Mithril: Stake-based Threshold Multisignatures

Pyrros Chaidos[1] and Aggelos Kiayias[2]

[1] National & Kapodistrian University of Athens
pchaidos@di.uoa.gr
[2] University of Edinburgh & IOHK
akiayias@inf.ed.ac.uk

Abstract. Stake-based multiparty cryptographic primitives operate in a setting where participants are associated with their stake, security is argued against an adversary that is bounded by the total stake it possesses —as opposed to number of parties— and we are interested in *scalability*, i.e., the complexity of critical operations depends only logarithmically in the number of participants (who are assumed to be numerous).
In this work we put forth a new stake-based primitive, *stake-based threshold multisignatures* (STM, or "Mithril" signatures), which allows the aggregation of individual signatures into a compact multisignature provided the stake that supports a given message exceeds a stake threshold. This is achieved by having for each message a pseudorandomly sampled subset of participants eligible to issue an individual signature; this ensures the scalability of signing, aggregation and verification.
We formalize the primitive in the universal composition setting and propose efficient constructions for STMs. We also showcase that STMs are eminently useful in the cryptocurrency setting by providing two applications: (i) stakeholder decision-making for Proof of Work (PoW) blockchains, specifically, Bitcoin, and (ii) fast bootstrapping for Proof of Stake (PoS) blockchains.

Keywords: Digital Signatures; Blockchains; Universal Composability; Non-Interactive Zero Knowledge

1 Introduction

A wide class of multiparty cryptographic protocols are currently considered in the *stake-based* setting, where a public-key directory of n keys associates each key mvk_i with a real number s_i, — the key's stake. In the stake-based setting, the adversary has a corruption bound expressed in terms of total stake controlled — rather than number of keys or identities — and the complexity metrics of the protocol aim to scale with $\log n$ rather than n.

While any standard "key-based" multiparty protocol can be trivially ported to the stake-based setting by "flattening" out the stake distribution and associating

each unit of stake (aka coin) to a distinct cryptographic key, the resulting constructions are typically extremely inefficient. Motivated by advances in blockchain technology, an array of recent protocol design efforts have focused on the topic of native stake-based design, with prominent examples in the area of consensus protocols, e.g., Algorand [14] and the Ouroboros protocols [32, 30, 16], and more recently secure multiparty computation [5, 15].

Pushing the state of the art forward in this direction, this work puts forth "mithril" or stake-based threshold multisignatures (STM).

- First, in an STM, as in a threshold signature, a quorum of signers is required to engage, in order for a signature to be produced. However, in line with the stake-based setting, that threshold is expressed in terms of stake rather than a number of keys or identities.
- Second, in an STM, as in a multisignature, signers can act independently and sign messages that can be individually verified. When they do sign the same message, their individual signatures can be aggregated as long as they exceed the agreed threshold. The aggregate that can be verified with respect to a global key that represents the whole stakeholder set.
- Third, in an STM, in line with the scalability objective of the stake-based setting, we want the operations of issuing a signature, aggregation of individual signatures and verification to depend logarithmically in n.

Beyond the theoretical interest in designing such a cryptographic scheme, STMs constitute an eminently useful primitive in the setting of cryptocurrencies. Specifically, by associating an STM key to their cryptocurrency account, it is possible for the set of owners of a cryptocurrency to certify any specific message in a collective manner. Observe that all three properties identified above are essential in the cryptocurrency setting. First, by imposing a stake threshold, e.g., 1/2 or 2/3, we ensure that the majority or supermajority of stakeholders endorse the message. Second, by allowing stakeholders to sign in an individually verifiable manner, we allow signed messages to be collected over a public peer-to-peer network while preventing DoS attacks. Third, logarithmic dependency in n, ensures the scalability of the operation even for billions of stakeholders.

STMs can have profound implications in the topic of blockchain governance, (e.g., it is possible for all Bitcoin holders to ratify a particular software upgrade) but also other applications such as fast blockchain bootstrapping of cryptocurrency wallets. Specifically, to articulate the latter application, in a proof-of-stake blockchain like Cardano or Tezos, using an STM, it is possible to certify the state of the ledger efficiently at regular intervals creating certified checkpoints. This can facilitate a fast bootstrapping process for a wallet application joining the system: instead of the wallet acting as a "full node" and processing all ledger transactions to sync up to the recent state, it can "hop" from checkpoint to checkpoint starting from the genesis block (or the most recently known trusted block) until the latest checkpoint is reached from which point it can process transactions normally.

Our contributions. In more details, our contributions are as follows.

Formalization of the Stake-based Threshold Multisignature primitive. The fundamental concept in achieving a scalable STM is to pseudorandomly associate with each message a sufficiently large committee drawn from the stakeholder distribution. Thus, for any message msg, the STM functionality can be thought of as initiating m lotteries and each prospective signer can check to see if it wins it. Here m is a security parameter of the primitive. Each winning ticket can be seen as an eligibility credential allowing the party to create a signature for msg. The probability of winning a ticket is a function of the party's stake calculated in such a way that the party would have the same probability of winning irrespectively of how her stake is organized (e.g., either aggregated in a single public-key or dispersed to many). Eligible parties for a message msg are subsequently capable to create a signature. Finally, once a sufficient number k of signatures are produced these can be aggregated in a public manner. We present our modeling as an ideal functionality in the universal composition (UC) setting.

A scalable instantiation. At a bird's eye view, our construction relies on two basic primitives: a unique multisignature equipped with a regular "dense mapping" and a non-interactive proof system. The mapping is used to issue a number of lottery tickets per message per signer so that winning tickets are publicly verifiable, while the NIZK will collect a sufficient number of winning tickets to a single signature. Assembling these primitives safely so we achieve our objective requires special care. In more detail there are three main ingredients to the full construction.

The first ingredient is a key registration functionality that organizes the participants' stake; to minimize the assumptions placed on the setup of the primitive we assume that key registration functionality is aware of the stake of participants and invites them to register their cryptographic keys. Upon termination of this phase the parties can retrieve those keys and organize them in a Merkle tree (note that this Merkle tree organization can take place as part of a trusted setup and hence need not encumber the parties computationally).

The second ingredient is an eligibility check per message, for which we rely on the uniqueness and unforgeability of the signature scheme, i.e., it is infeasible for an adversary to produce two distinct valid signatures. We apply the mapping on the signature and the lottery index from $\{1, \ldots, m\}$ and we compare the output in a suitable manner with the party's stake; this determines whether the stakeholder wins the particular lottery. In case of a winning ticket, the party communicates the individual signature, index, registration data (and corresponding Merkle tree witness). This makes her winning ticket publicly verifiable.

The third ingredient is using a suitable proof system to facilitate aggregation of the winning tickets into a succinct object. The resulting aggregate signature should be constant size, while we do not want to rely on a structured reference string that, in the blockchain setting, can be hard to generate and maintain. For this reason, in our construction we resort to a bulletproof [11] on a specially crafted statement. While proofs are not constant size, they remain compact and need only an unstructured reference string. The first challenge in applying this strategy is to remove any dependency of the underlying security argument to idealizing any of the hash functions that go into the circuit that is fed to

the bulletproof compiler. The second challenge is to minimize the expensive operations within the bulletproof circuit such as pairings. For that, we rely on multisignature aggregation inside the circuit so that we only need to calculate the inputs to a single pairing check which we can perform externally. We also give a simpler, concatenation-based alternative that produces significantly larger proofs but has lower verification and implementation complexity.

Efficiency Considerations and Applications. We show how our construction can be concretely implemented using elligator squared [42] curves. To maintain the efficiency of the circuit we rely on Poseidon hashes [26] for the Merkle tree. The number of constraints that are needed for the circuit is approximately 2^{21}, and aggregate proof sizes can be as small as 4KB using Bulleproofs. Concatenation based proofs are ca. 300KB in size, but are faster to verify.

In terms of applications we observe that our construction can be readily integrated into standard Bitcoin script to equip all accounts with STM functionality. In particular, using pay-to-script-hash P2SH it is possible to entangle an STM public-key to one's address and then use the Bitcoin blockchain as the key-registration service in our abstract construction above. Subsequently all enabled UTXOs can engage in STM generation.

We also examine the problem of bootstrapping light clients in Proof of Stake (PoS) blockchains. The general challenge in this setting is that the client needs to verify the ledger upon joining the network and that block verification fundamentally depends on stake (so it is cannot be conducted in the same way as an SPV client in the bitcoin setting, that can just count the blocks' aggregate difficulty). As a result a client bootstrapping in the PoS setting, needs to follow the stake as it moves between accounts to be in sync over time with the stakeholder distribution and validate all the blocks. The amount of work to be performed scales linearly with the number of transactions in the ledger which can be extremely large. Using mithril, a different approach can be followed: instead of verifying transactions, the stakeholders can issue checkpoints at regular intervals using an STM signature. The client needs only to verify all checkpoints till the most recent one after which individual blocks and transactions can be verified sequentially. In this way the operation becomes linear in the number of checkpoints instead of linear in the number of transactions. The frequency of the checkpoints can be set to be at regular intervals.

Related Work. Multisignatures, introduced in [28] enables combining multiple signatures of the same message into one. Note that the interesting case is the setting where verification complexity would be sublinear in the number of signers, otherwise one can simply string all signatures together in order to obtain a multi-signature.

In [39] Ristenpart and Yilek demonstrate how proofs of possession can enable more efficient aggregation for BLS-based constructions while avoiding "rogue-key" attacks, in which an adversary may create a malicious key related to an honest one with the goal that the malicious key can be used to sign a multisignature over both keys.

The related but distinct primitive of threshold signatures was introduced in [17]. In a threshold signature, there is a threshold t and a signature can be produced with respect to the group key as long as t shareholders engage. Many threshold signature schemes require a key generation protocol that requires the coordination of the signers over a number of rounds, e.g.,[25],[41]. [13]. Nevertheless it is desirable, especially in the blockchain setting, to have an *ad-hoc* key generation where signers can post their keys in asynchronous fashion and the subgroup that acts for a particular message is determined dynamically.

Threshold signatures and multisignatures were combined in [34] highlighting the properties of traceability in the context of threshold signatures. The concept of accountability, i.e., that the subgroup involved in a multisignature needs to be reliably identified by the verifier in the context of multisignatures was formalized in the form of accountable subgroup multisignatures (AMS) [36].

Ad-hoc threshold multisignatures (ATMS) were put forth in [24]. ATMS is like a threshold signature, in the sense that a quorum of signers need to issue "signature shares" that are subsequently combined. Signature shares however are verifiable as signatures too and key generation is ad-hoc without requiring coordination from participants. This allows a committee to be fixed ahead of time whilst allowing for individual members to abstain or be unavailable for some operations. In contrast, our notion of a "threshold" is predicated by the stake held by each users and additionally involves random elligibility sampling to keep participation requirements manageable. Essentially, whereas in an ATMS scheme selecting a committee is an external operation, in STM it is (implicitly) performed internally. This is beneficial to security (as there is no need to identify committee members) as well as liveness: a (partly) inactive committee stops progress in an ATMS scheme, but an STM scheme can recover by signing an alternative message (as eligibility is pseudorandomly redistributed per message).

The importance of forward-security in the context of blockchain protocols has already been higlighted in earlier work in consensus protocols including [14], [16] and [19]. Forward-security is not essential for all STM applications hence we do not incorporate it as fundamental property of the primitive - we examine the implications of active attacks and forward security type mitigations in Section 4.2.

Blockchains and Proof of stake. In terms of client bootstrapping, Proof of work blockchains admit simple solutions like SPV, where bootstrapping can be performed by verifying only the headers of the chain [38]. Further optimizations such as Non-interactive proofs of proof-of-work (NIPoPoWs) [31] and flyclient [12] drastically reduce the amount of headers required by attaching additional significance to blocks with a specific, rare property. This critically hinges on the ability to verify headers without the need to establish a stakeholder distribution.

Turning to PoS blockchains, the works of [1, 23] are somewhat orthogonal to our work: they describe how a single user can prove eligibility while maintaining privacy, whilst we describe how to efficiently demonstrate eligibility over multiple users. However, the technical toolset is similar as is the main hurdle: efficiently proving correct evaluation of a verifiable random function. A significant obstacle in that is the use of random oracles in such functions: a proof system based on

circuits needs to instantiate the oracle to define the verification circuit, which implies the complete construction no longer operates in the random oracle model.

A verifiable random function (VRF) [37, 18] allows one evaluate a random function f on a specific point x and prove correctness of that evaluation, without allowing others to evaluate the same function in other points. Security requires that without knowledge of the private evaluation key, or a proof of correctness $y = f(x)$ is indistinguishable from random. The weaker notion of a unique signature, or equivalently a verifiable unpredictable function (VUF), requires that adversaries are unable to guess y (but may be able to distinguish it from random). We use a public mapping M to apply a regular distribution to signatures i.e., given a signature σ on x, it holds that $y = f(x) \stackrel{\mathsf{def}}{=} M(\sigma, x)$ is pseudorandom without knowledge of the verification key. We then expand M to accept an additional evaluation parameter t such that $f(x, t')$ may be determined from $f(x, t)$ but $f(x', t)$ remains unpredictable for all $x' \neq x, t$. This relation between evaluations over the same x is crucial for the efficiency of our construction that relies on a batch verification step.

Similar to [16], we rely on Elligator to "convert" a random group element on an elliptic curve to a random field element. Due to our setting [27], we are unable to directly use the base version and rely on Elligator squared [42] with the additional contributions of Wahby and Boneh [43].

Vault [33] uses a construction similar to ours as a component in an efficient bootstrapping and storage solution for Algorand. Their construction does not utilize multisignatures, as multisignatures alone do eliminate the linear size dependency on committee size: the VRF and Merkle tree checks need to be aggregated as well. We opt to use a dense mapping, a notion similar to a VUF to make aggregation possible, which gives us greater flexibility by means of size-time tradeoffs in choosing the appropriate proof system.

Plumo [21] uses a two layer solution tailored to blockchain bootstrapping, where one layer proves epoch transitions and a the other aggregates over multiple epochs. Their system is highly efficient, but requires stronger setup assumptionsthan ours.

2 Preliminaries

2.1 Notation

We use λ as the security parameter. When S is a set, the assignment operator $x \leftarrow S$ stands for x being sampled from the set S uniformly at random. We use bold characters to denote vectors of variables i.e $\boldsymbol{b} := (b_1, \ldots, b_n)$.

2.2 Group Setting

We require a group $\mathbb{G}_H$ of order p, so that there exists an embedded pairing-friendly elliptic curve E on $\mathbb{F}_p$, forming groups $\mathbb{G}_1, \mathbb{G}_2$ of order q, with pairing function $e : G_1 \times G_2 \rightarrow \mathbb{G}_T$. We use g_1, g_2 to refer to generators of $\mathbb{G}_1, \mathbb{G}_2$

respectively. We optionally require that the structure of E is compatible with the Elligator [6] or Elligator squared [42] representation functions.

We require E to be pairing-friendly due to our choice of multi-signagure scheme. Compatibility with Elligator depends on our choice of dense mapping.

Definition 1 (The Discrete log Problem). *For a group $\mathbb{G} = \langle g \rangle$ of order q, and an adversary $\mathcal{A}$ we define $Adv^{dl}_{\mathbb{G}}$ as:*

$$\Pr\left[a \leftarrow \mathbb{Z}_q; h \leftarrow g^a : a \leftarrow \mathcal{A}(h)\right]$$

Definition 2 (The Discrete log Assumption). *We assume $Adv^{dl}_{\mathbb{G}}$ is negligible for all PPT $\mathcal{A}$ on $\mathbb{G}_H$, $\mathbb{G}_1$, $\mathbb{G}_2$.*

Definition 3 (The co-Computational Diffie-Hellman Problem). *For two groups $\mathbb{G}_1 = \langle g_1 \rangle, \mathbb{G}_2 = \langle g_2 \rangle$ of order q, and an adversary $\mathcal{A}$ we define $Adv^{co-CDH}_{\mathbb{G}_1,\mathbb{G}_2}$ as:*

$$\Pr\left[a, b \leftarrow {\mathbb{Z}_q}^2; h \leftarrow g_1^a; t_1 \leftarrow g_1^b; t_2 \leftarrow g_2^b : g_1^{ab} \leftarrow \mathcal{A}(h, t_1, t_2)\right]$$

Definition 4 (The co-CDH Assumption). *We assume $Adv^{co-CDH}_{\mathbb{G}_1,\mathbb{G}_2}$ is negligible for all PPT $\mathcal{A}$ on $\mathbb{G}_1, \mathbb{G}_2$.*

We can further strengthen the above assumption, by allowing $\mathcal{A}$ to run in super-polynomial, but still sub-exponential time. This can allow for higher efficiency in our construction, through the use of a complexity leveraging argument, but is not necessary to prove security.

Definition 5 (The leveraged co-CDH Assumption). *We assume $Adv^{co-CDH}_{\mathbb{G}_1,\mathbb{G}_2}$ is negligible on $\mathbb{G}_1, \mathbb{G}_2$ for all adversaries $\mathcal{A}$ running in time $O(\lambda^{\log \lambda})$.*

Common setup. We use $\mathsf{Setup}(1^\lambda)$ to refer to the group generator function which generates a group setting with the above requirements.

$\mathsf{Setup}(1^\lambda)$ generates groups $\mathbb{G}_1 = \langle g_1 \rangle, \mathbb{G}_2 = \langle g_2 \rangle$ of order q, as well as $\mathbb{G}_H$ of order p and returns system parameters $\mathsf{Param} = (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, q, \mathbb{G}_H, g_h, p)$.

2.3 Hash functions

We use the Poseidon hash [26] on $\mathbb{F}_p$ as the hash function H_p, used to produce the Merkle tree as it satisfies collision resistance and is efficient to implement inside proofs of knowledge.

We also need hash function $H_{\mathbb{G}_1} : \{0,1\}^* \to \mathbb{G}_1$, $H_q : \{0,1\}^* \to \mathbb{Z}_q$ modeled as random oracles, producing group elements in the corresponding groups for use with our our multisignature scheme and mapping. We note that $H_{\mathbb{G}_1}, H_q$ are not evaluated inside the proof of knowledge, allowing us to study the security of both constructions under the random oracle model [3] with no hindrance to the proof. This is relevant, as [1] point out: once the hash function has been instantiated and concretely represented (e.g. as a circuit) in order to construct the appropriate statement proof system, we can no longer invoke the random oracle model in the security analysis.

Merkle trees A Merkle tree is a well used data structure based on hash functions that allows one to represent N items [3] of arbitrary size by one hash value. Beyond that, it is efficient to verify that a value v exists within a Merkle Tree T, by providing a path p which consists of the position i of N in the tree, as well as the hashes of the siblings of i and the siblings of its parents.

- $\mathsf{MT.Create}(\boldsymbol{v})$. Parse $\boldsymbol{v}$ as a vector v_i of length N. Create an empty binary tree with N leaves. Label each leaf l_i with the hash of the corresponding value $H_p(v_i)$. For each level of the tree, label each node z with the hash $H_p(x, y)$ of the labels of its children x, y. Return the label T of the root.
- $\mathsf{MT.Check}(T, N, v, i, \boldsymbol{p})$. Parse $\boldsymbol{p}$ as a vector p_j of length $\log_2(N)$. Let i_k be the k-th least significant digit of i in binary. Let $h_0 \leftarrow H_p(v_i)$. for $k = 1$ to $\log_2(N)$, let $h_k \leftarrow H_p(h_{k-1}, p_{k-1})$ if i_k is 0 and $h_k \leftarrow H_p(p_{k-1}, h_{k-1})$ if it is 1. Return 1 if $h_{log_2(N)} = T$ and 0 otherwise.

For simplicity, we write that $v \in T$, for a fixed value of N if there exists an index i and path $\boldsymbol{p}$ such that $\mathsf{MT.Check}(T, N, v, i, \boldsymbol{p})$ is 1. In this work we will rely on the fact that Merkle trees are binding in the following sense:

Lemma 1. *If for a Merkle tree T, N there exist i, $v \neq v'$, and $\boldsymbol{p}, p'$ such that $\mathsf{MT.Check}(T, N, v, i, \boldsymbol{p}) = \mathsf{MT.Check}(T, N, v', i, \boldsymbol{p}') = 1$, we can extract a collision for H_p.*

Proof. Following the calculation of $\mathsf{MT.Check}$, we have $h_0 \neq h'_0$ unless v, v' are a collision. Furthermore, we know that $h_{log_2(N)} = h'_{log_2(N)}$. Thus, there must exist a minimal k such that $h_k \neq h'_k$ but $h_{k+1} = h'_{k+1}$. Thus, we find that $(h_k, p_k), (h'_k, p'_k)$ is a colission when i_k is 0, and $(p_k, h_k), (p'_k, h'_k)$ when it is not.

2.4 Multi-Signature Scheme

Multi-signature schemes [8] are a natural extension to the concept of a digital signature, by introducing the concept of aggregation for keys as well as signatures. In this work we will limit ourselves to aggregating signatures over the same message. Given any digital signature scheme, we are able to aggregate keys and signatures via concatenation, and naturally extend the verification algorithm to account for this. Practical multi-signature schemes aim to implement aggregation efficiently whilst still maintaining security.

We use a variant of MSP-PoP, a multisignature based on BLS with proofs of possession as described in [8, 39]. We further extend the proof of possession with an additional element as our security context is slightly different: standard security definitions of multi signature unforgeability require that the challenger provides a signing oracle only for one designated honest user, and in addition, it needs to be able to calculate signatures for every other user on a pre-selected point. In the proof of lemma 8 we will additionally need to be able to calculate arbitrary signatures on potentially malicious users on any message. This can be

[3] For ease of exposition, we assume N to be a power of 2.

solved by either requiring an isomorphism from $\mathbb{G}_2$ to $\mathbb{G}_1$ as in [39], or in our case by adding the equivalent image to the proof of possession.

- $\mathsf{MSP.Gen}(\mathsf{Param})$: $sk \leftarrow \mathbb{Z}_q; mvk \leftarrow g_2^x$;
  $\kappa_1 \leftarrow H_{\mathbb{G}_1}(\text{“PoP”}\|mvk)^x; \kappa_2 \leftarrow g_1^x$. Return secret key sk, verification key mvk and proof or possession $\boldsymbol{\kappa} = (\kappa_1, \kappa_2)$
- $\mathsf{MSP.Check}(mvk, \boldsymbol{\kappa})$ If $e(\kappa_1, g_2) = e(H_{\mathbb{G}_1}(\text{“PoP”}\|mvk), mvk)$ and $e(g_1, mvk) = e(\kappa_2, g_2)$ are both true, return 1, otherwise return 0.
- $\mathsf{MSP.Sig}(sk, msg)$ Return $\sigma \leftarrow H_{\mathbb{G}_1}(\text{“M”}\|msg)^x$.
- $\mathsf{MSP.Ver}(msg, mvk, \sigma)$ Return 1 if $e(\sigma, g_2) = e(H_{\mathbb{G}_1}(\text{“M”}\|msg), mvk)$. Otherwise return 0.
- $\mathsf{MSP.AKey}(\mathbf{mvk})$ Takes a vector $\mathbf{mvk}$ of (previously checked) verification keys and returns an intermediate aggregate public key $ivk = \prod mvk_i$.
- $\mathsf{MSP.Aggr}(msg, \boldsymbol{\sigma})$. Takes as input a vector of signatures $\boldsymbol{\sigma}$ and returns $\mu \leftarrow \prod_1^d \sigma_i$.
- $\mathsf{MSP.AVer}(msg, ivk, \mu)$ returns $\mathsf{MSP.Ver}(msg, ivk, \mu)$.

The MSP scheme has been shown to be *complete* and *unforgeable* in [39]. The signature scheme is also *unique* as the signing and verification operations are deterministic. It is impossible for any msg, mvk to have $\sigma \neq \sigma'$ so that $\mathsf{MSP.Ver}(msg, mvk, \sigma) = \mathsf{MSP.Ver}(msg, mvk, \sigma') = 1$

2.5 Dense Mappings for Unique Signatures

The works of [42, 6] show how one can map a point on an elliptic curve to a string indistinguishable from uniformly random. Given such a mapping we would be able to use a signature scheme with unique signatures as a regularly distributed verifiable unpredictable function (VUF).

Definition 6. *A deterministic function $M : \mathbb{G}_1 \rightarrow \mathbb{Z}_p \cup \{\perp\}$ is a dense mapping if, for some negligible ϵ, it holds that for any $y \in \mathbb{Z}_p$, $|Pr[M(x) = y | M(x) \neq \perp] - 1/p| \leq \epsilon$ and $Pr[M(x) \neq \perp]$ is non-negligible, where x is uniformly distributed over $\mathbb{G}_1$.*

Given a family $M_{msg,\mathsf{index}}$ of dense mappings indexed by index, we can add a new operation to a multisignature scheme as follows.

- $\mathsf{MSP.Eval}(msg, \mathsf{index}, \sigma)$ Return $ev \leftarrow M_{msg,\mathsf{index}}(\sigma)$.

Being able to deterministically attach a regularly-sampled value to signatures enables us to flag a small subset of signatures as eligible by requiring their values under the mapping for a sequence of indexes to be under a given threshold.

In Section 6 we show how to construct a dense mapping $M^E_{msg,\mathsf{index}}(\sigma)$ based on Elligator Squared, which avoids oracle calls on user-specific data i.e. we explicitly avoid hashing σ to sidestep soundness issues in circuit-based proofs.

For the concatenation proof system PS^C in section 2.9 we are able to use a random oracle $H : \{0,1\}^* \rightarrow \mathbb{Z}_p$ to implement the mapping as: $M^R_{msg,\mathsf{index}}(\sigma) := H(\text{“map”}\|msg\|\mathsf{index}\|\sigma)$.

2.6 Weighting Function

Looking forward, we will use the concept of weights to randomly assign eligibility to participants. In this way, a small number of participants can be considered to be a random (and therefore somewhat representative) sample of a large group. A straightforward approach would be to use weights directly, potentially with a scaling factor to the required level of participation.

However, this introduces pitfalls in the resulting distribution: basic probability indicates that winning a coin toss ($p_c = \frac{1}{2}$) is not equivalent to guessing a die roll in 3 tries (each with $p_d = \frac{1}{6}$, for a success probability of $1 - (5/6)^3$). The same problem was faced in [16], and we will opt to follow their solution in this work:

We will use the function $\phi(w) = 1 - (1 - f)^w$ to assign success probabilities to weights $w \in [0, 1]$. The value $f = \phi(1)$ is a tuning parameter, representing the success probability assigned to the maximum weight.

The end result is to make the probability of success for a given party irrespective of the exact distribution in virtual identities: i.e. an adversary controlling weight w has the same chance of success if she keeps the weight under a single identity or splits it in various ways. The same property is also useful in regards to honest parties, where behaviour may be more unpredictable.

2.7 Noninteractive Proof Systems

In our construction, we will use a proof system to demonstrate that given a merkle tree AVK, multiple eligible users have signed a particular message. Eligibility here means that a user's information is contained in AVK, and that the user's signature evaluation is valid with regards to their stake. For this we rely on bulletproofs [11], with some additional operations performed in the open.

The relation $\mathcal{R}_{avk}$ Our proof system operates on language $\mathcal{L}_{avk}$, i.e we prove knowledge of a witness w such that statement x holds, i.e. $R_{avk}(x, w) = 1$. Concretely, our statement is of the form $x = (\mathsf{AVK}, ivk, \mu, msg)$ and our witness is of the form $w = (mvk_i, \mathsf{stake}_i, \boldsymbol{p}_i, ev_i, \sigma_i, \mathsf{index}_i)$ for $i = 1 \dots k$. The relation R_{avk} is parametrized on $N, m, k, \phi()$, which are public information. $R_{avk}(x, w) = 1$ if and only if the following hold:

- $ivk = \prod_1^k \mathbf{mvk}_i$.
- $\mu = \prod_1^d \sigma_i$.
- $\forall i : \mathsf{index}_i \leq m$.
- $\forall i \neq j : \mathsf{index}_i \neq \mathsf{index}_j$.
- For $i = 1..k$: $(mvk_i, \mathsf{stake}_i)$ lies in Merkle tree AVK, N following path $\boldsymbol{p}_i$.
- For $i = 1..k$: $\mathsf{MSP.Eval}(msg, \mathsf{index}_i, \sigma_i) = ev_i$
- For $i = 1..k$: $ev_i \leq \phi(\mathsf{stake}_i)$

We will use the following notation to refer to the complete setup, prover and verifier algorithms as:

$\mathsf{PS.RS}(1^\lambda)$, with output a reference string $\mathsf{PS.P}$.
$\mathsf{PS.P}(\mathsf{PS.RS}, x, w)$, with output a proof π.
$\mathsf{PS.V}(\mathsf{PS.RS}, x, \pi)$, with output 0 or 1.

We will propose two constructions: one based on bulletproofs which may be expanded upon using Halo, and a simpler proof system based on concatenation (i.e releasing the witness).

2.8 A proof system based on bulletproofs

We will use standard notation to refer to the bulletproof reference string setup, prover and verifier algorithms as $\mathsf{BP.RS} \leftarrow \mathsf{BP.Setup}(\mathsf{Param})$, $\pi_C \leftarrow \mathsf{BP.P}(\mathsf{BP.RS}, C, x, w)$, $0/1 \leftarrow \mathsf{BP.V}(\mathsf{BP.RS}, C, x, \pi_C)$, where C, x, w refer to the relation cirquit, statement and witness respectively.

Avoiding random oracle calls. We thus need to represent the relation we will be proving, R_{avk} as a circuit, which can be problematic as we model $H_q, H_{\mathbb{G}_1}$ as random oracles. Fortunately, this is simple to overcome. As msg is part of the statement, and the maximum index m is a public parameter, it is simple to precalculate the H_q values used inside the mapping M^E as well as those used in the representation function. This enables relation R_{avk} to be compiled as a circuit without preventing $H_{\mathbb{G}_1}$ or H_q from being modelled as a random oracle: $H_{\mathbb{G}_1}$ and H_q are never evaluated inside the circuit.

Avoiding rewinding. Bulletproofs are *complete*, *zero-knowledge* and have the *witness-extended emulation* property, a generalization of knowledge soundness. As Universal Composability does not allow rewinding the environment, we are unable to directly invoke witness-extended emulation in our security proofs.

At the same time, invoking standard soundness is potentially vacuous as the possibility of collisions in the Merkle tree implies that it is hard to determine if any particular statement x is false (i.e there exists no witness w for it). Consider a trivial tree AVK, containing the public keys and stake $(mvk_0, \mathsf{stake}_0)$ of only a single user P_0, who is not eligible to sign message m. It is likely, that there exists a different set of public keys (mvk_0, stake') so that (1) they hash to the same value and (2) the second keyset is eligible for m –with multiple users there also exists a degree of freedom in mvk.

To overcome this, we will instead rely on an intermediate security notion, where we will "disallow" proofs of a particular set of statements. Informally, we say that a statement is *contradictory*, if a witness for it would contradict our existing knowledge.

Consider a predicate $Q(y, w)$ and a function $G(\cdot)$.

We are interested in the language $\mathcal{L} = \{x | \exists y, w : x = G(y) \text{ and } Q(y, w) = 1\}$ The reason we are interested in this language for instance is because G(y) can be much shorter in length than y.

In general, a statement x is *contradictory* with respect to information y in $G^{-1}(x)$, if for all $w : Q(y, w) = 0$. We can then easily show the following lemma:

Lemma 2. *. Suppose we have a proof π for a statement x, and y is in $G^{-1}(x)$ Then either the statement x is not contradictory w.r.t. y or there exists some $y' \neq y$, such that $G(y') = G(y)$.*

To apply the above in our setting: consider y is a stakeholder distribution and G is any function that creates a merkle tree root out of it and aggregates a subset of those keys that satisfy the lottery winning property for a given message.

Then, the witness w contains Merkle tree witnesses, signatures and evaluations that establish that there is a set of lottery winning keys. The predicate Q verifies those properties, w.r.t. y.

Now if we get a bulletproof for x, this means that either x is not contradictory w.r.t. y, or that there is another stakeholder distribution y' with $G(y') = G(y)$. In this latter case, we should get a collision against the $\mathsf{MT.Create}(\boldsymbol{v})$ construction.

Contradictions for $\mathcal{R}_{avk}$ For $\mathcal{R}_{avk}$, given $N, m, k, \phi()$, we say that statement $x = (\mathsf{AVK}, ivk, \mu, msg)$ is *contradictory* w.r.t. information $(mvk_i, \mathsf{stake}_i)$ for $i = 1 \dots N$ and $(ev_{i,k}, \sigma_i)$, if (1) $\mathsf{AVK} = \mathsf{MT.Create}((mvk_i, \mathsf{stake}_i))$ for $i = 1 \dots N$, (2) $ev_{i,t} = \mathsf{MSP.Eval}(msg, t, \sigma_i)$ for $i = 1 \dots N$, $t = 1 \dots m$, and (3) there exist no indexes p_j, t_j for $j = 0 \dots k-1$ such that:

- $ivk = \prod_1^k \mathbf{mvk}_{p_j}$.
- $\forall i \neq j : s_i \neq s_j$.
- For $i = 1..k$: $ev_{p_j,t_j} \leq \phi(\mathsf{stake}_{p_j})$

Using witness extended emulation, we can prove that:

Lemma 3 (Contradiction Soundness). *For any $N, m, k, \phi()$, any polynomial time P^*, and given information $(mvk_i, \mathsf{stake}_i)$ for $i = 1 \dots N$ and $(ev_{i,k}, \sigma_i)$ such that $ev_{i,t} = \mathsf{MSP.Eval}(msg, t, \sigma_i)$ for $i = 1 \dots N$, $t = 1 \dots m$, we have that for any* contradictory *statement x, the following probability is negligible.*

$$Pr\left[\sigma \leftarrow \mathsf{BP.Setup}(1^\lambda), \pi^* \leftarrow \mathsf{P}^*(\sigma, x) : \mathsf{BP.V}(\sigma, x, \pi^*) = 1\right]$$

Proof (Sketch). If P^* succeeds with non-negligible probability, we can use the Witness-Extended Emulation extractor to obtain a witness w with good probability in expected polynomial time. Given our information $(mvk_i, \mathsf{stake}_i)$, $(ev_{i,k}, \sigma_i)$ and witness w, we obtain a collision for H_p.

We will use the following notation to refer to the complete setup, prover and verifier algorithms as:

$\mathsf{PS}^B.\mathsf{RS}(1^\lambda)$: Return $\mathsf{BP.Setup}(1^\lambda)$
$\mathsf{PS}^B.\mathsf{P}(\mathsf{PS.RS}, x, w)$: Return $\mathsf{BP.P}(\mathsf{PS}^B.\mathsf{RS}, C_{avk}, x, w)$
$\mathsf{PS}^B.\mathsf{V}(\mathsf{PS.RS}, x, \pi)$: Return $\mathsf{BP.V}(\mathsf{PS}^B.\mathsf{RS}, C_{avk}, x, \pi)$

2.9 A concatenation based proof system

The concatenation-based proof system PS^C consists of releasing the witness w and letting the verifier check if $R_{avk}(x, w) = 1$. Concretely, we have:

$\mathsf{PS}^C.\mathsf{RS}(1^\lambda)$: Return $\perp$
$\mathsf{PS}^C.\mathsf{P}(\mathsf{PS}^C.\mathsf{RS}, x, w)$: Return w
$\mathsf{PS}^C.\mathsf{V}(\mathsf{PS}^C.\mathsf{RS}, x, \pi)$: Return $\mathcal{R}_{avk}(x, w)$

We can trivially prove contradiction soundness as with PS^B.

Lemma 4 (Contradiction Soundness). *For any $N, m, k, \phi()$, any polynomial time P^*, and given information $(mvk_i, \mathsf{stake}_i)$ for $i = 1 \dots N$ and $(ev_{i,k}, \sigma_i)$ such that $ev_{i,t} = \mathsf{MSP.Eval}(msg, t, \sigma_i)$ for $i = 1 \dots N$, $t = 1 \dots m$, we have that for any* contradictory *statement x, the following probability is negligible:*

$$Pr\left[\pi^* \leftarrow \mathsf{P}^*(\perp, x) : \mathsf{PS}^C.\mathsf{V}(\perp, x, \pi^*) = 1\right]$$

Utilizing Oracle calls As the previous proof systems rely on partly representing $\mathcal{R}_{avk}$ as a circuit, care must be taken to avoid oracle calls inside the circuit itself. In this instantiation however, there is no such restriction. As such, we are free to use M^R as the dense mapping in $\mathsf{MSP.Eval}$.

3 Ideal Functionality for Stake Based Threshold Multisignatures

We will now describe a stake based threshold multisignatures functionality similar to the PoS Anonymous Selection of [1].

The functionality maintains a list $\mathcal{L}$ of signatures produced by itself, and a list $\mathcal{E}$ storing the eligibility of the various parties. The functionality operates on a fixed player list $\mathcal{P} = (P_i, \mathsf{stake}_i)$, where $|\mathcal{P}| = n$, a scaling function $\phi(w)$, security parameter $m \geq log^2\lambda$ and quorum parameter $k = m \cdot \phi(\frac{1}{2} + a)$. The functionality operates on a static corruption model where the adversary is allowed to corrupt up to $\frac{1}{2} - a$ of the total stake.

The functionality operates by sampling eligibility over m indices. Users are made eligible in proportion to their stake and independently of each other. Producing an aggregate signature requires individual signatures over k different indices. The functionality operates in two phases. It starts in the initialisation phase which we present in Figure 1. The decision to move to the operation phase, presented in Figure 2 is left to the adversary.

A trivial realization using concatenation. It is simple to see that if we assume uniform stake distribution, we can realise the above using only signature schemes. We set $k = n = m$, and fix the eligibility function to assign $\mathcal{E}(msg, P_i, \mathsf{index}) = 1$ iff $i = \mathsf{index}$ and 0 otherwise. CreateSig is implemented by signing, with the addition that verification only accepts signatures for *index* i from user P_i.

Aggregate is implemented by concatenating signatures and signer identities. VerifyAggregatethen consists of parsing, and counting the number of valid signatures.

While simple, the above protocol produces aggregate signature with size linear in the number of users which is cost-prohibitive in practice. Assuming uniform stake is also problematic in general. One could argue that a user holding s units of stake could be simulated by s users each holding 1 unit, but this only exacerbates the size issue. In the next section we will expand our treatment to cover the more general case, and use dense mappings as a form of lottery so that only a limited number of stakeholders need to participate at any one time.

4 A Stake Based Threshold Multisignature scheme

We present a protocol $\Pi.\mathsf{STM}$ realizing $\mathcal{F}_{\mathsf{STM}}^{\phi}(\mathcal{P}, m, k)$ in the $\mathcal{F}_{\mathsf{RS}}(\mathcal{P}), \mathcal{F}_{\mathsf{Kr}}^{\psi_0}(\mathcal{P})$-hybrid model. As with the functionality, the protocol operates in two phases. The initialisation phase is presented in Fig. 5 and the operation phase in Fig. 6. The functionality operates on a fixed player list $\mathcal{P} = (P_i, \mathsf{stake}_i)$, where $|\mathcal{P}| = n$, a scaling function $\phi(w)$, security parameter $m \geq \log^2 \lambda$ and quorum parameter $k = m \cdot \phi(\frac{1}{2} + a)$, where $\psi_0(mvk, \boldsymbol{\kappa}) := \mathsf{MSP.Check}(mvk, \boldsymbol{\kappa})$.

Our scheme requires two main components: a multi-signature scheme equipped with a dense mapping, and a proof system to produce proofs of multiple signatures with specific mapping constraints, i.e each signature must map to a value smaller than the target value implied by the signer's stake. The simplest option would be to construct aggregate proofs by simply concatenating individual signatures. This allows for simple and efficient choices in the other parameters but produces a

The STM functionality $\mathcal{F}_{\mathsf{STM}}^{\phi}(\mathcal{P}, m, k)$. Initialisation phase

$\mathcal{F}_{\mathsf{Kr}}^{\psi}(\mathcal{P})$ initializes the variable Allow to 1 and proceeds as follows:

- Upon receiving (Register, sid) on behalf of party P_i:
  1. If Allow is 0, $P_i \notin \mathcal{P}$, or $K(P_i)$ is already defined, ignore the request.
  2. Otherwise, set $K(P_i) = 1$ send (Registered, sid, P_i) to $\mathcal{A}$ and output (Registered, sid) to P_i.
- Upon receiving (Start, sid) on behalf of the adversary $\mathcal{A}$:
  1. Set Allow to 0.

Fig. 1. The Stake Based Threshold Multisignatures functionality $\mathcal{F}_{\mathsf{STM}}^{\phi}(\mathcal{P}, m, k)$ in the Initialisation phase interacting with the adversary $\mathcal{A}$.

The STM functionality $\mathcal{F}_{\mathsf{STM}}^{\phi}(\mathcal{P}, m, k)$, operation phase.

- Upon receiving $(\mathsf{EligibilityCheck}, sid, msg, \mathsf{index})$ from a party P_i:
  1. if $K(P_i)$ is undefined, or $P_i \notin \mathcal{P}$ ignore the request.
  2. If $\mathsf{flag}(msg)$ is undefined, send $(\mathsf{EligibilityCheck}, sid, msg, \mathcal{P})$ to $\mathcal{A}$.
  3. On receiving $(\mathsf{Eligible}, sid, msg, \mathcal{B}, t)$ parse $\mathcal{B}$ as a $n \times m$ bit matrix and let $\mathcal{E}(msg, P_i, \mathsf{index}) \leftarrow \mathcal{B}(i, \mathsf{index})$, and let $\mathsf{flag}(msg) \leftarrow 1$.
  4. If, $\mathcal{B}$ assigns eligibility to corrupted users on k or more indices, abort.
  5. Output $(\mathsf{EligibilityCheck}, sid, \mathcal{E}(msg, P_i, \mathsf{index}))$ to P_i.
- Upon receiving $(\mathsf{CreateSig}, sid, msg, \mathsf{index})$ from a party P_i:
  1. if $K(P_i)$ is undefined, ignore the request.
  2. If $\phi(\mathsf{tag})$ is empty, send $(\mathsf{Declined}, sid, msg)$ to P_i. Otherwise, check $\mathcal{E}(msg, P_i, \mathsf{index})$. If it is 0, send $(\mathsf{Declined}, sid, msg)$ to P_i. Otherwise if it is 1, send $(\mathsf{Prove}, sid, P_i, msg, \mathsf{index})$ to $\mathcal{A}$.
  3. When receiving $(\mathsf{Done}, sid, P_i, \psi, msg, \mathsf{index})$ from $\mathcal{A}$, let $\pi = \psi$, store $(P_i, \pi, msg, \mathsf{index})$ in $\mathcal{L}$. Send $(\mathsf{Proof}, sid, \pi, msg, \mathsf{index})$ to P_i.
- Upon receiving $(\mathsf{Verify}, sid, P_i, \pi, msg, \mathsf{index})$ from a party P':
  1. If $K(P_i)$ is undefined, ignore the request.
  2. If $(P_i, \pi, msg, \mathsf{index}) \in \mathcal{L}$, output $(\mathsf{Verified}, sid, (P_i, \pi, msg, \mathsf{index}), 1)$ to P'.
  3. Else, if $\mathcal{E}(msg, P_i, \mathsf{index})$ is 0 or P_i is honest, send $(\mathsf{Verified}, sid, (P_i, \pi, msg, \mathsf{index}), 0)$ to P'.
  4. Else send $(\mathsf{Verify}, sid, (P_i, \pi, msg))$ to $\mathcal{A}$, and wait for $(\mathsf{Verified}, sid, (\pi, msg), v)$ from $\mathcal{A}$. If v is 1 store $(P_i, \pi, msg, \mathsf{index})$ in $\mathcal{L}$ and reply $(\mathsf{Verified}, sid, (P_i, \pi, msg, \mathsf{index}), 1)$ to P'.
  5. Otherwise send $(\mathsf{Verified}, sid, (P_i, \pi, msg, \mathsf{index}), 0)$ to P'.
- Upon receiving $(\mathsf{Aggregate}, sid, \boldsymbol{P}, \boldsymbol{\pi}, \mathsf{index}, msg)$ from a party P' :
  1. Parse $\boldsymbol{P}, \boldsymbol{\pi}, \mathsf{index}$ as vectors of length k containing $P_i, \pi_i, \mathsf{index}_i$.
  2. If $K(P_i)$ is undefined for any i, ignore the request.
     Run $(\mathsf{Verify}, sid, P_i, \pi_i, msg, \mathsf{index}_i)$ for each i.
  3. If any produce 0, or if $\mathsf{index}_i = \mathsf{index}_j$ for $i \neq j$, reply $(\mathsf{Aggregation}, sid, (\boldsymbol{P}, \boldsymbol{\pi}, msg), 0)$.
  4. Otherwise, send $(\mathsf{Aggr}, sid, \boldsymbol{P}, \boldsymbol{\pi}, \mathsf{index}, msg)$ to $\mathcal{A}$.
  5. When receiving $(\mathsf{AggrDone}, sid, \boldsymbol{P}, \boldsymbol{\pi}, \mathsf{index}, \rho, msg)$ from $\mathcal{A}$, let $\tau = \rho$, store (m, τ, msg) in $\mathcal{L}$.
  6. Send $(\mathsf{Aggr}, \tau, \boldsymbol{P}, \boldsymbol{\pi}, msg)$ to P'.
- Upon receiving $(\mathsf{VerifyAggregate}, sid, \tau, msg)$ from a party P' :
  1. If (τ, msg) exists in $\mathcal{L}$ send $(\mathsf{Verified}, sid, m, \tau, msg), 1)$ to P'.
  2. Else, send $(\mathsf{AVerify}, sid, (\tau, msg))$ to $\mathcal{A}$, and wait for $(\mathsf{Verified}, sid, (\tau, msg), v)$ from $\mathcal{A}$.
  3. If $v = 1$, count the number of indexes with either (1) a previously produced signature for msg in $\mathcal{L}$ or (2) a corrupted player eligible to sign. If the total is k or more, store (τ, msg) in $\mathcal{L}$ and output $(\mathsf{Verified}, sid, (m, \tau, msg), 1)$ to P'.
  4. Otherwise, output $(\mathsf{Verified}, sid, (m, \tau, msg), 0)$ to P'.

Fig. 2. The Stake Based Threshold Multisignatures functionality on the operation phase $\mathcal{F}_{\mathsf{STM}}^{\phi}(\mathcal{P}, m, k)$ interacting with the adversary $\mathcal{A}$.

large aggregate proof. On the other hand, we can use a circuit-based proof system such as Bulletproofs, which will produce much smaller proofs. However, this choice requires careful selection of the other primitives, as we need to e.g avoid evaluating random oracles in the circuit. We will further explore the instantiation options in section 4.3, and compare their efficiency in section 5.

We note that both of the hybrid functionalities we use are practical to realise in common applications. For $\mathcal{F}_{\mathsf{RS}}$, the group choice can be realistically hardcoded, leaving only the proof system reference string. In the options we explore in this section, the reference string is either empty or unstructured. For an unstructured reference string, we can use $H_{\mathbb{G}_1}$, and a random seed, as we only require random elements in $\mathbb{G}_1$. The key registration functionality, $\mathcal{F}_{\mathsf{RS}}$ can be realized by means of a broadcast channel which can be implemented via a blockchain.

4.1 Security

Theorem 1. *The protocol* $\Pi.\mathsf{STM}$ *of Sect. 4 realizes* $\mathcal{F}_{\mathsf{STM}}^{\phi}(\mathcal{P}, m, k)$ *in the* $\mathcal{F}_{\mathsf{RS}}(\mathcal{P})$, $\mathcal{F}_{\mathsf{Kr}}^{\psi_0}(\mathcal{P})$*-hybrid model, under the leveraged co-CDH assumption, if* H_p *is collision resistant and* $H_{\mathbb{G}_1\{0,1\}^* \to \mathbb{G}_1}$, $H_q : \{0,1\}^* \to \mathbb{Z}_q$ *are modeled as random oracles.*

Proof. We first describe the operation of the simulator:

The Key Registration functionality functionality $\mathcal{F}_{\mathsf{Kr}}^{\psi}(\mathcal{P})$.

$\mathcal{F}_{\mathsf{Kr}}^{\psi}(\mathcal{P})$ initializes the variable Allow to 1 and proceeds as follows:

- Upon receiving (Register, sid, vk) on behalf of party P_i:
  1. If Allow is 0, $P_i \notin \mathcal{P}$, $K(P_i)$ is already defined, or $vk \in K$, ignore the request.
  2. If $\psi(vk) = 1$, let $K(P_i) \leftarrow vk$, and output (RegKey, sid, 1) to P_i.
- Upon receiving (Retrieve, sid, P_i) on behalf of party P_j:
  1. $P_j \notin \mathcal{P}$, or $K(P_i)$ is not defined, output (Retrieve, $sid, P_i, \perp$) to P_j.
  2. Otherwise, output (Retrieve, $sid, P_1, K(P_i)$) to P_j
- Upon receiving (CloseRegistration, sid) on behalf of the adversary $\mathcal{A}$:
  1. Set Allow to 0.
  2. For each $P_i \in \mathcal{P}$, send (RetrieveAll, sid, K) to P_i.

Fig. 3. The Key Registration functionality $\mathcal{F}_{\mathsf{Kr}}^{\psi}(\mathcal{P})$ interacting with the adversary $\mathcal{A}$.

The Reference String functionality functionality $\mathcal{F}_{\mathsf{RS}}(\mathcal{P})$.

- Upon Initialization, let Param $\leftarrow$ Setup(1^{λ}); PS.RS $\leftarrow$ PS.Setup(Param) RS := (Param, PS.RS), and send (GetRS, sid, RS) to $\mathcal{A}$.
- Upon receiving (GetRS, sid) on behalf of party P_i:
  1. If $P_1 \in \mathcal{P}$ Output (GetRS, sid, RS) to P_i.

Fig. 4. The Reference String functionality $\mathcal{F}_{\mathsf{RS}}(\mathcal{P})$ interacting with the adversary $\mathcal{A}$.

Protocol Π.STM. *Initialisation phase*

- Setup: Users start in the initialisation phase. Each user locally sets $\mathsf{Reg} \leftarrow \emptyset$, and sends (GetRS, sid) to $\mathcal{F}_{\mathsf{RS}}(\mathcal{P})$. Upon receiving $(\mathsf{GetRS}, sid, \mathsf{RS})$, store RS.
- Register: Each user P_i gets their keys by running $(msk_i, mvk_i, \boldsymbol{\kappa}_i) \leftarrow \mathsf{MSP.Gen}(\mathsf{Param})$. They set $(vk_i, sk_i) := ((mvk_i, \boldsymbol{\kappa}_i), msk_i,)$. A user then sends $(\mathsf{Register}, sid, vk_i)$ to $\mathcal{F}_{\mathsf{Kr}}^{\psi_0}(\mathcal{P})$.
- Startup: When a user receives $(\mathsf{RetrieveAll}, sid, K)$, from $\mathcal{F}_{\mathsf{Kr}}^{\psi_0}(\mathcal{P})$ it sets $\mathsf{Reg} := (K(P_i), \mathsf{stake_i})$ for $P_i \in \mathcal{P}$, and Reg is padded to length N, using null entries of stake 0. Let $\mathsf{AVK} \leftarrow \mathsf{MT.Create}(\mathsf{Reg})$. The user moves to the operation phase.

Fig. 5. The Stake Based Threshold Multisignature Protocol Π.STM in the Initialisation Phase.

Protocol Π.STM. *Operation Phase*

- EligibilityCheck: On input (msg, index), user P_i runs: Let $\overline{msg} \leftarrow \mathsf{AVK}||msg$, $\sigma \leftarrow \mathsf{MSP.Sig}(msk, \overline{msg}); ev \leftarrow \mathsf{MSP.Eval}(\overline{msg}, \mathsf{index}, \sigma)$. Return 1 if $ev < \phi(\mathsf{stake})$, else return 0.
- CreateSig: On input (msg, index): If $\mathsf{EligibilityCheck}(msg, \mathsf{index})$ is 1, then let $\overline{msg} \leftarrow \mathsf{AVK}||msg; \sigma \leftarrow \mathsf{MSP.Sig}(msk, \overline{msg})$ and produce an individual signature $\pi = (\sigma, reg_i, i, \boldsymbol{p}_i)$, where $\boldsymbol{p}_i$ is the user's path inside the merkle tree AVK and reg_i is $(mvk_i, \mathsf{stake}_i)$.
- Verify: On input a party P_i, a signature π, index index, and message msg, parse $\pi = (\sigma, reg_i, i, \boldsymbol{p}_i)$. Parse reg_i as $(mvk_i, \mathsf{stake}_i)$. Check that reg_i corresponds to party P_i, let $\overline{msg} \leftarrow \mathsf{AVK}||msg; ev \leftarrow \mathsf{MSP.Eval}(\overline{msg}, \mathsf{index}, \sigma)$ check that $ev < \phi(\mathsf{stake}_i)$ and check $\mathsf{MT.Check}(\mathsf{AVK}, N, (vk_i, \mathsf{stake}_i), i, \boldsymbol{p}_i) = 1$. If parsing or checking fails, return 0. Otherwise, return $\mathsf{MSP.Ver}(\overline{msg}, mvk_i, \sigma)$.
- Aggregate: On input vectors $\boldsymbol{P}, \boldsymbol{\pi}, \mathbf{index}$ and message msg, parse $\boldsymbol{P}, \boldsymbol{\pi}$ and **index** as a vectors $P_j, \pi_j, \mathsf{index}_j$ of size k, let $\overline{msg} \leftarrow \mathsf{AVK}||msg$ and run $\mathsf{Verify}(P_j, \mathsf{index}_j, m, \pi_j)$.
  If parsing or checking fails, return $\perp$. If any $\mathsf{index}_j = \mathsf{index}_i$ for $j \neq i$ return 0. Otherwise, parse $\pi_j = (\sigma_j, reg_j, i_j, \boldsymbol{p}_j)$ and reg_j as $(mvk_j, \mathsf{stake}_j)$.
  Let $ivk \leftarrow \mathsf{MSP.AKey}(\mathbf{mvk})$, $\mu \leftarrow \mathsf{MSP.Aggr}(\overline{msg}, \boldsymbol{\sigma})$, set $x = (\mathsf{AVK}, ivk, \mu, msg)$ and $\boldsymbol{w} = (mvk_j, \mathsf{stake}_j, \boldsymbol{p}_j, ev_j, \sigma_j, \mathsf{index}_j)$ for $j = 1 \ldots k$. Then, $\pi_{avk} \leftarrow \mathsf{PS.P}(\mathsf{PS.RS}, x, \boldsymbol{w})$. Return $\tau = (ivk, \mu, \pi_{avk})$.
- VerifyAggregate: On input (τ, msg), parse $\tau = (ivk, \mu, \pi_{avk})$, check that $\mathsf{PS.V}(\mathsf{PS.RS}, (\mathsf{AVK}, ivk, \mu, msg), \pi_{avk})$ is true. If parsing and checking is successful, let $\overline{msg} \leftarrow \mathsf{AVK}||msg$ and return $\mathsf{MSP.AVer}(\overline{msg}, ivk, \mu)$.

Fig. 6. The Stake Based Threshold Multisignature Protocol Π.STM in the Operation Phase.

– Oracle Calls: The Simulator will always program the random oracle H_{G_1} with uniformly sampled group elements g_1^r with a known discrete logarithm $r \leftarrow \mathbb{Z}_q$ and stores their discrete log. This enables the simulator to produce a signature on behalf of any user-message pair by utilizing $\kappa_1 = g_1^{xr}$ for a known r from the proof of possession of the user and the log r' of the messages hash $h_{\mathbb{G}_1}(\text{“M”}\|\overline{msg}) = g^{r'}$, by setting $\sigma = k_1^{(1/r)r'}$.
– Register: The simulator runs the key generator MSP.Gen(Param) normally, returns the verification key vk_i and stores the private key sk_i.
– RegKey: The simulator runs the key verification algorithm MSP.Check and returns the output.
– EligibilityCheck: The simulator can evaluate eligibility for all participants, by signing on behalf of each user and then sets ideal functionality accordingly. This distribution is same as in real world, apart from potentially causing the functionality to abort, but that only occurs with only negligible probability.
– CreateSig: For honest users the simulator creates signatures normally. For malicious ones, it uses random oracle programability and the submitted proof of possession to create signatures that areindistinguishable from standard ones. In both cases, the simulator keeps an internal list $\mathcal{L}$ of produced signatures.
– Aggregate: Aggregation uses no private information, so the simulator can simply evaluate it using only public information. Any signatures produced this way are added to $\mathcal{L}$
– Verify: The simulator checks if the submitted signature exists in $\mathcal{L}$, and accepts if it is. Else, it verifies the signature and adds it to $\mathcal{L}$. If a signature belonging to an honest user is valid but was not in $\mathcal{L}$, the simulator aborts with output “forgery failure”. If a signature verifies but the corresponding user is not eligible, the simulator fails with output “individual failure” (this happens with negligible probability due to collision resistance).
– VerifyAggregate: On VerifyAggregate queries, the simulator checks if the submitted aggregate signature exists in $\mathcal{L}$, and accepts if it is. Else, it runs the verification algorithm on the aggregate signature. If verification succeeds, it counts the number of slots with either (1) previously produced single proofs for ($\overline{msg}$ in $\mathcal{L}$ or (2) a corrupted player eligible to sign. If the total is k or more, it accepts, otherwise it outputs “aggregate failure”.

Next, we will give a series of hybrid games between the interaction of the environment with the real protocol and between the environment and the simulator interacting with the ideal functionality.

The first game, H_0 represents the real protocol. We define H_1 to be identical to H_0, but with calls to the random oracle $H_{\mathbb{G}_1}$ being answered with elements with known discrete logs. I.e on query x, the simulator checks if there exists an entry (x, a, r) in table $\mathcal{R}$. If so, it returns a. If not, it sets $r \leftarrow \mathbb{Z}_q; a \leftarrow g_1^r$. It then stores (x, a, r) in table $\mathcal{R}$. Game H_1 is perfectly indistinguishable to H_0, as g_1 is a generator.

We define H_2 similar to H_1, but with Eligibility requests answered by the simulator. This is performed by the simulator evaluating the eligibility predicate across all users in $\mathcal{P}$ and indexes index. This is possible for all users, because the

simulator can derive signatures via the proofs of possession. It is clear that H_1 and H_2 are also perfectly indistinguishable.

In H_3, whenever Eligibility is queried for a message, the simulator calculates eligibility for each user and index to produce $\mathcal{B}$ with which it initializes the ideal functionality. If the Ideal Functionality aborts, the simulator also aborts. Clearly, H_3 only differs from H_2 if the ideal functionality aborts. However, that only happens with negligible probability (lemma 5). Thus, H_2 and H_3 are also statistically indistinguishable.

In H_4 the ideal functionality and simulator are used for $\mathsf{CreateSig}$ and Verify. The simulator is able to produce signatures for any user by programming the random oracle calls used for proofs of possession. Games H_3 and H_4 are indistinguishable unless the simulator outputs "forgery failure" or "individual failure". In lemma 7 we show that "forgery failure" reduces to the co-CDH problem and in lemma 6 we show that "individual failure" reduces to unique provability and collision resistance. Thus, either event only happens with negligible probability.

In H_5 the simulator now answers calls to both $\mathsf{Aggregate}$ and $\mathsf{VerifyAggregate}$. The simulation fails when the simulator outputs "aggregate failure" but is otherwise identical to the previous execution. The output "aggregate failure" happens with negligible probability due to lemma 8. At this point, it suffices to point out that H_5 is identical to the environment interacting with the simulator and the ideal functionality.

Lemma 5 (Sampling Property). *When $f \leq \frac{1}{4}$ and $a \leq \sqrt{1-f}$, the matrix sampled by the simulator causes the functionality to abort with only negligible probability.*

Proof. Let $\phi(\frac{1}{2}) = p$. Then $k = mp$

First, we point out that for $f \leq \frac{1}{4}$ and $a \leq \sqrt{1-f}$, it holds that for $p' = \phi(\frac{1}{2} - a)$ we have $\frac{p}{p'} = \frac{\phi(1/2)}{\phi(1/2-a)} \geq 1 + a$.

Each of the m columns of the matrix represents an independent trial in which with the adversary has a probability p' of being eligible via at least one corrupted user. Thus, the expected number of successes is the mean, i.e. $p'm = \frac{k}{1+a}$. The functionality will thus abort, if the actual number of successes, X is greater than $1 + a$ times the mean.

By chernoff bounds, the above probability is: $\Pr[X > k] = \Pr[X > p'm \cdot (1 + a)] \leq e^{\frac{-a^2 \cdot p'm}{2+a}} = e^{\frac{-a^2 pm}{(2+a)(1+a)}}$, which is negligible in m. For $m \geq \log^2 \lambda$ we obtain that the above is also negligible in λ.

As a corollary, by repeating the above analysis for $\phi(\frac{1}{2} + a)$, we can show that with overwhelming probability, the set of all honest parties will be able to produce a valid aggregate signature.

Lemma 6. *The simulator outputs "individual failure" with negligible probability.*

Proof. The simulator only outputs the above message if an adversarial signature $\pi = (\sigma^*, reg_i^*, i, \boldsymbol{p}_i)$ where reg_i^* as $(mvk_i^*, \mathsf{stake}_i^*)$ is valid but belongs to a user

who is not eligible. The user being non-eligible implies that an honest signature over the user's registered keyset $reg_i = (mvk_i, \mathsf{stake}_i)$ evaluates to a non-eligible value. As both signing and evaluating is deterministic, it must be that $reg_i^* \neq reg_i$ This directly produces a collision for $\mathsf{MT.Create}$ and thus for H_p.

Lemma 7. *The simulator outputs "forgery failure" with negligible probability.*

Proof. We will show that we can adapt the simulation so that if "forgery failure" occurs with non-negligible probability, the simulator is able to solve a co-CDH instance.

We carry out the reduction as follows. We assume the environment issues a maximum of q_{msg} non-PoP queries to the oracle $H_{\mathbb{G}_1}$. We select q^* randomly between 1 and q_{msg}. The simulator receives a co-CDH instance g_1^a, g_1^b, g_2^b. We select one honest user P^* to "trap" at random. We set the verification key of that user to $vk^* = (g_2^b, \pi^*)$, where $\pi^* = (g_1^b, g_1^{br})$, and program the random oracle so that $H_{\mathbb{G}_1}(\text{"PoP"} \| g_2^b) = g_1^r$. For all queries "PoP"$\|vk$ to the random oracle, we reply with g_1^{as} for $s \leftarrow \mathbb{Z}_q$ and save (vk, g_1^{as}, s) to a list $\mathcal{L}_{pop}$. For other queries "M"$\|\overline{msg}$ to $H_{\mathbb{G}_1}$, if this is not the $q*$-th query, we reply with g_1^t for $t \leftarrow \mathbb{Z}_q$ and save "M"$\|\overline{msg}, (g_1^t, t)$ to a list $\mathcal{L}_{msg}$. For the $q*$-th query, we reply with g_1^a, and store $(g_1^a, \perp)$ to $\mathcal{L}_{msg}$.

This configuration enables the simulator to sign most messages on behalf on any user, with the exception that P^* cannot sign the $q*$-th message querried. To produce a signature on $\overline{msg}$, under key $vk = g_2^x, (g_1^x, g_1^{sx})$ we lookup "M"$\|\overline{msg}, (g_1^t, t)$ on $\mathcal{L}_{msg}$. The signature is then $\sigma = {\pi_1}^t = g_1^{tx}$.

In the special case where t is $\perp$ we retrieve s from (vk, g_1^{as}, s) in $\mathcal{L}_{pop}$, and output $\sigma = {\pi_2}^{(1/s)} = g_1^{(asx)/s} = g_1^{ax}$. This is possible for all users apart from P^*.

If the simulator is about to output "forgery failure", then the signature σ^* must be such that $e(\sigma, g_2) = e(g_1^a, g_2^b)$ i.e. a solution to the coCDH problem.

Lemma 8. *The simulator outputs "aggregate failure" with only negligible probability.*

Proof. We distinguish between two cases:

- The statement $x = (\mathsf{AVK}, ivk, \mu, msg)$ is *contradictory* w.r.t the information the simulator holds. I.e ivk is not a product of eligible users' verification keys. This only happens with negligible probability due to lemma 3.
- The ivk contained in the statement is $ivk = \prod_{i=1}^{k} vk_i$ where each vk_i belongs to a user eligible for index index_i, and $\mathsf{index}_i \neq \mathsf{index}_j$ when $i \neq j$. In this case, the environment has produced a signature forgery, so we can reduce to co-CDH, similar to "forgery failure".

In the latter case, we carry out the reduction as follows.

First, the simulator determines the user keys used to construct ivk. This can be done by performing an exhaustive search on the set of eligible users at a cost of $\binom{m \cdot \phi(1)}{k} \approx \binom{m}{m/2} = O(2^m)$. For $m \approx \log^2 \lambda$, 2^m is $O(\lambda^{\log \lambda})$ which is super-polynomial, but not exponential in λ.

We assume the environment issues a maximum of q_{msg} non-PoP queries to the oracle $H_{\mathbb{G}_1}$. We select q^* randomly between 1 and q_{msg}. The simulator receives a co-CDH instance g_1^a, g_1^b, g_2^b. We select one honest user P^* to "trap" at random, in proportion to their stake. We set the verification key of that user to $vk^* = (g_2^b, \pi^*)$, where $\pi^* = (g_1^b, g_1^{br})$, and program the random oracle so that $H_{\mathbb{G}_1}(\text{"PoP"} \| g_2^b) = g_1^r$. For all queries "PoP"$\|vk$ to the random oracle, we reply with g_1^{as} for $s \leftarrow \mathbb{Z}_q$ and save (vk, g_1^{as}, s) to a list $\mathcal{L}_{pop}$. For other queries "M"$\|\overline{msg}$ to $H_{\mathbb{G}_1}$, if this is not the $q*$-th query, we reply with g_1^t for $t \leftarrow \mathbb{Z}_q$ and save "M"$\|\overline{msg}, (g_1^t, t)$ to a list $\mathcal{L}_{msg}$. For the $q*$-th query, we reply with g_1^a, and store $(g_1^a, \perp)$ to $\mathcal{L}_{msg}$.

This configuration enables the simulator to sign most messages on behalf on any user, with the exception that P^* cannot sign the $q*$-th message querried. To produce a signature on $\overline{msg}$, under key $vk = g_2^x, (g_1^x, g_1^s x)$ we lookup "M"$\|(\overline{msg}, (g_1^t, t)$ on $\mathcal{L}_{msg}$. The signature is then $\sigma = {\pi_1}^t = g_1^{tx}$.

In the special case where t is $\perp$ we retrieve s from (vk, g_1^{as}, s) in $\mathcal{L}_{pop}$, and output $\sigma = {\pi_2}^{1/s} = g_1^{(asx)/s} = g_1^{ax}$. This is possible for all users apart from P^*.

Before the simulator outputs "aggregate failure", on a correctly formed ivk, it checks to see if P^* is included in it. If it is, it is able to isolate σ^* from the aggregate signature by calculating the signature of every other user included in the key. The signature σ^* must be such that $e(\sigma, g_2) = e(g_1^a, g_2^b)$ i.e. a solution to the co-CDH problem.

This contradicts assumption 5 which states that there is no $O(\lambda^{\log \lambda})$ time solver for co-CDH.

Avoiding Complexity Leveraging. It is also possible to obtain the above result without using complexity leveraging. We can simply modify the proof system so that the user identities $\boldsymbol{i}$ are part of the statement instead of the witness. As such, they are immediately available to the simulator without an exhaustive search. This comes at a cost of $k \cdot \log N$ extra bits in τ.

4.2 Active Adversaries and Forward Security

We have modeled our functionality and scheme in a model with passive corruptions. In most proof of stake applications the possibility for active corruption greatly enhances the power of the adversary: the adversary waits to see which users are eligible to perform a particular action (e.g. the ability to produce the next block) and then selectively corrupts them. This allows the adversary to have a disproportionate amount of influence in comparison to the stake they hold. In our functionality, this is made weaker: eligibility is predicated on the message, and is independently distributed across different messages. That is, user P_1 being eligible for message msg_1 is independent of user P_1 being eligible for message msg_2. Nevertheless, in the ideal world, the adversary is able to set eligibility before performing corruptions, and would thus be able to assign eligibility to users before corrupting them.

In the real world, it is hard for the adversary to determine a user's ev values for any message the user has not signed due to the unforgeability of the signature

scheme and regularity of the mapping. If a user signs a particular message for a single index index_0, then the adversary *can* determine that user's evalution for every other index, but it is reasonable to assume that in most applications honest users will elect to either sign over as many indices they are able to, or not at all.

What a real-world adversary might do however is calculate the eligibility predicate over some indices without calculating ev (or equivalently,the CDH term σ). A line of research [9, 7, 20, 40] on the bit-security of CDH supports the assumption that guessing even partial information about the CDH term is hard. With this assumption in place, active corruptions only allow the adversary to take hold of a user who is known to be able to sign message msg, after she has already signed it.

A different issue, that exists beyond our modeling is that the stake distribution used by the functionality might lose relevance with time: that may be due to inflation or users selling their stake after the functionality has started. This implies that after a long period of time, the adversary might be able to acquire more than $\frac{1}{2} - a$ of the stake. This of course directly violates our model's assumptions, but it is an important real-world issue. As such, honest users should be assumed to delete their keys after a set of conditions has taken place (e.g an aggregate message has successfully been produced, containing an updated stake distribution or X amount of time has passed). Alternatively, multi signatures allowing for key evolution [19] can be used to avoid rekeying with every stake distribution update.

4.3 Instantiation Options

We can obtain two versions of the scheme by setting either $\mathsf{PS} = \mathsf{PS}^C$ and $M = M^R$, or $\mathsf{PS} = \mathsf{PS}^B$ and $M = M^E$.

For the first option, we are using a concatenation-based proof system, which in turn allows us to instantiate the dense mapping via a hash function modeled as a random oracle. At the same time, we only require that our group structure is pairing friendly, as that is required by the BLS based (multi-)signature scheme. Mutlisignature aggregation is somewhat underutilized as we require individual signatures to verify the mapping. However, we are able to batch verify faster than we would normally be able to: aggregating uses simple multiplication (compared to random exponents for batching)due to the security properties of the multisignature scheme.

The second option performs most of the checks in the circuit, leaving only the final pairing check, as well as random oracle calls to be performed in the open. This requires a "parent" group with order p so that we can design circuits performing modulo p arithmetic in order to efficiently perform group operations in $\mathbb{G}_1, \mathbb{G}_2$. At the same time, we need to use a mapping that only calls the random oracle on pre-determined points while achieving a near-uniform distribution. For this, we use $M = M^E$, described in section 6.

5 Efficiency

5.1 Quorum parameters

In the proof of lemma 5 we saw that the probability of an adversarial minority achieving a quorum is negligible. In this section, we investigate concrete values required for settings where the adversarial stake is 2/5 or 1/3, and the quorum percentage $\frac{k}{m}$ is set to approximately $\phi(\frac{11}{20}), \phi(\frac{12}{20}), \phi(\frac{13}{20})$. Increased values for the quorum percentage decrease the probability of an adversarial quorum, but also decrease the probability of an honest one. However, this is mitigated in that the probability of an honest quorum remains significant and can thus be boosted by allowing retries (e.g by attaching a short counter to the message). Furthermore, if an incentive structure is in place, rational adversaries who cannot directly subvert the protocol will choose to participate in signing honest messages. This could allow us to choose e.g. $\phi(\frac{13}{20})$ as the bound, while tolerating a 40% adversarial stake as such parameters require a rational adversary for liveness (since in the Byzantine setting only safety will follow but not liveness). For ease of presentation, we present our findings for $\phi(1) = \frac{1}{5}$. Decreasing the value slightly reduces k while increasing m.

	Adversarial Stake			
	40%		33%	
$\frac{k}{m}$	k	m	k	m
$\phi(11/20)$	1830	16385	654	5787
$\phi(12/20)$	1092	9010	467	3769
$\phi(13/20)$	743	5660	357	2642

Table 1. Required values of k, n so that an adversarial quorum is formed with $P \leq 2^{-100}$.

5.2 Proof Efficiency

Here, we investigate the costs of producing, transmitting and verifying individual as well as aggregate signatures.

Individual signatures For producing an individual signatures, a user needs to produce: $(\sigma, reg_i, i, \boldsymbol{p}_i)$. Producing $\boldsymbol{p}_i$, requires $\log N$ evaluations of H_p which can be amortised over multiple signatures on the same AVK. The signature itself, consists of one evaluation of $H_{\mathbb{G}_1}$ and one exponentiation. The cost of the mapping evaluation is the dominant factor, as a user needs to evaluate the representation function over all m possible indexes. The total cost is thus one exponentiation plus m representation evaluations. The length of individual signatures consists of is 2 group elements (one in $\mathbb{G}_2$), 3 bitstings for the stake, path, & index , and $\log N$ hashes, and is thus dominated by the hashes in $\boldsymbol{p}_i$. For simplicity, we

assume that the 3 bit strings can be packed in the equivalent of a single hash output: path needs $\log k$ bits, index needs $\log m$ and stake can be limited to 128 bit precision. When communicating between users who have the contents of AVK in memory, signatures can be reduced to 1 element for σ plus $\log N$ bits for i, as ev can be computed from $\sigma, \mathsf{index}, msg$.

The costs of the verifier are $\log N$ evaluations of H_p, a pairing and one verification of the mapping function. We note that a verifier who holds the (public) contents of AVK in memory can replace the hash evaluations with a lookup.

The final step is checking the mapping evaluation. In the vase of M^R this consists of a single hash evaluation. Verifying the elligator-based mapping M^E is more involved: the function selects one of many possible pre-images based on the index, which implies that the entire set $f^{-1}(Q)$ of pre-images needs to be made verified. Fortunately, in the analysis of Section 6, the pre-image set has a size[4] of either 4 or 2, depending on the quadratic character of an intermediate value. A square can be verified by providing its "root" as a witness, while a non-square can be verified by multiplying with a fixed, pre-determined non-square and providing a root for the product. This way, we can allow for exactly 4 pre-images r_1, r_2, r_3, r_4 where $r_2 < r_3$, with the additional condition that either $r_1 < r_2, r_3 < r_4$ or $r_1 = r_2, r_3 = r_4$ depending on the characteristic. The checking of characteristics, verification of roots and isogeny evaluation can be performed very efficient as verifying the value of a characteristic is much cheaper than calculating it: i.e for any y and a known non-square d, it is enough to produce a "root" r and a bit χ such that: $r \cdot r = y \cdot \chi + y \cdot d \cdot (1 - \chi)$. Enforcing uniqueness and correct ordering of the roots is the most expensive operation, requiring 3 range checks as we verify that $r_{i+1} - r_i$ is positive in the integers.

[4] The case of size 0 is also possible, but we will never be called to verify it.

Proof	Assymptotic	k=357	k=743	k=1092
Single PS^B	$G_1 + G_2 + (\log N + 1) \cdot H$	3.3KB	3.3KB	3.3KB
Single PS^B (FN)	$G_1 + G_2 + H$	224B	224B	224B
Single PS^C	$G_1 + G_2 + (\log N + 1) \cdot H$	1.1KB	1.1KB	1.1KB
Single PS^C (FN)	$G_1 + G_2 + H$	176B	176B	176B
Aggregate PS^B	$G_1 + G_2 + O(\log(k \log q)) \cdot G_p$	4KB	4.5 KB	4.5 KB
Aggregate PS^C	$G_1 + G_2 + (\log N - \log k + 3) \cdot H$	315KB	631 KB	900 KB

Table 2. Proof sizes for the PS^B and PS^C proof systems. G_i represent $\mathbb{G}_i$ elements, and H are hash outputs. Concrete values are based on the parameters on the text: $k = 170$, 446 bit base elements for the PS^B setting, and 384, 256 for elements and hashes in the PS^C setting. Single PS^B are over an arity-8 tree. The k values were derived from table 1. The indication (FN) is the setting where the verifier is a full node and hence certain metadata can be eliminated from the signature.

Proof	Operations
Single PS^B	$(\log N + 1)H_s + 1P$
Single PS^C	$\log_8 NH_p + 2H_s + 400F + 1P$
Aggregate PS^B	$O(k \log q)E + 1P$
Aggregate PS^C	$k \cdot (M_1 + M_2) + 1P$

Table 3. Verification complexity comparison for the dominant operations and terms. M_i represent $\mathbb{G}_i$ multiplications, F field operations, and E represent group multi exponentiations. H_s and H_p represent symmetric and Poseidon hashes respectively.

Given that, the cost of verifying a M^E evaluation is dominated by one evaluation of the isogeny map in the encoding function, and 4 evaluations of the SWU encoding.

PS^B aggregate signatures. For aggregate signatures in this settingthe dominating factor is the bulletproof. The circuit needs to verify the following operations:

- $k(\log N + 3)$ H_p evaluations for Merkle Tree lookups.
- $k - 1$ multiplications in $\mathbb{G}_2$ to produce ivk.
- $k - 1$ multiplications in $\mathbb{G}_1$ to produce μ.
- $2k$ range checks with bound m (for index bounds, and index uniqueness).
- k Comparisons between ev and $\phi(\mathsf{stake})$.
- k Mapping evaluations for ev.
- k ϕ evaluations.

We note that most of the above checks can be performed efficiently as they involve group operations in $\mathbb{G}_1, \mathbb{G}_2$ or field operations in $\mathbb{G}_p$ for which our proof systems are more efficient. The main outliers are the evaluation of ϕ. Fortunately, we don't actually need to evaluate ϕ in the proof: we can replace stake in the tree with $\phi(\mathsf{stake})$ and proceed with the comparison directly. This gives us a circuit size of $O(k \log q)$, and verifier complexity of $O(\frac{k \log^4 q}{\log (k \log q)})$ as verification is dominated by a multiexponentiation based on the circuit size.

Estimate for constraints We now give an estimation on the number constraints required for our scheme, with a k value of 170, $m = 2000$, and $\log p = \log q = 446$. We assume $\mathbb{G}_1$ operations to require 12 constraints, $\mathbb{G}_2$ operations 4 times as much, range checks from 0 to $2^b - 1$: b constraints, Merkle tree lookups approximately cost 7290 constraints, but can be brought down to 4050 by changing the arity of the tree to 8:1. This estimates the cost of performing the lookups individually. Given that we are doing multiple lookups, we can perform an additional optimization: the top layers of the tree are evaluated once for each user which is redundant: I.e the root hash is checked $k = 170$ times whereas it should be checked only once. The lower levels are more dense, but still provide benefits: the second level can be exhaustively checked with only 8 evaluations and the third with

64. This implies that the ammortized cost per lookup is ca 3009 constraints. H_p evaluations at $8:1$ compression cost 300 constraints and comparisons between values cost 2^b, 3b constraints, ammortized to 2b when values are used twice. Mapping representation operations involve 60 constraints plus 3 range checks.

In total we have:

- $k \cdot (2951 + 300)$ constraints for Merkle Tree lookups.
- $k \cdot 48$ constraints for multiplications in $\mathbb{G}_2$ for ivk.
- $k \cdot 12$ constraints for multiplications in $\mathbb{G}_1$ for ivk.
- $4k \cdot \log m$ for index range checks and comparisons with bound m.
- $3k \log q + 60k$ for representation function evaluations.

In total, we obtain $3k \log q + 4k \log m + 3429k \approx 2^{21}$ constraints. Extrapolating from [11, 26, 27], for k=357 this gives us a proof size of under 4KB with a batched verification time of ca. 50sec. Due to the incremental nature of signature and public key aggregation it is simple to split it into a constant number of steps and use a recursive proof system like Halo [10] to obtain a constant-time improvement in verification speed as well as a (small) improvement to proof size. As we only perform a constant number of recursion steps we are able to sidestep potential soundness issues with regard to extraction efficiency.

PS^C aggregate signatures. Concatenation based aggregate signatures are simpler to check: The verifier can simply check every index separately at a cost of k single verifications. This can be further optimized by checking the signatures themselves in aggregate. This replaces k pairing checks with $k-1$ multiplications in G_1 and G_2 and a single pairing check for the products. We point out that this is significantly faster than randomized checking with small exponents.

The size of the proofs is $2k$ group elements (k in $\mathbb{G}_2$), k hash output equivalents for the stake, path & index , and $k \log N$ hashes. For $k = 357, \log N = 30$, 446 bits per element and 256 bits per hash, this produces a proof size of ca. 400KB.

It is however possible to do better. We can reuse the previous observation about Merkle tree proofs over multiple leaves: for $k = 170$ leaves, revealing the entirety of the 8th level of the tree can be accomplished by publishing 128 hashes. In turn, this reduces the length for each individual inclusion proofs by 6 hashes: rather than giving a path to the root, inclusion proofs can terminate 6 levels early. This brings down the cost to the equivalent of $3k$ G_1 elements and $k(\log N - 6.25)$ hashes. As we don't need cycles, we can opt for a 384 bit curve following [2], and limit hashes to 256 bits. This produces a proof size of ca. 315KB.

5.3 Further PS Options

There exist a number of alternative circuit-based proof systems. SNARKs, such as Plonk [22] and Sonic [35] offer constant prover complexity with the main drawback of a trusted setup string. Our approximation of the circuit complexity should be representative of performance with such systems, though further optimizations may be possible, e.g with custom Plonk gates for Poseidon. STARKs, such as

Redshift [29] and Aurora [4] offer similar verifier performance at the cost of large proofs. In our application, zero knowledge is not a requirement, so the size required makes them less competitive compared to PS^C. Finally, recursive proof systems such as Halo [10] can also be explored: constant depth recursion can be used to reduce the verifier load by a constant, with little impact to soundness.Unbounded recursion may also be possible depending on the application, though technical complexities with oracle calls and extraction depth make such an adaption less than straightforward.

6 A Dense Mapping from Elligator Squared

In this section we propose a dense mappings based on Elligator Squared with a representation function compatible with the Pluto/Eris [27] BN curves. While constructions based on Elligator squared can be used with a very broad family of elliptic curves, efficiency can be lacking if the mapping used inside the representation function cannot be evaluated and inverted efficiently. Tailoring the representation function to a specific curve or curve family is thus necessary to arrive at meaningful efficiency estimates. The Praos MUPRF [16] uses a similar technique, but the additional requirements on group structure do not provide us with curves compatible with the original Elligator [6] construction. Elligator squared [42] uses a general technique that is compatible with a greater range of curves, but provides an efficient encoding function only for a subset of curves.

Boneh and Wahby [43] show how one can bridge this gap by using isogenies to tranfer points to a curve that is more efficient to represent. Their work focuses on the task of hashing into a curve as opposed to representing points as random-looking bitstrings, but the isogeny can be evaluated in reverse at a similar computational cost. A final obstacle is that Elligator squared uses randomness in the calculation of the representation which can be problematic to reason about inside a zero knowledge proof. We overcome this by pre-setting this randomness via a random oracle, and accepting a significant probability of evaluation failure. This is not a problem for our application, as we can account for the probability of failure by adjusting the weighting function.

The representation function $R : G_1 \times \{0,1\}^l \to \{0,1\}^l$ is specified below, adjusted from [42]. We modify it so that it always terminates after a single iterration with the caveat that it can fail (i.e produce $\perp$ as output) with significant probability. R is parametrised by the curve modulus p, and a a d-well bounded encoding f for $d = 4$.

The encoding f, is adapted from [43]. It is parametrized by a curve E_I, isogenous to E, where $G_1 \in E$, with an isogeny $\mu : E \to E_I$ of degree 3 [27].To evaluate $f(Q)$, we let $Q_2 \leftarrow \mu(Q)$, and then evaluate the simplified SWU encoding on $Q_2 \in E_I$. To calculate the inverse, we raise to the inverse of 3 $\mod q$, apply the dual of μ, and calculate the inverse encoding in E_I as in [42]. A key observation from the investigation of [42, 43] into this calculation is that $f^{-1}(Q)$ consists of the roots of a bicubic equation and is thus efficient to both calculate as well as prove.

Algorithm 1 Elligator Squared Representation

procedure FUNCTION R(y,x,t)
  $Q \leftarrow y - h_{\mathbb{G}_1}(x||t)$
  $n \leftarrow \#f^{-1}(Q)$
  $j \leftarrow H_q(x||t) \mod 4$
  if $n < j$ **then return** $\perp$
  end if
  $\{z_0, \ldots, z_n\} \leftarrow f^{-1}(Q)$
  return Return z_x, where $z_j = (z_x, z_y)$
end procedure

To calculate the success probability of Algorithm 1 we invoke Lemma 5 of [42], which we restate for the reader's convenience. Let $P(y) = \Pr[R(y, x, t) \neq \perp]$ and $N(y) = \frac{1}{P(y)}$.

Lemma 9 (Lemma 5,[42]). *For all y, let $\epsilon_T(y) = N(y)/d - 1$, where d is the bound of the encoding function f. Then, for all points y except possibly a fraction of $\leq p^{-1/2}$ of them, we have:*

$$\epsilon_T(y) \leq O(p^{-1/4})$$

Corollary 1. *Algorithm 1 terminates with an output other than $\perp$ with probability at least $\frac{1}{5}$.*

Proof. From lemma 9, and for $d = 4$ we know that for all but a fraction of $\leq p^{-1/2}$ y, $N(y) \leq 4 + O(p^{-1/4})$, thus $P(y) \geq \frac{1}{4+O(p^{-1/4})}$. Thus, for all y, we have $P(y) \geq \frac{1}{5}$.

The regularity of the output is a direct consequence of applying Elligator Squared to a uniformly random point Q. The only difference is that we choose to abort early, and allow for a significant probability of returning $\perp$.

Theorem 2 ([42]). *The non-$\perp$ outputs of Algorithm 1 are ϵ-close to uniform for $\epsilon = O(p^{-1/2})$.*

We are now ready to show the main result of this section. Let $R(\cdot)$ be the representation function described in Algorithm 1.We can prove the following lemma as an immediate outcome of Corollary 1 and Theorem 2.

Lemma 10. *For all $msg \in \{0,1\}^*$, and all $\mathsf{index} \in \mathbb{Z}$, the function $M^E_{msg,\mathsf{index}}(y) = R(msg, y^{H_q(msg,\mathsf{index})}, \mathsf{index})$ is a dense mapping with $Pr[M(y) \neq \perp] < \frac{1}{5}$.*

7 Applications

In this section we delve with some more detail to some applications of mithril (STM) signatures in the blockchain setting. In general, STMs could be applied in any setting where we can associate an amount of stake to a set of public-keys. Given such arrangement, stakeholders can produce certificates for any given message msg of interest. Even though grinding attacks have a negligible probability to produce a forgery, cf. Lemma 5, an attacker who knows msg prior to the keys being finalized, can attempt to grind the probability of signing msg by trying multiple keys. In this way the attacker will boost somewhat the number of lottery tickets it wins, something undesirable in practice (since e.g., we would need to take this opportunity into account when selecting the number of lotteries m). In practice, it will be sufficient to verify that any msg considered for certification is unpredictable during the pubic-key generation stage (in the blockchain setting, this can be done by e.g., including an unpredictable fresh nonce drawn from the blockchain itself as part of the message).For simplicity of exposition, we take this as a given in the examples below.

Bitcoin Referendums. We first consider using mithril in the context of a proof-of-work cryptocurrency such as Bitcoin as a decision-making tool. Using STM it is possible to probe the population of Bitcoin holders (as opposed to, say, the miners) regarding a particular topic or action. The idea is to express the action in a message msg, agree on a stake threshold, (e.g., over 1/2 of all Bitcoin supply) and then have them use STM to sign msg. If the threshold is exceeded then it is possible to aggregate all individual signatures into a final signature certification that assures the topic has been accepted by over 1/2 of the Bitcoin supply. Below we provide an overview of how STM can be incorporated without requiring any hard or soft fork of the Bitcoin codebase.

In Bitcoin, balances are sent to a ScriptPubKey and are spentable by revealing a corresponding ScriptSig. The ScriptPubKey value can be either of the form pay to public-key (p2pk) or pay-to-script-hash (p2sh). Payments of the latter form are made to ScriptPubKey = `OP_HASH160 <scripthash> OP_EQUAL` where `<scripthash>` is the hash of a "redeem script" that needs to be provided when the UTXO is spent. Using p2sh it is possible to receive payments and associate the resulting UTXO with an STM public-key. Specifically we can use the following redeem script: `OP_HASH160 <STMpkhash> OP_EQUALVERIFY OP_HASH160 <pkhash> OP_EQUALVERIFY OP_CHECKSIG` which contains the hashes of the STM public-key and of an additional ECDSA key that controls the script balance; spending it requires opening both keys and the signature for the ECDSA key.

Such a p2sh can be spent with the following ScriptSig `<Sig> <pk> <STMpk> <RedeemScript>`. Evaluating this script by itself, will verify `<STMpkhash>`, `<pk>` and the ECDSA signature. Subsequently it is also verified that the `<RedeemScript>` verifies correctly to `<scripthash>`.

We observe that the above mechanism achieves the following objectives: the `STMpk` value is hashed into ScriptPubKey as well as `<RedeemScript>` . Revealing the latter, enables anyone offchain to verify, *but not spend*, the stake of

`STMpk` –spending would also require the ECDSA signature `<Sig>`. Thus, individual STM signatures can be verified and matched to the stake they correspond to.

Based on the above it is straightforward to use our STM construction as a decision-making tool for Bitcoin holders. A proposal msg will be announced together with a threshold. Interested bitcoin owners reveal their `<RedeemScript>` values and can issue an individual signature on msg. Note that all that is happening off-chain as a layer 2 type of coordination. When a sufficient number of those individual signatures are collected on msg, they can be aggregated to issue the aggregate signature on behalf of the Bitcoin holders collectively.

Fast bootstrapping in PoS Blockchains. In this scenario we want to facilitate the expedient synchronization of a client for a proof of stake blockchain. The problem is similar to the problem of simplified payment verification (SPV) as in [38], with the challenge that in a PoS blockchain, e.g., [32], there is no way to verify blocks just by looking at the headers (as in the case of a PoW-based blockchain); some transactional information is essential to establish the stakeholder distribution that is eligible to issue blocks.

In order to facilitate the use of mithril in this setting first we have to expand the blockchain accounting model so that each account is also associated with an STM key — in addition to any other cryptographic keys necessary for spending the balance or other operations such as delegating stake to other accounts. We assume a synchronous system operation and divide time in periods; the length of each period is sufficient to allow ledger settlement. Let SD_i be a stakeholder distribution that has become settled in the ledger (and hence all honest parties are in agreement of) during period i. Note that SD_0 is the stakeholder distribution embedded in the genesis block; we assume that all parties are in agreement regarding SD_0.

When the distribution SD_i is derived from the blockchain, the message $msg_i = (i, C_i)$ is formed where C_i is a Merkle tree commitment to SD_i. Subsequently the stakeholders in SD_{i-1} attempt to issue an STM on msg_i. Whenever a stakeholder is eligible, they release the individual signature over the peer-2-peer network. If sufficient individual signatures are collected with respect to the given stake threshold (e.g., 1/2 or 2/3 as desired), the resulting signature, denoted by chp_i can be computed and disseminated. The triple (i, C_i, chp_i) is considered the i-th checkpoint of the blockchain.

In this way, the system continuously issues checkpoints. When a new client joins for the first time with only knowledge of the genesis block, it queries and verifies the sequence of checkpoints starting from the genesis block and arriving up to the most recent one SD_n. Subsequently individual blocks can be verified with respect to SD_n.

We observe that the above mechanism can be made to be, asymptotically, of the same complexity as the SPV verification mechanism in PoW blockchains. In particular, for a blockchain of length N, SPV requires clients to perform work $O(N \log q)$ work (this is because of the linear in $\log q$ cryptographic operations that need to be performed per block to verify the headers). To match this, in our application of STM, we can set the period frequency to be every $\delta =$

$k \log^3 q / \log(k \log q)$ blocks, so that the verifier complexity will be proportional to $N/\delta \cdot O(k \log^4 q / \log(k \log q)) = O(N \log q)$.

8 Conclusion

We introduced a new cryptographic primitive, stake-based threshold multisignatures that is well suited for applications in the cryptocurrency setting. In particular, it allows a population of stakeholders to issue a certificate for a given message that can be coordinated safely and efficiently in a peer-to-peer manner. Contrary to previously known methods that achieve similar objectives, our primitive does not require a committee to be predetermined and agreed between the stakeholders; instead, every message is implicitly associated with a pseudorandomly sampled committee and committee membership is reliably and importantly *privately* determined by each stakeholder. To others, committee membership is disclosed only after being exercised.

Our construction, based on bulletproofs and multisignatures, is efficient and relies on an unstructured reference string — something that makes it immediately relevant to cryptocurrencies without a structured reference string, as well as a complexity leveraging security argument. We do expect that our work will motivate further research on the primitive and improve both its performance, as well as its security characteristics (e.g., relax further the cryptographic assumptions).

References

1. Baldimtsi, F., Madathil, V., Scafuro, A., Zhou, L.: Anonymous lottery in the proof-of-stake setting. In: Computer Security Foundations Symposium - CSF (2020)
2. Barbulescu, R., Duquesne, S.: Updating key size estimations for pairings. Journal of Cryptology **32**(4), 1298–1336 (2019)
3. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Conference on Computer and Communications Security - CCS. pp. 62–73 (1993)
4. Ben-Sasson, E., Chiesa, A., Riabzev, M., Spooner, N., Virza, M., Ward, N.P.: Aurora: Transparent succinct arguments for r1cs. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 103–128. Springer (2019)
5. Benhamouda, F., Gentry, C., Gorbunov, S., Halevi, S., Krawczyk, H., Lin, C., Rabin, T., Reyzin, L.: Can a public blockchain keep a secret? In: Theory of Cryptography, TCC. pp. 260–290 (2020)
6. Bernstein, D.J., Hamburg, M., Krasnova, A., Lange, T.: Elligator: elliptic-curve points indistinguishable from uniform random strings. In: Conference on Computer & communications security - CCS. pp. 967–980 (2013)
7. Blake, I.F., Garefalakis, T., Shparlinski, I.E.: On the bit security of the diffie-hellman key. Applicable Algebra in Engineering, Communication and Computing **16**(6), 397–404 (2006)
8. Boneh, D., Drijvers, M., Neven, G.: Compact multi-signatures for smaller blockchains. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 435–464. Springer (2018)

9. Boneh, D., Shparlinski, I.E.: On the unpredictability of bits of the elliptic curve diffie-hellman scheme. In: Annual International Cryptology Conference. pp. 201–212. Springer (2001)
10. Bowe, S., Grigg, J., Hopwood, D.: Recursive proof composition without a trusted setup. Tech. rep., Cryptology ePrint Archive, Report 2019/1021, 2019. https://eprint.iacr.org/2019/1021 (2019)
11. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 315–334. IEEE (2018)
12. Bünz, B., Kiffer, L., Luu, L., Zamani, M.: Flyclient: Super-light clients for cryptocurrencies. In: IEEE Symposium on Security and Privacy (SP). pp. 928–946. IEEE (2020)
13. Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N., Peled, U.: UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In: Conference on Computer and Communications Security - CCS '20. pp. 1769–1787 (2020)
14. Chen, J., Micali, S.: Algorand: A secure and efficient distributed ledger. Theor. Comput. Sci. **777**, 155–183 (2019)
15. Choudhuri, A.R., Goel, A., Green, M., Jain, A., Kaptchuk, G.: Fluid MPC: secure multiparty computation with dynamic participants. IACR Cryptol. ePrint Arch. **2020**, 754 (2020), https://eprint.iacr.org/2020/754
16. David, B., Gazi, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Advances in Cryptology - EUROCRYPT 2018. pp. 66–98 (2018)
17. Desmedt, Y., Frankel, Y.: Shared generation of authenticators and signatures (extended abstract). In: Advances in Cryptology - CRYPTO '91. pp. 457–469 (1991)
18. Dodis, Y.: Efficient construction of (distributed) verifiable random functions. In: International Workshop on Public Key Cryptography. pp. 1–17. Springer (2003)
19. Drijvers, M., Gorbunov, S., Neven, G., Wee, H.: Pixel: Multi-signatures for consensus. In: USENIX Security Symposium - USENIX Security 20. pp. 2093–2110 (2020)
20. Fazio, N., Gennaro, R., Perera, I.M., Skeith, W.E.: Hard-core predicates for a diffie-hellman problem over finite fields. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology – CRYPTO 2013. pp. 148–165. Springer (2013)
21. Gabizon, A., Gurkan, K., Jovanovic, P., Konstantopoulos, G., Oines, A., Olszewski, M., Straka, M., Tromer, E.: Plumo: Towards scalable interoperable blockchains using ultra light validation systems (2020)
22. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953 (2020), https://ia.cr/2019/953
23. Ganesh, C., Orlandi, C., Tschudi, D.: Proof-of-stake protocols for privacy-aware blockchains. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 690–719. Springer (2019)
24. Gaži, P., Kiayias, A., Zindros, D.: Proof-of-stake sidechains. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 139–156. IEEE (2019)
25. Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Secure distributed key generation for discrete-log based cryptosystems. J. Cryptol. **20**(1), 51–83 (2007)
26. Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: A new hash function for zero-knowledge proof systems. In: USENIX Security Symposium - USENIX Security 21. USENIX Association (2021)
27. Hopwood, D.: Pluto/eris half-pairing cycle of elliptic curves, https://github.com/daira/pluto-eris

28. Itakura, K., Nakamura, N.: A public-key cryptosystem suitable for digital multisignatures. NEC Research and Development **71**, 1–8 (October 1983)
29. Kattis, A., Panarin, K., Vlasov, A.: Redshift: Transparent snarks from list polynomial commitment iops. IACR Cryptol. ePrint Arch. **2019**, 1400 (2019)
30. Kerber, T., Kiayias, A., Kohlweiss, M., Zikas, V.: Ouroboros crypsinous: Privacy-preserving proof-of-stake. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 157–174. IEEE (2019)
31. Kiayias, A., Miller, A., Zindros, D.: Non-interactive proofs of proof-of-work. In: International Conference on Financial Cryptography and Data Security. pp. 505–522. Springer (2020)
32. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Annual International Cryptology Conference. pp. 357–388. Springer (2017)
33. Leung, D., Suhl, A., Gilad, Y., Zeldovich, N.: Vault: Fast bootstrapping for the algorand cryptocurrency. In: NDSS (2019)
34. Li, C., Hwang, T., Lee, N.: Threshold-multisignature schemes where suspected forgery implies traceability of adversarial shareholders. In: Advances in Cryptology - EUROCRYPT '94. pp. 194–204 (1994)
35. Maller, M., Bowe, S., Kohlweiss, M., Meiklejohn, S.: Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In: Conference on Computer and Communications Security - CCS. pp. 2111–2128 (2019)
36. Micali, S., Ohta, K., Reyzin, L.: Accountable-subgroup multisignatures: extended abstract. In: Conference on Computer and Communications Security - CCS. pp. 245–254 (2001)
37. Micali, S., Rabin, M., Vadhan, S.: Verifiable random functions. In: 40th annual symposium on foundations of computer science (cat. No. 99CB37039). pp. 120–130. IEEE (1999)
38. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2008)
39. Ristenpart, T., Yilek, S.: The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In: Naor, M. (ed.) Advances in Cryptology - EUROCRYPT 2007. pp. 228–245. Springer (2007)
40. Shani, B.: On the bit security of elliptic curve diffie–hellman. In: IACR International Workshop on Public Key Cryptography. pp. 361–387. Springer (2017)
41. Shoup, V.: Practical threshold signatures. In: Advances in Cryptology - EUROCRYPT. pp. 207–220 (2000)
42. Tibouchi, M.: Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In: International Conference on Financial Cryptography and Data Security. pp. 139–156. Springer (2014)
43. Wahby, R.S., Boneh, D.: Fast and simple constant-time hashing to the bls12-381 elliptic curve. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 154–179 (2019)