{(w)}$ and $T_i^{(w)}$. If $y \neq \prod_{\substack{i \in B \\ 1 \leq w \leq w_i}} S_i^{(w)}$, the protocol aborts.
- Phase 7: Each player broadcasts $s_i^{(w)} = mk_i^{(w)} + r\sigma_i^{(w)}$ and sets $s = \sum_{\substack{i \in B \\ 1 \leq w \leq w_i}} s_i$. If (r, s) is not a valid signature, abort.

5.4 Weighted Threshold ECDSA Complexity Estimation

Here we present communication and computation complexity of the weighted threshold signature scheme. As recommended in [21] the construction is based on the Random Oracle Heuristic as the choice for the hash function with size λ used for Non-Malleable Equivocable Commitment Scheme, with a decommitment string size of r bits and a group element.

Also in [21], two zero-knowledge schemes are considered: (1) Proof of knowledge for factoring [43], used in the homomorphic scheme, and (2) Schnorr based Proof of knowledge for the regular shares of the threshold signature scheme. Both can be made non-interactive with the Fiat-Shamir Technique, yielding in case of (1), very short proofs. The complexities are summarized in the Tables 1 and 3 for respectively, key generation and signing messages.

	Communication	Computational
Phase 1	$W\lambda$ bits, $2n\ \mathbb{Z}_{N^2}$	$w_i\ \mathcal{O}$
Phase 2	$W\ \mathbb{G}$, $Wr\ bits$, $w_i(W-w_i)\ \mathbb{Z}_p$, $w_i(W-w_i)(T+1)\ \mathbb{G}$	$(W-w_i)\ \mathcal{O}$, $w_i(T+1)\ exp\ \mathbb{G}$, $(T-w_i)(T+1)\ exp\ \mathbb{G}$
Phase 3	$w_i(\lambda\ bits + \mathbb{Z}_p + \mathbb{G})$	$w_i\ exp\ \mathbb{G}$, $w_i\ \mathcal{O}$, $2(W-w_i)\ exp\ \mathbb{G}$

Table 1. Key Generation with security parameter λ complexities: communication complexity is given in terms of group elements $\mathbb{G}$ and ring elements $\mathbb{Z}_p$ and $\mathbb{Z}_{N^2}$ for party p_i with weight w_i, the sum of all weights is W and the signature scheme threshold is T. Moreover computational complexity is defined by group exponentiations, random oracle queries $\mathcal{O}$, and modular multiplication.

For the signing procedure, the construction, as defined in [21], relies in multiple executions of the conversion protocol MtA and MtAwc, the version with checks. In the following Table 3 let MtA_{comm} and MtA_{comp} be the communication and computational complexities for MtA, and analogously for $MtAwc_{comm}$ and $MtAwc_{comp}$. The extra check is a zero knowledge proof of knowledge which requires one extra group exponentiation and a query for the random oracle to produce. The verification requires 2 exponentiation for group $\mathbb{G}$ and a random oracle query. Regarding the size, it is composed by a λ-bit string, and a group $\mathbb{G}$ and ring $\mathbb{Z}_p$ elements.

MtA_{comm}	$4\mathbb{Z}_{N^2} + 4\mathbb{Z}_{\widetilde{N}} + 2\mathbb{Z}_p$
MtA_{comp}	$4\ mult\ \mathbb{Z}_{\widetilde{N}} + 4\ mult\ \mathbb{Z}_{N^2} + 6\ exp\ \mathbb{Z}_{\widetilde{N}} + 6\ exp\ \mathbb{Z}_{N^2}$

Table 2. Complexity of conversion from multiple shares to Additive shares.

	Communication	Computational
Phase 1	$w_i(W_B - w_i)\lambda$ *bits*	w_i *exp* $\mathbb{G}$
Phase 2		$w_i(W_B + 1 - w_i)$ *mult* $\mathbb{Z}_p$
Phase 2A		$w_i(w_i - 1)$ MtA_{comp}
Phase 2B	$w_i(W_B - w_i - 1)$ MtA_{comm}	$w_i(W_B - w_i - 1)$ MtA_{comp}
Phase 2C		w_i *mult* $\mathbb{Z}_p$
Phase 2D		$w_i(w_i - 1)$ MtA_{comp}
Phase 2E	$w_i(W_B - w_i - 1)$ $MtAwc_{comm}$	$w_i(W_B - w_i - 1)$ $MtAwc_{comm}$
Phase 2F		w_i *mult* $\mathbb{Z}_p$
Phase 3	$w_i(W_B - w_i)$ $\mathbb{Z}_p$	
Phase 4	$w_i(W_B - w_i + 1)\mathbb{G} + w_i(W_B - w_i)r$ *bits* $w_i(\lambda$ *bits* $+ \mathbb{Z}_p)$	$(2W_B - w_i + 1)$ *exp* $\mathbb{G}$ $+(W_B - w_i)\mathcal{O}$
Phase 5		$2w_i$ *mult* $\mathbb{Z}_p$
Phase 5A	$w_i\lambda$ *bits*	$w_i(2$ *mult* $\mathbb{Z}_p + 2$ *exp* $\mathbb{G}$ $+$ *exp* $\mathbb{Z}_p + \mathcal{O})$
Phase 5B	$w_i(\mathbb{G} + rbits)$	at least $(2W_B - w_i)(\mathcal{O} +$ *exp* $\mathbb{G})$ $+2W_B$ *mult* $\mathbb{Z}_p$
Phase 5C	w_i λ *bits*	w_i $\mathcal{O}$
Phase 5D	$w_i(\mathbb{G} + r$ *bits*$)$	at least $2W_B$ *mult* $\mathbb{Z}_p$
Phase 5E	w_i $\mathbb{Z}_p$	

Table 3. Complexities of sign protocol; Phases 5A and 5D depend on the size of the stake of the signing set W_B, which may contain more than the threshold T.

Identifiable Abort. Table 4 illustrates the communication and computational complexities for the signing protocol of [22]. Note that again we take into account the weight of the set B, namely W_B, which can be larger than the threshold T.

6 Conclusion

Our work explores a novel design for collective stake pools for Proof-of-Stake ledgers, i.e., pools without a central operator. Our first contribution is a security definition for collective stake pools, which takes the form of the ideal functionality $\mathcal{F}_{pool}$ that articulates the security properties and functions that a collective pool should offer. Following, we propose the concrete protocol *Conclave* which UC-realizes $\mathcal{F}_{pool}$. Our construction incorporates a load balancing mechanism for transaction verification, to boost performance, as well as a Weighted Threshold Signature Scheme (WTSS). Regarding the latter, we present the ideal functionality $\mathcal{F}_{wtss}$ (Appendix 3) that formalizes this new definition and might be of independent interest, and propose two constructions based on threshold ECDSA. We stress that the collective pool is modular and agnostic to the WTSS implementation, so any scheme that securely realizes $\mathcal{F}_{wtss}$ suffices.

Our design satisfies most of the desiderata outlined in Section 1. Some (e.g., pool proportional rewards or stake reallocation) are dependent on the underlying ledger system's details, therefore are outside of our scope; nevertheless, our

	Communication	Computational
Phase 3	$W_B(2\ \mathbb{G}\ + 5\ \mathbb{Z}_p)$	$W_B\ mult\ \mathbb{Z}_p + 2(W_B + w_i\ exp\ \mathbb{G}) + 2W_B\ \mathcal{O}$
Phase 5	$W_B\ (2\ \mathbb{Z}_p\ + 2\ \mathbb{Z}_{\widetilde{N}}\ +\ \mathbb{Z}_{N^2})$	$W_B\ (mult\ \mathbb{G} + 2\ mult\ \mathbb{Z}_{\widetilde{N}} + 3\ exp\ \mathbb{Z}_{\widetilde{N}})$ $+(2W_B - w_i)\ mult\ \mathbb{Z}_{N^2} + (3W_B - w_i)\ exp\ \mathbb{Z}_{N^2}$
Phase 6	$W_B(2\ \mathbb{G}\ + 5\ \mathbb{Z}_p)$	$W_B\ mult\ \mathbb{Z}_p$ $+2(W_B + w_i\ exp\ \mathbb{G})$ $+2W_B\ \mathcal{O}$
Phase 7	$W_B(2\ \mathbb{G}\ + 5\ \mathbb{Z}_p)$	$(W_B + 2w_i)\ mult\ \mathbb{Z}_p$

Table 4. Complexities of sign protocol with identifiable abort.

design does not pose restrictions in capturing them. The reward functionality Γ_{reward} handles the reward-specific desiderata, while $\mathcal{F}_{pool}$'s first part (Figure 4) covers the requirements for permissioned access and closing of the pool. However, $\mathcal{F}_{pool}$'s handling of stake reallocation and updating of the pool's parameters could be more dynamic, as it currently requires closing and re-creating a pool with the new parameters; a more efficient design is an interesting direction for future research. Additionally, an improvement to the WTSS scheme of Section 5, which would be directly applicable by π_{pool}, could assign a single weighted share to each party, instead of using multiple shares depending on each party's weight.

References

1. Almansa, J.F., Damgård, I., Nielsen, J.B.: Simplified threshold RSA with adaptive and proactive security. In: Vaudenay, S. (ed.) Advances in Cryptology – EUROCRYPT 2006. Lecture Notes in Computer Science, vol. 4004, pp. 593–611. Springer, Heidelberg, Germany, St. Petersburg, Russia (May 28 – Jun 1, 2006). https://doi.org/10.1007/11761679_35
2. Badertscher, C., Gazi, P., Kiayias, A., Russell, A., Zikas, V.: Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In: Lie et al. [37], pp. 913–930. https://doi.org/10.1145/3243734.3243848
3. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. In: Katz and Shacham [30], pp. 324–356. https://doi.org/10.1007/978-3-319-63688-7_11
4. Brünjes, L., Kiayias, A., Koutsoupias, E., Stouka, A.: Reward sharing schemes for stake pools. In: IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020. pp. 256–275. IEEE (2020). https://doi.org/10.1109/EuroSP48549.2020.00024, https://doi.org/10.1109/EuroSP48549.2020.00024
5. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 (2000), https://eprint.iacr.org/2000/067
6. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd Annual Symposium on Foundations of Computer Science. pp. 136–145. IEEE Computer Society Press, Las Vegas, NV, USA (Oct 14–17, 2001). https://doi.org/10.1109/SFCS.2001.959888

7. Canetti, R.: Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239 (2003), https://eprint.iacr.org/2003/239
8. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) TCC 2007: 4th Theory of Cryptography Conference. Lecture Notes in Computer Science, vol. 4392, pp. 61–85. Springer, Heidelberg, Germany, Amsterdam, The Netherlands (Feb 21–24, 2007). https://doi.org/10.1007/978-3-540-70936-7_4
9. Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N., Peled, U.: UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 20: 27th Conference on Computer and Communications Security. pp. 1769–1787. ACM Press, Virtual Event, USA (Nov 9–13, 2020). https://doi.org/10.1145/3372297.3423367
10. Community, E.: Eos.io technical white paper v2 (2018), https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md
11. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed systems - concepts and designs (3. ed.). International computer science series, Addison-Wesley-Longman (2002)
12. Daian, P., Pass, R., Shi, E.: Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In: Goldberg and Moore [24], pp. 23–41. https://doi.org/10.1007/978-3-030-32101-7_2
13. David, B., Gazi, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2018, Part II. Lecture Notes in Computer Science, vol. 10821, pp. 66–98. Springer, Heidelberg, Germany, Tel Aviv, Israel (Apr 29 – May 3, 2018). https://doi.org/10.1007/978-3-319-78375-8_3
14. decred.org: Decred—an autonomous digital currency (2019), https://decred.org
15. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys (CSUR) **36**(4), 372–421 (2004)
16. Dolev, D., Strong, H.R.: Authenticated algorithms for byzantine agreement. SIAM Journal on Computing **12**(4), 656–666 (1983)
17. Fanti, G.C., Kogan, L., Oh, S., Ruan, K., Viswanath, P., Wang, G.: Compounding of wealth in proof-of-stake cryptocurrencies. In: Goldberg and Moore [24], pp. 42–61. https://doi.org/10.1007/978-3-030-32101-7_3
18. Gallagher, P.: Digital signature standard (dss). Federal Information Processing Standards Publications, volume FIPS **186** (2013)
19. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. Cryptology ePrint Archive, Report 2014/765 (2014), https://eprint.iacr.org/2014/765
20. Garay, J.A., Katz, J., Kumaresan, R., Zhou, H.S.: Adaptively secure broadcast, revisited. In: Gavoille, C., Fraigniaud, P. (eds.) 30th ACM Symposium Annual on Principles of Distributed Computing. pp. 179–186. Association for Computing Machinery, San Jose, CA, USA (Jun 6–8, 2011). https://doi.org/10.1145/1993806.1993832
21. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ECDSA with fast trustless setup. In: Lie et al. [37], pp. 1179–1194. https://doi.org/10.1145/3243734.3243859
22. Gennaro, R., Goldfeder, S.: One round threshold ECDSA with identifiable abort. Cryptology ePrint Archive, Report 2020/540 (2020), https://eprint.iacr.org/2020/540

23. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017. pp. 51–68. ACM (2017). https://doi.org/10.1145/3132747.3132757, https://doi.org/10.1145/3132747.3132757
24. Goldberg, I., Moore, T. (eds.): FC 2019: 23rd International Conference on Financial Cryptography and Data Security, Lecture Notes in Computer Science, vol. 11598. Springer, Heidelberg, Germany, Frigate Bay, St. Kitts and Nevis (Feb 18–22, 2019)
25. Goodman, L.: Tezos—a self-amending crypto-ledger white paper (2014)
26. Hirt, M., Zikas, V.: Adaptively secure broadcast. In: Gilbert, H. (ed.) Advances in Cryptology – EUROCRYPT 2010. Lecture Notes in Computer Science, vol. 6110, pp. 466–485. Springer, Heidelberg, Germany, French Riviera (May 30 – Jun 3, 2010). https://doi.org/10.1007/978-3-642-13190-5_24
27. Johnson, B., Laszka, A., Grossklags, J., Vasek, M., Moore, T.: Game-theoretic analysis of DDoS attacks against bitcoin mining pools. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) FC 2014 Workshops. Lecture Notes in Computer Science, vol. 8438, pp. 72–86. Springer, Heidelberg, Germany, Christ Church, Barbados (Mar 7, 2014). https://doi.org/10.1007/978-3-662-44774-1_6
28. Karakostas, D., Kiayias, A., Larangeira, M.: Account management in proof of stake ledgers. In: Galdi, C., Kolesnikov, V. (eds.) SCN 20: 12th International Conference on Security in Communication Networks. Lecture Notes in Computer Science, vol. 12238, pp. 3–23. Springer, Heidelberg, Germany, Amalfi, Italy (Sep 14–16, 2020). https://doi.org/10.1007/978-3-030-57990-6_1
29. Karakostas, D., Kiayias, A., Nasikas, C., Zindros, D.: Cryptocurrency egalitarianism: A quantitative approach. In: Danos, V., Herlihy, M., Potop-Butucaru, M., Prat, J., Piergiovanni, S.T. (eds.) International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2019, May 6-7, 2019, Paris, France. OASIcs, vol. 71, pp. 7:1–7:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/OASIcs.Tokenomics.2019.7, https://doi.org/10.4230/OASIcs.Tokenomics.2019.7
30. Katz, J., Shacham, H. (eds.): Advances in Cryptology – CRYPTO 2017, Part I, Lecture Notes in Computer Science, vol. 10401. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 2017)
31. Kerber, T., Kiayias, A., Kohlweiss, M.: Kachina - foundations of private smart contracts. In: 2021 2021 IEEE 34th Computer Security Foundations Symposium (CSF). pp. 47–62. IEEE Computer Society, Los Alamitos, CA, USA (jun 2021). https://doi.org/10.1109/CSF51468.2021.00002, https://doi.ieeecomputersociety.org/10.1109/CSF51468.2021.00002
32. Kerber, T., Kiayias, A., Kohlweiss, M., Zikas, V.: Ouroboros crypsinous: Privacy-preserving proof-of-stake. In: 2019 IEEE Symposium on Security and Privacy. pp. 157–174. IEEE Computer Society Press, San Francisco, CA, USA (May 19–23, 2019). https://doi.org/10.1109/SP.2019.00063
33. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Katz and Shacham [30], pp. 357–388. https://doi.org/10.1007/978-3-319-63688-7_12
34. Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B.: Enhancing bitcoin security and performance with strong consistency via collective signing. In: Holz, T., Savage, S. (eds.) USENIX Security 2016: 25th USENIX Security Symposium. pp. 279–296. USENIX Association, Austin, TX, USA (Aug 10–12, 2016)

35. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS) **4**(3), 382–401 (1982)
36. Laszka, A., Johnson, B., Grossklags, J.: When bitcoin mining pools run dry - A game-theoretic analysis of the long-term impact of attacks between mining pools. In: Brenner, M., Christin, N., Johnson, B., Rohloff, K. (eds.) FC 2015 Workshops. Lecture Notes in Computer Science, vol. 8976, pp. 63–77. Springer, Heidelberg, Germany, San Juan, Puerto Rico (Jan 30, 2015). https://doi.org/10.1007/978-3-662-48051-9_5
37. Lie, D., Mannan, M., Backes, M., Wang, X. (eds.): ACM CCS 2018: 25th Conference on Computer and Communications Security. ACM Press, Toronto, ON, Canada (Oct 15–19, 2018)
38. Morillo, P., Padró, C., Sáez, G., Villar, J.L.: Weighted threshold secret sharing schemes. Information processing letters **70**(5), 211–216 (1999)
39. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
40. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) Advances in Cryptology – EUROCRYPT'99. Lecture Notes in Computer Science, vol. 1592, pp. 223–238. Springer, Heidelberg, Germany, Prague, Czech Republic (May 2–6, 1999). https://doi.org/10.1007/3-540-48910-X_16
41. Pass, R., Shi, E.: The sleepy model of consensus. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017, Part II. Lecture Notes in Computer Science, vol. 10625, pp. 380–409. Springer, Heidelberg, Germany, Hong Kong, China (Dec 3–7, 2017). https://doi.org/10.1007/978-3-319-70697-9_14
42. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM (JACM) **27**(2), 228–234 (1980)
43. Poupard, G., Stern, J.: Short proofs of knowledge for factoring. In: Imai, H., Zheng, Y. (eds.) PKC 2000: 3rd International Workshop on Theory and Practice in Public Key Cryptography. Lecture Notes in Computer Science, vol. 1751, pp. 147–166. Springer, Heidelberg, Germany, Melbourne, Victoria, Australia (Jan 18–20, 2000). https://doi.org/10.1007/978-3-540-46588-1_11
44. Reed, D.D., Luiselli, J.K.: Temporal Discounting, pp. 1474–1474. Springer US, Boston, MA (2011). https://doi.org/10.1007/978-0-387-79061-9_3162, https://doi.org/10.1007/978-0-387-79061-9_3162
45. Shamir, A.: How to share a secret. Communications of the Association for Computing Machinery **22**(11), 612–613 (Nov 1979)
46. Wallrabenstein, J.R., Clifton, C.: Privacy preserving Tâtonnement - A cryptographic construction of an incentive compatible market. In: Christin, N., Safavi-Naini, R. (eds.) FC 2014: 18th International Conference on Financial Cryptography and Data Security. Lecture Notes in Computer Science, vol. 8437, pp. 399–416. Springer, Heidelberg, Germany, Christ Church, Barbados (Mar 3–7, 2014). https://doi.org/10.1007/978-3-662-45472-5_26
47. Wood, G.: Ethereum yellow paper (2014)