Mithril: Stake-based Threshold Multisignatures

Pyrros Chaidos[1] and Aggelos Kiayias[2]

[1] National & Kapodistrian University of Athens & IOG
`pchaidos@di.uoa.gr`
[2] University of Edinburgh & IOG
`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 certificate 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, communicating the signatures, aggregating them to a certificate and verifying it.
We formalize the primitive in the universal composition setting and propose efficient constructions for STMs in the unstructured reference string model. We also showcase that STMs are eminently useful in the blockchain setting by providing three applications: (i) stakeholder decision-making for Proof of Work (PoW) blockchains, specifically, Bitcoin, (ii) fast bootstrapping for Proof of Stake (PoS) blockchains, and (iii) proofs of data availability for consensus scaling.

1 Introduction

A wide class of multiparty cryptographic protocols is 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 which is 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 [20] and the Ouroboros protocols [43, 41, 24], and more recently secure multiparty computation [8, 21].

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

– 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, the threshold is expressed in terms of stake rather than a number of keys or identities.
– Second, in an STM, contrary to a multisignature, not all signers are eligible to sign all messages — this is necessary in order to match the communication scalability requirement. On the other hand, when they are eligible, similar to a multisignature, they can act independently producing (pre-)signatures that can be individually verified.
– 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, verification *as well as total communication* to depend logarithmically in N. Furthermore, we allow for the verifier to operate using a concise verification key.

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, Algorand or Tezos, using an STM, it is possible to certify the state of the ledger efficiently at regular intervals by 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" across checkpoints 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 detail, 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. For this reason, we introduce the notion of an eligibility check before signing. At the same time, we also use the notion of an index, iterating over the available seats in the committee.
  Thus, for any message msg, the STM functionality can be thought of as initiating a lottery for each of the m available committee seats, and each prospective signer can check to see if they win it (it is feasible for somebody to win multiple seats). 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 a ticket winning or not is a function of the party's stake, and it is calculated so that the party has 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 signatures from k different "seats" 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.* We describe two instantiations of our primitive: one optimized for speed and simplicity of implementation, and one that is optimized for space. We do so in a modular way, by building two proof systems around the same relation. Our relation directly uses batch verification for efficiency and to also enable random oracle calls to be outsourced to the verifier. In this way, it is simple to extend our current design in view of different requirements or assumptions.
- *Efficiency Considerations and Applications.* We compare the space efficiency of our construction with that of similar primitives and describe three potential applications in which our design is readily applicable. First, we describe how STM functionality can be integrated into bitcoin by using pay-to-script-hash P2SH to facilitate registration. Second, we describe how STMs can facilitate bootstrapping in Proof of Stake (PoS) blockchains. Finally, we comment on how STMs can play a role in the design of high performance permissionless distributed ledger protocols.

1.1 System Overview and Design Challenges

The operation of our primitive, Stake-based Threshold Multisignatures (detailed in Section 4) is as follows: the semantics are similar to those of a standard threshold signature scheme, with the addition of an eligibility predicate based on user stake. The purpose of the predicate is to pre-emptively filter the number of users signing each message to a quantity independent of the number of total users, and independent of the particulars of the stake distribution.

In typical stake-based blockchain constructions, blocks are produced by turning to a verifiable or distributed randomness generation to select the users responsible for block production, and then by having the selected users sign the blocks. Our construction (Section 5) aims to instantiate our primitive by combining this random selection with the signature. To extend the lottery analogy, in our construction the individual signatures will also serve as eligibility tickets. The odds of a ticket winning are proportional to the signer's stake.

From stake to tickets. Potential signers can check eligibility locally leading to clear efficiency gains: non-winning tickets incur no communication, storage or aggregation costs. On top of this, we will also need a mechanism that checks that a particular message is in fact supported by stakeholders of a sufficient amount of stake — a form of signature aggregation. To accomplish this, we run m lotteries (signing sessions) in parallel and require that at least k of them are won (produce a successful signature), for suitable choices of the parameters k, m. Subsequently we facilitate signature aggregation by using them as witnesses in a properly crafted aggregation relation. A particular challenge in our setting is to ensure that the adversary cannot bias the lotteries to its advantage, especially given the fact that our constructions cannot always rely fully on idealized abstractions such as the random oracle model.

Verifying signatures in this system would require verifiers to know the public keys and stake held by each user, which can often be cost-prohibitive in large user sets. We formalize this requirement by requiring a key registration functionality that organizes the participants' stake; to minimize the assumptions placed on the setup of the primitive, we assume the 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 setup and hence need not encumber the parties computationally). This way, verifiers only need to be aware of the tree root rather than the contents.

In this way, verifiers only need to be made aware of the tree root rather than the entirety of the contents. In turn, this implies that signatures need to contain the path to their key and stake alongside their signature and session index(es) for which they claim they are eligible. This is still a net gain, as the length of the Merkle tree path is only logarithmic with regard to to the number of users.

A natural tool for concisely aggregating Merkle proofs as well as signatures, are zero knowledge proofs. Interestingly, we only require compactness and not secrecy. Even so, we face a number of design challenges: the hash function in the Merkle tree needs to be optimized for use inside a proof system, as well as the signature and mapping verification. Furthermore, we cannot directly encapsulate a random oracle inside the proof system as that would swap the oracle for a concrete function [3]; calls must be avoided or externalized. We lay out the groundwork for a circuit-based approach by utilizing a modular approach for our design, and elaborate on a Bulletproof based construction describing the necessary components and an Elligator-based mapping function.

A particular challenge in this setting, is that independence of eligibility for different keys does not hold (as, without random oracles, the adversary can potentially craft adversarial keys that enjoy increased probability of being selected) and hence we have to be able to ensure that this attack vector does not invalidate the security of the construction.

Armed with the above design approach, we utilize bulletproofs [15] and an efficient arithmetic hash such as Poseidon [37] to implement the Merkle tree resulting in a space efficient STM with length independent of k, (Sect. 5.3). For completeness we also present a simpler instantiation (Sect 5.2) where we just use hashing, in the random oracle model, and as a proof system, we simply reveal the witness. Note that in both cases, we use proofs of possession to ensure resistance to rogue key attacks.

In Section 6.1 we evaluate the efficiency of our construction in terms of committee size, proof sizes and an estimate for constraints on the bulletproof-based instantiation. The number of constraints that are needed for the circuit is approximately 2^{22}, and aggregate proof sizes can be as small as 4KB using Bulleproofs. Concatenation based proofs are ca. 100-350KB in size, but are faster to verify.

In terms of applications, in Section 8 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 for our construction as described 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 (unlike an SPV client in the bitcoin setting, that can simply 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.

1.2 Comparisons to Related Work.

Multisignatures, introduced in [39] enable 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 multisignature. In [53] 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 [25]. In a threshold signature, there is a threshold t so that a signature only 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.,[34], [55], [18].Nevertheless it is desirable, especially in the blockchain setting, to have an *ad-hoc* key generation where signers can post their keys in an asynchronous fashion and that the subgroup which acts for a particular message is determined dynamically.

Threshold signatures and multisignatures were combined in [45] 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 was formalized in this context in the form of accountable subgroup multisignatures (AMS) [49]. Ad-hoc threshold multisignatures (ATMS) were put forth in [33]. 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. Key generation is ad-hoc without participant coordination. This allows a maximal committee to be fixed ahead of time whilst allowing for individual members to abstain or be unavailable. The multisignature-based construction in [33] operates by first committing the verification keys of all users to a Merkle tree and also producing an aggregate verification key for the entire user set. Signing operates by producing a multisignature representing all the users who *did* participate, as well as a list of all the verification keys of users who did not. The list is supported by Merkle tree proofs verifying their membership in the set. This results in a size linear to the number of abstaining users (regardless of their amount of stake).

In contrast, our notion of a "threshold" is predicated by the stake held by each user and additionally involves random eligibility 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).

More recently, Micali et al. [50] introduced compact certificate schemes (CCCK) which can be seen as the stake-based version of ATMS. Compared to our primitive, they lack the concept of eligibility. As a result, depending on the stakeholder distribution, a significant percentage of the user base needs to produce and transmit their individual signatures in order for the protocol to succeed. They do utilize sampling during aggregation however, something that enables them to only reveal a small number of signatures as proof of a certificate's validity. The construction of [50], uses a Merkle tree for registration, similar to ATMS and Mithril. For signing, it first commits to the set of collected signatures, and then uses random sampling to determine which of the committed signatures will be revealed to the verifier.

Interestingly, in terms of efficiency, this adaptive sampling enables the use of a more aggressive quorum parameter, producing certificates that are 2-3 times smaller than our concatenation-based instantiation, with similar asymptotics. However, this comes at the expense of a centralized aggregator that needs to collect all signatures making the communication of the approach unattractive.

The contemporary work on the Telescope [19] family of protocols is also applicable to our setting. Though the authors frame their design as a framework to prove knowledge of elements satisfying a predicate, it is straightforward to adapt it in a signature setting by means of a unique signature scheme. For a certain message m, an element satisfies the predicate iff it correctly verifies against on of the verification keys in a fixed set. The authors offer a centralized version where every signer needs to contact the aggregator, and a decentralized version where eligibility sampling is performed beforehand, though the protocol exhibits a very high lower bound in the number of transmitted signatures. This is visible in Table 1 as the decentralized version tracks the centralized one even for 2^{20} users. To adapt their scheme to our setting, we need to enable certificate recipients to verify signatures: in the Telescope setting this functionality is a given. Towards this, we assume the set of verifier keys is stored in a Merkle Tree, and include costs for the Merkle paths in the certificate size only. We also provide costs were the signed message is split in two parts, $(\mathsf{topic}, \mathsf{body})$ which are signed separately with only topic determining eligibility as in Sect 4.

Our construction instead is scalable in terms of communication costs and aggregation effort as only a small subset of users is involved in signature production. We can also implement STMs using bulletproofs for the proof system, something that squashes the proof length (at the cost of higher computation). We note that

System	$\log N = 10$		$\log N = 13$		$\log N = 20$		$\log N = 30$	
	comms	size	comms	size	comms	size	comms	size
Baseline - Participation	64	42	512	335	64 MB	42 MB	64 GB	42 GB
ATMS [33]	48	.05	384	.05	48 MB	.05	48 GB	.05
CCCK [50]	64	34	512	49	64 MB	84	64 GB	134
Telescope, Cent. [19]	102 (54)	17	816 (432)	25	102 (54) MB	44.5	102 (54) GB	72
Telescope, Dec. [19]	102 (54)	18	816 (432)	26.5	102 (54) MB	47	179 (97) MB	76.5
PS^C [Sec 5.2]	61 (32)	102	61 (32)	141	61 (32)	234	61 (32)	367
PS^C CH [Sec 5.2, 6.1]	132 (70)	69	132 (70)	96	132 (70)	158	132 (70)	248
PS^B [Sec 5.3]	70 (37)	4.5	70 (37)	4.7	70 (37)	5.1	70 (37)	5.6
PS^B CH [Sec 5.3, 6.1]	153 (80)	4.3	153 (80)	4.4	153 (80)	4.6	153 (80)	4.9
Baseline - Abstention	43	42	341	335	43 MB	42 MB	43 GB	42 GB
ATMS [33]	32	64	256	512	32 MB	64 MB	32 GB	64 GB
CCCK [50]	43	46	341	70	43 MB	126	43 GB	206
Telescope, Cent. [19]	68 (36)	24	544 (288)	37.5	68 (36) MB	68	68 (36) GB	112
Telescope, Dec.. [19]	68 (36)	25.5	544 (288)	39.5	68 (36) MB	72.5	311 (165) MB	120
PS^C [Sec 5.2]	88 (47)	102	88 (47)	141	88 (47)	234	88 (47)	367
PS^C CH [Sec 5.2, 6.1]	92 (49)	161	92 (49)	232	92 (49)	401	92 (49)	641
PS^B [Sec 5.3]	102 (54)	4.5	102 (54)	4.7	102 (54)	5.1	102 (54)	5.6
PS^B CH [Sec 5.3, 6.1]	106 (56)	5.5	106 (56)	5.9	106 (56)	6.6	106 (56)	7.6

Table 1. Comparison to previous work for N users with sizes in kilobytes (KB) unless noted. Communication costs are the sum of all individual signatures produced by signers. We assume a flat (uniform) stake distribution, $\frac{1}{3}$ adversarial stake and full adversarial abstention (bottom subtable) or participation (top). This leads to $\texttt{numreveals} = 128/80$ for CCCK when the adversary is abstaining/participating. We use $k = 424$ for $\mathsf{PS}^B, \mathsf{PS}^C$. Signature and hash bit lengths are 512/256 for the baseline and CCCK systems, 384/256 for ATMS, Telescope and PS^C and 446/446 for PS^B respectively. In all cases aggregation must be performed by a full node, see Table 4. CH indicates a concurrent hybrid of $k = (286, 769), m = (1747, 6654)$, see Section 6.1. For PS^B we have included the cost to avoid complexity leveraging Sect. 5.5. For an abstaining adversary, we calculate the expected communication cost including retries. We optimize all Merkle tree proofs as in Section 6.2. The parenthesised values correspond to the "empty body" variant in Section 6.5.

the constructions of [50, 19] could possibly similarly be augmented with a more compact proof system but has not been explored in the corresponding works.

We provide a comparison with concrete numbers between the schemes in Table 1, showcasing the scalability of Mithril against a naive base scheme, the ATMS construction of [33] the compact certificates of [50] and Telescope [19]. The naive baseline system polls all users and produces a certificate by fully revealing enough signatures to overtake the presumed adversarial stake, as described in [50]. The PS^B instantiations of Mithril make use of bulletproofs to reduce the proof size. The security proof for the PS^B instantiation requires an optional leveraging argument. If we eschew leveraging, we will need to also add the leaf index corresponding to each of the k contained signatures. This adds $k \cdot \log N$ bits to the proof size, and has no effect to the other metrics. In the most extreme case, this will add 2.7KiB to CH proof sizes for 2^{30} users, and as little as .6KiB for 1024 users. For convenience, we include these costs in the table.

In short, ATMS is best-suited for a small number of parties, whereas Mithril, CCCK and Telescope scale better. Mithril best "compresses" the signer set before transmission and thus wins on communication. On the other hand, CCCK performs that compression after the fact, and Telescope at both ends, with both revealing fewer signatures than the concatenation version of Mithril, PS^C.

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 [51]. Further optimizations such as Non-interactive proofs of proof-of-work (NIPoPoWs) [42] and flyclient [16] drastically reduce the number 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 [3, 31] are orthogonal to our work: they describe how a single user can privately prove eligibility, whilst we tackle eligibility over multiple users. Vault [44] 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 [29] uses a two layer solution tailored to blockchain bootstrapping, where one layer proves epoch transitions and the other aggregates over multiple epochs. Their system is highly efficient, but requires stronger setup assumptions to utilize SNARKS. Similarly, the subsequent works of Das et al. [23] and Garg et al. [32] make use of customized snarks to produce weighted threshold signatures with a trusted setup. Agrawal et al. [1] introduce an efficient interactive bootstrapping solution with some online requirements. It is orthogonal to (and compatible with) our work: e.g. their solution can be applied to a chain of Mithril certificates.

Table 2. Notation used in this work

Numbering	
N	Number of users registered.
m	Number of lotteries to be held.
k	Number of lotteries to be won for a certificate to be accepted.
Registration	
mvk_i	Verification key of user i.
stake_i	Stake held by user i.
AVK	Aggregate key of N users, defined as the root of a Merkle Tree with $reg_i = (mvk_i, \mathsf{stake}_i)$ as leaves.
Messages	
mesg	A message for the STM, in the form of $\mathsf{mesg} = (\mathsf{topic}, \mathsf{body})$
topic	Part of a mesg, used to determine eligibility.
body	Part of a mesg, not used to determine eligibility.
msg	The message passed to the underlying unique signature primitive, as a function of mesg and AVK.
Sampling	
$\phi(\mathsf{stake}_i)$	A function mapping the stake stake_i of an individual user, or set of users to the probability of wining one of the lotteries.

2 Preliminaries

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. $\boldsymbol{b} := (b_1, \ldots, b_n)$. We require the DL and co-CDH problems to be difficult in this setting.

Group Setting We require a pairing-friendly elliptic curve E on $\mathbb{F}_p$, forming groups $\mathbb{G}_1, \mathbb{G}_2$ of order q, with pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. We use g_1, g_2 to refer to generators of $\mathbb{G}_1, \mathbb{G}_2$ respectively.

2.1 Group setting assumptions

Definition 1 (The Discrete log Problem). *For a group $\mathbb{G} = \langle g \rangle$ of order q, and an adversary $\mathcal{A}$ we define $Adv_{\mathbb{G}}^{dl}$ 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 $Adv_{\mathbb{G}}^{dl}$ is negligible for all probabilistic polynomial time (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 $Adv_{\mathbb{G}_1,\mathbb{G}_2}^{co-CDH}$ 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 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 PS^B, through the **optional** use of a complexity leveraging argument.

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 $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, and returns system parameters $\mathsf{Param} = (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, q, e, \mathbb{G}_T)$.

We optionally require a group $\mathbb{G}_H$ of order p so that E can be embedded in $\mathbb{G}_H$, and additionally that the structure of E is compatible with the Elligator [9] or Elligator squared [58] representation functions. We require E to be pairing-friendly due to our choice of signature scheme. Compatibility with Elligator depends on our choice of dense mapping. In that case, we set $\mathsf{Param} = (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, q, e, \mathbb{G}_T, \mathbb{G}_H, g_h, p)$.

Hash functions We need hash functions $H_{\mathbb{G}_1} : \{0,1\}^* \to \mathbb{G}_1$, $H_q : \{0,1\}^* \to \mathbb{Z}_q$ modeled as random oracles, producing group elements in the corresponding groups for use with our unique signature scheme and mapping. For batching, we also use a truncated version of H_q, $H_\lambda : \{0,1\}^* \to \mathbb{Z}_{2^\lambda}$.

We also require a collision resistant hash function H_p on $\mathbb{F}_p$ to produce Merkle trees. Depending on our choice of a proof system (see Sect. 5.1), we can opt to use a prime p and an arithmetic friendly hash that is believed to be collision resistant, such as Poseidon [37] to instantiate H_p when using an arithmetic proof system that is efficient for $\mathbb{F}_p$. If the proof system evaluates H_p only natively, we can set p to a large power of 2 and opt to use any collision resistant hash.

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.

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

$\mathsf{MT.Check}(T, N, v, i, \boldsymbol{p})$: 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 bindingin 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 collision when i_k is 0, and $(p_k, h_k), (p'_k, h'_k)$ when it is not.*

Weighting Function Looking forward, we will use the concept of weights to randomly assign eligibility to participants. As we want eligibility to be calculated independently, a simple linear weighting is not desirable. Take 2 parties of equal weight w_0, each with an assigned 10% probability of eligibility. Between them, they will have a 19% probability of at least one of them being eligible, rather than 20% if they merge. Like Ouroboros [24], we use the function $\phi(w) = 1 - (1 - f)^w$ to assign success probabilities to weights $w \in [0, 1]$. The value $\phi(1) = f$ is a tuning parameter, representing the success probability of the total 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. More concretely, we have that $\phi(a + b) = 1 - (1 - \phi(a)) \cdot (1 - \phi(b))$, that is, the probability of success assigned to one party with stake $a + b$ is equal to the probability that at least one of two independent parties with stakes a, b respectively achieves success.

Non Interactive Proof Systems In our construction, we use a proof system to allow a prover to prove statement x is true by demonstrating she knows a witness w such that $R(x, w)$ is true.

Bulletproofs Bulletproofs [15] are an efficient proof system with transparent setup where a relation is represented as an arithmetic circuit. For a fixed relation R, and system parameters Param, we refer to the reference string setup, prover and verifier algorithms as $\mathsf{PS}^B.\mathsf{RS} \leftarrow \mathsf{PS}^B.\mathsf{S}(\mathsf{Param})$ $\pi_C \leftarrow \mathsf{PS}^B.\mathsf{P}(\mathsf{PS}^B.\mathsf{RS}, x, w)$, $0/1 \leftarrow \mathsf{PS}^B.\mathsf{V}(\mathsf{PS}^B.\mathsf{RS}, x, \pi_C)$, where x, w refer to the statement and witness respectively. Bulletproofs are complete andknowledge sound via witness-extended emulation.

A concatenation based proof system

A concatenation based proof system The proof system PS^C consists of releasing the witness w and letting the verifier check if $R(x, w) = 1$. Looking forward, w will be a concatenation of individual signatures, hence the name. Concretely, we have:

$\mathsf{PS}^C.\mathsf{S}(1^\lambda)$: Return $\mathsf{PS}^C.\mathsf{RS} := \bot$
$\mathsf{PS}^C.\mathsf{P}(\mathsf{PS}^C.\mathsf{RS}, x, w)$: Return w
$\mathsf{PS}^C.\mathsf{V}(\mathsf{PS}^C.\mathsf{RS}, x, \pi)$: Return $R(x, w)$

3 Unique Signature Scheme with Dense Mappings

Unique signature schemes [46, 26, 36] guarantee that for any given message m, a user associated with a verification key vk is only able to produce exactly one *valid* signature σ. This enables predicating eligibility by evaluating our dense mapping on σ.

We use a variant of MSP-PoP, a multisignature based on Boneh Lynn Shacham (BLS) signatures [12] with proofs of possession (PoPs) as described in [11, 53].

- $\mathsf{MSP.Gen}(\mathsf{Param})$: $sk \leftarrow \mathbb{Z}_q; mvk \leftarrow g_2^{sk}$;
  $\kappa_1 \leftarrow H_{\mathbb{G}_1}(\text{“PoP”}\|mvk)^{sk}; \kappa_2 \leftarrow g_1^{sk}$. Return secret key sk, verification key mvk and proof of 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 $\sigma \leftarrow H_{\mathbb{G}_1}(\text{“M”}\|msg)^{sk}$.
- $\mathsf{MSP.Ver}(msg, mvk, \sigma)$: 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 $ivk = \prod mvk_i$.
- $\mathsf{MSP.ASig}(\boldsymbol{\sigma})$: Takes as input a vector $\boldsymbol{\sigma}$ and returns $\mu \leftarrow \prod_1^d \sigma_i$.
- $\mathsf{MSP.BKey}(\mathbf{mvk}, \boldsymbol{e_\sigma})$: Takes a vector $\mathbf{mvk}$ of (previously checked) verification keys and weighting seed $e_{\boldsymbol{\sigma}}$, and returns an intermediate aggregate public key $ivk = \prod mvk_i^{e_i}$, where $e_i \leftarrow H_\lambda(i, e_{\boldsymbol{\sigma}})$.
- $\mathsf{MSP.BSig}(\boldsymbol{\sigma})$: Takes as input a vector of signatures $\boldsymbol{\sigma}$ and returns $(\mu, e_{\boldsymbol{\sigma}})$ where $\mu \leftarrow \prod \sigma_i^{e_i}$, where $e_i \leftarrow H_\lambda(i, e_{\boldsymbol{\sigma}})$ and $e_{\boldsymbol{\sigma}} \leftarrow H_p(\boldsymbol{\sigma})$.

The MSP scheme has been shown to be *complete* and *unforgeable* in [53] and [12]. We will redo the unforgeability proof as a number of differences are important for out application. First, in the definition of Boneh et al. [12], there exists only a single honest user so there is no possibility of the adversary issuing a singing query on the message targeted by the forgery: if a message has been queried, it becomes ineligible for the adversary to win with. With multiple honest users however, it becomes possible that the adversary has honest user a sign a message and produces a forgery on account of honest user b. Second, instead of the ψ isomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$ used by [53] and [12], we instead add a second element to the Proof of Possession. We present our security definition and corresponding unforgeability proof for our variant of MSP in Section 3.1.

The MSP.Check function is used to verify that the proofs of possession $\boldsymbol{\kappa}$ attached to a public key mvk are correct. The scheme operates as a standard multisignature, aggregating keys via MSP.AKey and signatures via MSP.ASig.

We also use the MSP.BKey and MSP.BSig functions, which enforce more stringent checking than that of standard multisignatures by utilizing the short random exponent batching of Bellare et al. [5]. The difference from standard multisignature aggregation (via MSP.AKey and MSP.ASig), is that the randomized check will fail with overwhelming probability if any of the individual signatures is invalid, whereas standard aggregation allows for spurious individual signatures as long as they sum up to the correct aggregate. Furthermore, MSP.BKey uses a weighting seed $e_{\boldsymbol{\sigma}}$ as input; in practice this is produced by the signature set to be verified and cannot be run ahead of time. In our use case, this can be overcome by having MSP.BKey be evaluated inside a proof system.

3.1 Security of MSP

Definition 6. *We say that a signature scheme is unforgeable in aggregate if any PPT adversary $\mathcal{A}$ wins the following game with only negligible probability.*

- *The Challenger runs $\mathsf{Param} = (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, q, e, \mathbb{G}_T) \leftarrow \mathsf{Setup}(1^\lambda)$.*
- *The Adversary $\mathcal{A}$ selects a number of honest users n.*
- *The Challenger provides the verification keys mvk_i, for $i \in [n]$ and proofs of possession $\boldsymbol{\kappa}_i$, for $i \in [n]$ of the honest users to the adversary.*
- *The Challenger allows the adversary to issue (individual) singing queries on any message and on the behalf of any user.*
- *The Adversary outputs a tuple $(m^*, \sigma^*, \boldsymbol{mvk}^*, \boldsymbol{\kappa}^*)$.*
- *The Adversary wins if and only if:*
  1. *The vectors $\boldsymbol{mvk}^*$ and $\boldsymbol{\kappa}^*$ have the same length, l.*
  2. *$\mathsf{MSP.Check}(mvk_i^*, \boldsymbol{\kappa}_i^*) = 1$ for $i \in [l]$.*
  3. *There exists at least one index $j \in [l]$ such that mvk_j corresponds to an honest user, and the message m has not been queried w.r.t. mvk_j in the signing oracle.*
  4. *$\mathsf{MSP.Ver}(m^*, ivk^*, \sigma^*) = 1$, where $ivk^* \leftarrow \mathsf{MSP.AKey}(\boldsymbol{mvk}^*)$.*

Theorem 1. *Our unique signature scheme is unforgeable in aggregate assuming the hardness of the co-CDH problem.*

Proof. We will describe a simulator that uses a forger $\mathcal{A}$ for the signature scheme in order to solve a co-CDH instance. The simulator works as follows:

We assume $\mathcal{A}$ 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 index j^* 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 other users, we create and store their keys normally.

Random Oracle queries We distinguish between two classes of queries: PoP-related queries that we use to extract information about adversarial users, and message queries that we use to enable use to forge signatures (except for the q^*-th query).

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 $\mathcal{L}_{msg}$.

Signing queries 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 queried. 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^*. As the signature scheme is unique, the adversary cannot discern which way was used to answer.

If the adversary wins, the simulator checks to see if P^* is included in mvk^*. If it is, we are able to isolate $\sigma^*_{j^*}$ from the aggregate signature σ by calculating the signature of every other user included in the key. The signature $\sigma^*_{j^*}$ 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 our assumption for the difficulty of the co-CDH problem.

We note that the proof covers the case of standard single-signer forgery when $l = 1$.

Lemma 2. *Let* $\boldsymbol{mvk} \in \mathbb{G}_2^n$*, and* $\boldsymbol{\sigma} \in \mathbb{G}_1^n$ *be two vectors of verification keys and signatures. Then, for any message msg, the check* $\prod_i \mathsf{MSP.Ver}(msg, mvk_i, \sigma_i)$ *is equivalent to* $\mathsf{MSP.Ver}(msg, \mathsf{MSP.BKey}(\boldsymbol{mvk}, \boldsymbol{e_\sigma}), \mathsf{MSP.BSig}(\boldsymbol{\sigma}))$ *where* $\boldsymbol{e_\sigma} \leftarrow H_p(\boldsymbol{\sigma})$*, except with negligible probability, taken over the outputs of the random oracle* H_λ*.*

Proof. Let $h \leftarrow H_{\mathbb{G}_1}(\text{"M"}\|msg)$. Without loss of generality, we assume h is not the identity element of $\mathbb{G}_1$ (in which case the result holds trivially). Let $\mu_i = \log_{g_2} mvk_i$ and $\psi_i = \log_h \sigma_i$. Then, the check $\prod_i \mathsf{MSP.Ver}(msg, mvk_i, \sigma_i)$ can be rewritten as $\prod_i EqCheck(\mu_i, psi_i))$. Where $EqCheck(x, y) = 1$ if $x = y \bmod q$ and 0 otherwise. The second check can be rewritten as $EqCheck(\sum_i e_i \cdot \mu_i, \sum_i e_i \cdot \psi_i) = EqCheck(\sum_i e_i \cdot (\mu_i - \psi_i), 0)$.

We can consider $\sum_i e_i \cdot (\mu_i - \psi_i)$ as a degree 1 multivariate polynomial with variables e_i and coefficients $(\mu_i - \psi_i)$. Thus, by the Schwartz-Zippel lemma, unless all the coefficients are 0, the equality test will pass with probability only $\frac{1}{q}$.

3.2 Dense Mappings for Unique Signatures.

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.

The works of [58, 9] 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 7. *A deterministic function $M : \mathbb{G}_1 \to \mathbb{Z}_p \cup \{\perp\}$ 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, when x is uniform 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 unique signature scheme as follows.

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

In Section 7 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 5.2 we use a random oracle $H : \{0,1\}^* \to \mathbb{Z}_p$ for the mapping as: $M^R_{msg,\mathsf{index}}(\sigma) := H(\text{“map”}\|msg\|\mathsf{index}\|\sigma)$.

In Section 7 we show how to construct a dense mapping $M^E_{msg,\mathsf{index}}(\sigma)$ based on Elligator Squared, which avoids oracle calls on witness-specific data.

4 Ideal Functionality for Stake Based Threshold Multisignatures

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

$\mathcal{F}^{\phi}_{\mathsf{STM}}(\mathcal{P}, m, k)$ initializes the variable Allow to 1, and table K to be empty and proceeds as follows:

- Upon receiving $(\mathsf{Register}, sid)$ 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 $(\mathsf{Registered}, sid, P_i)$ to $\mathcal{A}$ and output $(\mathsf{Registered}, sid)$ to P_i.
- Upon receiving (Start, sid) from the adversary $\mathcal{A}$:
  1. Set Allow to 0.

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

We will now describe a stake based threshold multisignature functionality similar to the PoS Anonymous Selection of [3]. The messages mesg to be signed are of the form $\mathsf{mesg} = (\mathsf{topic}, \mathsf{body})$, where the eligibility of a signer is a function of their stake and topic, but not body.

Separation of topic, and body. This distinction can be used to limit the feasibility of an adversary trying to "grind" by trying different messages in the hope they are indeed eligible for one. By prescribing that topic must follow a narrowly defined format the adversary's options are limited. Alternatively, body may be left blank, simplifying the construction and also making it harder for adversaries to leverage dynamic corruptions: a user who is eligible to sign for (topic=President, body=Alice) can be corrupted to sign for (topic=President, body=Bob) as eligibility will persist. By mandating that the body is blank, a user who signs (topic=President,Alice) is not more or less likely to be able to sign (topic=President,Bob). We revisit this distinction and tradeoffs in Section 6.5.

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 $k = m \cdot \phi(\frac{1}{2})$. 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 works 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 is split 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.

At a high level, the ideal functionality allows users to call EligibilityCheck and CreateSig to check whether they are able to sign, and if so, produce signatures. Both calls are parametrized by an index value representing which of the m parallel lotteries the user is referring to. In practice, users will check all lotteries. Eligibility is decided by the adversary under the condition that corrupt users cannot form a quorum (i.e eligibility over at least k indices). Since corruptions are static, this implies that corrupt users can never succeed in producing an aggregate signature. Aggregate will only produce an aggregate signature if there exist k individual signatures for the same mesg over different index values, and VerifyAggregate requires that an aggregate signature came from Aggregate, or that enough individual signatures have or could have been produced to support it. That is, we allow that aggregation can be performed by any party as there is no private information required.

A trivial realization. If we assume uniform stake distribution, we can realize the above using only signatures: we set $m = N$, and fix the eligibility function to $\mathcal{E}(\mathsf{topic}, P_i, \mathsf{index}) = 1$ iff $i = \mathsf{index}$ and 0 otherwise. CreateSig is implemented by signing. Verification only accepts signatures for index i from user P_i. Aggregate is implemented by concatenating signatures and signer identities. VerifyAggregate then consists of parsing, and counting the number of valid signatures.

While simple, the aggregate signatures have 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 expand our treatment to cover the more general case, and use dense mappings as a form of lottery so that only a few stakeholders need to participate at any one time.

Our notion of lottery based sampling alleviates both of these concerns: stake is used to determine the odds of wining with no need of duplication. Additionally, it operates as a filter, reducing both the number of signatures communicated to the aggregator as well as the size of the certificate.

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

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

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

5 A Stake Based Threshold Multisignature Scheme

Protocol $\mathit{\Pi}$.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. 3. The STM Protocol $\mathit{\Pi}$.STM in the initialisation phase.

In Figures 3 and 6 we present a protocol $\mathit{\Pi}$.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. The functionality $\mathcal{F}_{\mathsf{RS}}(\mathcal{P})$ provides access to a reference string, whereas $F_{\mathsf{Kr}}^{\psi_0}(\mathcal{P})$ provides key registration so that a key can only be used by one party. Both hybrid functionalities we use are practical to realize in common applications. For $\mathcal{F}_{\mathsf{RS}}$, the group can be realistically hardcoded, leaving only the proof system reference string. We present functionality $\mathcal{F}_{\mathsf{Kr}}^{\psi}(\mathcal{P})$ in Figure 4. The parameter ψ is a function that checks public keys by calling MSP.Check. Functionality $\mathcal{F}_{\mathsf{RS}}(\mathcal{P})$ is presented in Figure 5.

The Key Registration 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 $(\mathsf{Register}, sid, vk)$ 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 $(\mathsf{RegKey}, sid, 1)$ to P_i.
- Upon receiving $(\mathsf{Retrieve}, sid, P_i)$ on behalf of party P_j:
  1. $P_j \notin \mathcal{P}$, or $K(P_i)$ is not defined, output $(\mathsf{Retrieve}, sid, P_i, \perp)$ to P_j.
  2. Otherwise, output $(\mathsf{Retrieve}, sid, P_1, K(P_i))$ to P_j
- Upon receiving $(\mathsf{CloseRegistration}, sid)$ on behalf of the adversary $\mathcal{A}$:
  1. Set Allow to 0.
  2. For each $P_i \in \mathcal{P}$, send $(\mathsf{RetrieveAll}, sid, K)$ to P_i.

Fig. 4. The Key Registration functionality $\mathcal{F}_{\mathsf{Kr}}^{\psi}(\mathcal{P})$, with key checking function ψ.

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

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

Fig. 5. The Reference String functionality $\mathcal{F}_{\mathsf{RS}}(\mathcal{P})$ interacting with the adversary $\mathcal{A}$.

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, $\mathcal{F}_{\mathsf{Kr}}$ can be realized by means of a broadcast channel which can be implemented via a blockchain.

As with the ideal functionality, the protocol operates in two phases. The initialisation phase is presented in Figure 3 and the operation phase in Figure 6. The protocol operates on a fixed player list $\mathcal{P} = (P_i, \mathsf{stake}_i)$, where $|\mathcal{P}| = n$, a scaling function $\phi(w)$, a lottery 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 unique signature scheme 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 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 Sections 5.3 and 5.2, and compare their efficiency in Section 6.1.

5.1 The Relation $\mathcal{R}_{avk}$

Our proof systems operate on language $\mathcal{L}_{avk}$, i.e we prove knowledge of a witness w such that statement x holds, i.e. $\mathcal{R}_{avk}(x, w) = 1$. Concretely, statements are $x = (\mathsf{AVK}, ivk, ivk_{\mathsf{body}}, \mu, e_{\boldsymbol{\sigma}}, \mathsf{mesg})$ and witnesses are of the form $w = (mvk_i, \mathsf{stake}_i, \boldsymbol{p}_i, ev_i, \sigma_i, \mathsf{index}_i)$ for $i \in \{1 \ldots k\}$. The relation $\mathcal{R}_{avk}$ is parametrized on $N, m, k, \phi()$, which are public. $R_{avk}(x, w) = 1$ if and only if the following hold:

- $ivk = \mathsf{MSP.BKey}(\mathbf{mvk}, e_{\boldsymbol{\sigma}})$ and $ivk_{\mathsf{body}} = \mathsf{MSP.AKey}(\mathbf{mvk})$.
- $(\mu, e_{\boldsymbol{\sigma}}) = \mathsf{MSP.BSig}(\boldsymbol{\sigma})$.
- $\forall i : \mathsf{index}_i \leq m$ and $\forall i \neq j : \mathsf{index}_i \neq \mathsf{index}_j$.
- For $i \in \{1 \ldots k\}$: $(mvk_i, \mathsf{stake}_i)$ lies in Merkle tree AVK, N following path $\boldsymbol{p}_i$.
- For $i \in \{1 \ldots k\}$: $\mathsf{MSP.Eval}(\mathsf{topic}, \mathsf{index}_i, \sigma_i) = ev_i$
- For $i \in \{1 \ldots k\}$: $ev_i \leq \phi(\mathsf{stake}_i)$

Contradictions for $\boldsymbol{\mathcal{R}_{avk}}$ Due to the use of Merkle trees in the *avk* relation, there may exist numerous "alternative" openings for given a root, enabling the adversary to produce a proof by means of using such an opening as part of the witness. Obviously, any such witness combined with our known opening of the Merkle tree would contradict the collision resistance of H_p. However, it might not be possible to extract that witness from a proof in the UC setting. For this reason, we will define a class of statements to be *contradictory* if they are not consistent with our known opening of the Merkle tree. We then need to show for our proof system, that proofs of such statements can only be produced with negligible probability.

Definition 8. *For $\mathcal{R}_{avk}$, given $N, m, k, \phi()$, we say that statement $x = (\mathsf{AVK}, ivk, \mu, e_{\boldsymbol{\sigma}}, msg)$ 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:*

- *$ivk = \mathsf{MSP.BKey}(\boldsymbol{mvk}_{p_j}, \boldsymbol{\sigma}_{p_j})$.*
- *$\forall i \neq j : s_i \neq s_j$.*
- *For $i = 1..k$: $ev_{p_j,t_j} \leq \phi(\mathsf{stake}_{p_j})$*

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

We will propose two constructions: one based on bulletproofs which may also be used as a template for other circuit-based systems, and a simpler system based on releasing the witness. In the first case we let $\mathsf{PS} = \mathsf{PS}^B$ and $M = M^E$, and in the second, $\mathsf{PS} = \mathsf{PS}^C$ and $M = M^R$.

5.2 An Instantiation via Concatenation Proofs

The proof system PS^C consists of releasing the witness w and letting the verifier check if $R(x, w) = 1$. Looking forward, w will be a concatenation of individual signatures, hence the name. Contradiction soundness is trivial for PS^C, as the full witness is present without rewinding. We use mapping $M = M^R$ for PS^C.

Contradiction Soundness for PS^C

Lemma 3 (Contradiction Soundness for PS^C). *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:*

$$Pr[\mathsf{AVK} \leftarrow \mathsf{MT.Create}(mvk_i, \mathsf{stake}_i), (ivk^*, \mu^*, msg^*, \pi)^* \leftarrow \mathsf{P}^*(\sigma, \mathsf{AVK}) : \\ \mathsf{PS}^C.\mathsf{V}(\bot, x, \pi^*) = 1 \text{ where } x = (\mathsf{AVK}, ivk^*, \mu^*, msg^*),]$$

5.3 An instantiation based on Bulletproofs

We have ensured that the overall design is modular so that the proof system can be changed with relatively few changes. Nonetheless, some issues require attention. Specifically, for the bulletproof based system PS^B, we need to (a) ensure performance by matching the arithmetic of the proof system to the curve arithmetic of signatures, (b) ensure no oracle calls are required inside the circuit and (c) establish contradiction soundness because standard soundness is inadequate (due to spurious hash preimages), and rewinding cannot be invoked by the simulator.

Curve Choices In order to efficiently produce proofs about group elements of $\mathbb{G}_1, \mathbb{G}_2$ which are based on a pairing friendly curve E on $\mathbb{F}_p$, we *additionally* require a group $\mathbb{G}_H$ of order p so that E can be embedded in $\mathbb{G}_H$, and additionally that the structure of E is compatible with the Elligator [9] or Elligator squared [58] representation functions. The Pluto-Eris [38] cycle of curves satisfies these properties.

Hash use inside Circuits We note that $H_{\mathbb{G}_1}, H_q$ are not evaluated inside the circuit-based proof, allowing us to study the security of either construction under the random oracle model [6] with no hindrance to the proof. This is relevant, as Baldimtsi et al. [3] point out: if we concretely represent the hash (e.g. as a circuit) to construct the appropriate statement proof system, we can no longer invoke the random oracle model.

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. BLS aggregation is somewhat underutilized as we require individual signatures to verify the mapping. However, we are able to batch verify efficiently using short random exponents.

Furthermore, we use a mapping $M^E_{msg,\mathsf{index}}(\sigma) := R(msg, \sigma^{H_q(msg,\mathsf{index})}, \mathsf{index})$, based on Elligator squared [58]. We not that R is a deterministic representation function, so that we do not need to call the random oracle with elements of the witness. We provide a full description of the mapping in Section 7.

Contradiction Soundness for PS^B We note our argument for proving contradiction soundness invokes rewinding to perform extraction, but said rewinding is performed on the **entire ensemble** of the UC simulator and the environment. I.e., if there exists an environment such that proofs of *contradictory* statements are produced with non-negligible probability, we are able to produce collisions for H_p . This *external* leveraging of rewinding is similar to that of Canetti et. al. [17] who perform rewinding outside the UC proof to assert an indistinguishability property inside it. The rewinding is *not* performed by the simulator in order to obtain information to continue execution (which would require that any extraction is straightline i.e. without rewinding). Rather, the implication that "there exists an efficient collision finder for H_p", is leveraged to bound the probability that the simulator fails after receiving a proof that it recognised as contradictory.

Using the witness extractors from [35, 2], we can prove that:

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

$$\begin{aligned} &Pr[\sigma \leftarrow \mathsf{PS}^B.\mathsf{RS}(1^\lambda), \mathsf{AVK} \leftarrow \mathsf{MT.Create}(mvk_i, \mathsf{stake}_i),\\ &\quad (ivk^*, \mu^*, msg^*, \pi^*) \leftarrow \mathsf{P}^*(\sigma, \mathsf{AVK}) :\\ &\quad \mathsf{PS}^B.\mathsf{V}(\sigma, x, \pi^*) = 1 \text{ where } x = (\mathsf{AVK}, ivk^*, \mu^*, msg^*)] \end{aligned}$$

Proof (Sketch). If P^* succeeds with non-negligible probability, we can use the witness 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.

5.4 Adversarial Eligibility

A core component in the security argument is proving that the adversary has a negligible probability of achieving eligibility across enough lotteries, and thus any success by the adversary would involve breaking at least one of the other underlying primitives. Towards that, we argue that if the adversary has probability p' of winning a single lottery, he will win an average of $m \cdot p'$ lotteries. We therefore set $k = mp$ high enough that the adversary will only win k lotteries with negligible probability. Let the adversary's stake be $\frac{1}{2} - a$. We now need to calculate the probability p' of the adversary wining a single lottery.

For honest parties, this would be simple to compute due to the properties of the weighting function: take for example two parties A, B with stakes a,b, their individual probability of wining a lottery will be by definition $p_a = \phi(a)$ and $p_b = \phi(b)$. The probability that either party wins will then be $1-(1-p_a)\cdot(1-p_b)$ which is $1-(1-\phi(a))\cdot(1-\phi(b))$. However, the latter term is exactly equal to $\phi(a+b)$. A crucial requirement for this to hold is that the probabilities of A and B wining the lottery are independent. For honest users, this is true as their keys are independent, and their signatures are deterministic functions of the message and their key.

Adversarial parties however, may try to produce correlated keys in the hope of increasing the odds of an adversarial party wining the lottery (one winer per lottery is sufficient so multiple wins are in effect "wasted"). For this reason, we require that either the mapping function enforces independence even if the keys are somehow correlated, or that the gain to the adversary is minimal.

Adversarial Eligibility for Concatenation proofs For the concatenation proof system PS^C, our mapping of signatures to eligibility is a simple random oracle, so the eligibility of different users is independent across the same lottery, even if the adversarial keys themselves are related, so that $p' = \phi(\frac{1}{2} - a)$.

Lemma 5. *For the mapping $M^R_{msg,\mathsf{index}}(\sigma) := H(\text{“map”}\|msg\|\mathsf{index}\|\sigma)$, the eligibility of each potential signer for any of the lotteries is independent of that of others, as long as the public keys of users are not repeated.*

Proof. The eligibility predicate for a potential signer with stake s_a is calculated by checking iff $\phi(s_a) > H(\text{“map”}\|msg\|\mathsf{index}\|\sigma)$. Due to the signature scheme being unique, the signatures of different voters cannot be equal [4], the value of each random oracle call is independent of all others, and thus the eligibility predicates of different users are always independent.

[4] Apart from a negligible fraction of pathological messages hashing to $1_{\mathbb{G}_1}$

Protocol Π.STM. Operation phase

- $\mathsf{EligibilityCheck}$: On input $(\mathsf{mesg}, \mathsf{index})$, user P_i runs: Let $\overline{\mathsf{topic}} \leftarrow$ “0”$||\mathsf{AVK}||\mathsf{topic}$, $\sigma \leftarrow \mathsf{MSP.Sig}(msk, \overline{\mathsf{topic}})$; $ev \leftarrow \mathsf{MSP.Eval}(\overline{\mathsf{topic}}, \mathsf{index}, \sigma)$. Return 1 if $ev < \phi(\mathsf{stake})$, else return 0.
- $\mathsf{CreateSig}$: On input $(\mathsf{mesg}, \mathsf{index})$: If $\mathsf{EligibilityCheck}(\mathsf{mesg}, \mathsf{index})$ is 1, then let $\overline{\mathsf{topic}} \leftarrow$ “0”$||\mathsf{AVK}||\mathsf{topic}$; $\sigma \leftarrow \mathsf{MSP.Sig}(msk, \overline{\mathsf{topic}})$; $\sigma_{\mathsf{body}} \leftarrow \mathsf{MSP.Sig}(msk,$ “1”$||\overline{\mathsf{topic}}||\mathsf{body})$ and produce an individual signature $\pi = (\sigma, \sigma_{\mathsf{body}}, reg_i, i, \boldsymbol{p}_i)$, 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 (mesg), parse $\pi = (\sigma, \sigma_{\mathsf{body}}, reg_i, i, \boldsymbol{p}_i)$. Parse reg_i as $(mvk_i, \mathsf{stake}_i)$. Check that reg_i corresponds to party P_i, let $\overline{\mathsf{topic}} \leftarrow$ “0”$||\mathsf{AVK}||\mathsf{topic}$; $ev \leftarrow \mathsf{MSP.Eval}(\overline{\mathsf{topic}}, \mathsf{index}, \sigma)$ 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 $\mathsf{MSP.Ver}(\overline{\mathsf{topic}}, mvk_i, \sigma) \wedge \mathsf{MSP.Ver}($“1”$||\overline{\mathsf{topic}}||\mathsf{body}, mvk_i, \sigma_{\mathsf{body}})$.
- $\mathsf{Aggregate}$: On input vectors $\boldsymbol{P}, \boldsymbol{\pi}, \mathbf{index}$ and message (mesg), parse $\boldsymbol{P}, \boldsymbol{\pi}$ and $\mathbf{index}$ as a vector $P_j, \pi_j, \mathsf{index}_j$ of size k, let $\overline{\mathsf{topic}} \leftarrow$ “0”$||\mathsf{AVK}||\mathsf{topic}$ and run $\mathsf{Verify}(P_j, \mathsf{index}_j, \mathsf{mesg}, \pi_j)$.
  If parsing or checking fails, return $\perp$. If any $\mathsf{index}_j = \mathsf{index}_i$ for $j \neq i$ return 0. Otherwise, parse $\pi_j = (\sigma_j, \sigma_{\mathsf{body},j}, reg_j, i_j, \boldsymbol{p}_j)$ and reg_j as $(mvk_j, \mathsf{stake}_j)$. Let $ivk \leftarrow \mathsf{MSP.BKey}(\mathbf{mvk}, \boldsymbol{\sigma})$,$ivk_{\mathsf{body}} \leftarrow \mathsf{MSP.AKey}(\mathbf{mvk})$, $\mu \leftarrow \mathsf{MSP.BSig}(\boldsymbol{\sigma})$, $\mu_{\mathsf{body}} \leftarrow \mathsf{MSP.ASig}(\boldsymbol{\sigma}_{\mathsf{body}})$, set $x = (\mathsf{AVK}, ivk, \mu, e_{\boldsymbol{\sigma}}, \mathsf{mesg})$ and $\boldsymbol{w} = (mvk_j, \mathsf{stake}_j, \boldsymbol{p}_j, ev_j, \sigma_j, \mathsf{index}_j)$ for $j \in \{1 \ldots k\}$. Then, $\pi_{avk} \leftarrow \mathsf{PS.P}(\mathsf{PS.RS}, x, \boldsymbol{w})$. Return $\tau = (ivk, \mu, e_{\boldsymbol{\sigma}}, ivk_{\mathsf{body}}, \mu_{\mathsf{body}}, \pi_{avk})$.
- $\mathsf{VerifyAggregate}$: On input (τ, mesg), parse $\tau \rightarrow (ivk, \mu, e_{\boldsymbol{\sigma}}, ivk_{\mathsf{body}}, \mu_{\mathsf{body}}, \pi_{avk})$, check that $\mathsf{PS.V}(\mathsf{PS.RS}, (\mathsf{AVK}, ivk, \mu, e_{\boldsymbol{\sigma}}, \mathsf{mesg}), \pi_{avk})$ is true. If parsing and checking is successful, let $\overline{\mathsf{topic}} \leftarrow$ “0”$||\mathsf{AVK}||\mathsf{topic}$ and return $\mathsf{MSP.Ver}(\overline{\mathsf{topic}}, ivk, \mu) \wedge \mathsf{MSP.Ver}($“1”$||\overline{\mathsf{topic}}||\mathsf{body}, ivk_{\mathsf{body}}, \mu_{\mathsf{body}})$.

Fig. 6. The STM Protocol Π.STM in the operation phase.

Adversarial Eligibility for Bulletproofs For PS^B, we show a weaker result: evaluations are independent across lotteries (Lemma 6), but correlated keys (and by extension, correlated signatures) may potentially produce correlated values. The reason for this is that for PS^B, the used mapping does not pass signatures through a random oracle: we cannot instantiate the oracle inside a circuit, and do not wish to send the signatures to the verifier. Instead, we pass $H_q(msg, \mathsf{index})$ through the oracle and use it as a "randomizer" for σ.

In theory, this allows the adversary to gain a slight advantage by stake splitting and exploiting the subadditivity of ϕ. However, in Lemma 7 we show that we can bound the Adversary's gain by a small value.

Lemma 6. *For the mapping $M^E_{msg,\mathsf{index}}(\sigma) := R(msg, \sigma^{H_q(msg,\mathsf{index})}, \mathsf{index})$, adversarial eligibility is independent across lotteries.*

Proof. We allow the adversary to control various parties each with stake s_i such that the keys of adversarial users may somehow be correlated. We consider the adversary to be eligible for a given index idx_0 iff at least one of the parties she controls is eligible for that index, i.e. $R(msg, \sigma^{H_q(msg,\mathsf{index})}, \mathsf{index}) < \phi(s_i)$. Consider a fixed message msg_0, and also fix the set of adversarial keys and stakes (the second restriction is implied in our application as the AVK of all public keys is appended to the message).

The eligibility of each adversarial party (and therefore the eligibility of the adversary in general) is then completely determined by the value of $H_q(msg_0, \mathsf{index})$, its stake and public key. We also observe, that the eligibility of the adversary in general can also be expressed as a (slightly more complex function) of $H_q(msg_0, \mathsf{index})$ and the set of adversarial keys and corresponding stakes.

As H_q is modelled to be a random oracle, the distribution of $H_q(msg_0, \mathsf{index})$ is independent across different values of index, therefore the adversary's eligibility is also independent across different values of index, as it is a fixed function of $H_q(msg_0, \mathsf{index})$.

Lemma 7. *For the mapping $M^E_{msg,\mathsf{index}}(\sigma) := R(msg, \sigma^{H_q(msg,\mathsf{index})}, \mathsf{index})$, and for fixed values of msg, idx, the probability of an adversary controlling a a fraction of the stake winning the lottery is bounded by $\phi(a)\cdot(1+c))$, where c is $f \cdot ln\left(\frac{1}{1-f}\right) - 1$.*

Proof. We know that the probability of a potential signer controlling a percentage s of stake is $\phi(s)$, where $\phi(s) = 1 - (1-f)^s$ and $f = \phi(1)$. An adversary controlling a a share can therefore split into n parties of stake s_i such that $\sum_{i=1}^n s_i = a$, so that party i succeeds with probability $\phi(s_i)$. We do not know that probabilities of the adversarial parties are indeed independent, but we can initially bound their joint probability by $\sum_{i=1}^n \phi(s_i)$. One difficulty here is that the value of this bound depends on the exact split chosen by the adversary.

As ϕ is subadditive (i.e. $\phi(a+b) < \phi(a) + \phi(b)$), we can further bound our initial bound by the limit $\lim_{n\to\infty} n \cdot \phi(a/n)$, which does not depend on the exact split.

We have that $\phi(x) = 1 - (1-f)^x$. Let $\kappa = (1-f)^a$. Then $\phi(a) = 1 - \kappa$ and $\phi(a/n) = 1 - \kappa^{1/n}$. Our limit is thus

$$\lim_{n\to\infty} n \cdot (1 - \kappa^{1/n}) = -\ln(\kappa)$$

We know that $\phi(a) = 1 - \kappa$, so $\frac{n\cdot\phi(a/n)}{\phi(a)}$ is bounded by $\frac{-\ln(\kappa)}{1-\kappa} = \frac{-a\ln(1-f)}{\phi(a)} \leq f \cdot ln\left(\frac{1}{1-f}\right)$ as $\phi(x) \geq x \cdot f$ in $(0,1)$. Thus $\frac{n\cdot\phi(a/n)}{\phi(a)}$ is bounded by $\phi(a)\cdot(1+c))$, where $c = f \cdot ln\left(\frac{1}{1-f}\right) - 1$.

In practical terms, the advantage of an adversary that can create correlations amongst keys is not large. Let $\phi_{\max}(a) := a \cdot ln\left(\frac{1}{1-f}\right)$. Then, for $f = .2$ we have that $\phi_{\max}(.3) \approx \phi(.31)$ and $\phi_{\max}(.4) \approx \phi(.419)$. For $f = .1$ we have that $\phi_{\max}(.3) \approx \phi(.305)$ and $\phi_{\max}(.4) \approx \phi(.409)$. In plain terms, for $f = .1$ we can replace a (potentially) corelating 40% adversary by a non-corelating 41% one.

Adversarial Eligibility over *k* Lotteries In the following lemma, we calculate the probability that an adversary with probability $p' = \phi(\frac{1}{2} - a)$ to win a single lottery, manages to be eligible over enough lotteries to form a certificate by winning at least k out of m different lotteries. Looking forward to Theorem 2, this would cause our simulation to fail as the ideal functionality will abort.

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 $\frac{p}{p'} = \frac{\phi(1/2)}{\phi(1/2-a)} \geq 1 + a$.

Lemma 8. *[Sampling Property] Let p' be the probability that the adversary succeeds in any single lottery, and $\phi(\frac{1}{2}) = p$. When $\frac{p}{p'} \geq 1 + a$, the eligibility matrix sampled by the simulator causes the functionality to abort with probability negligible in m. Furthermore, for $m = -(2+a)/(a^2 \cdot \phi(\frac{1}{2} - a))\ln(\varsigma)$, the probability of failure is at most ς.*

Proof. Each of the m columns of the matrix represents an independent trial in which 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. $p'm \leq \frac{k}{1+a}$. The functionality will thus abort only if the actual number of successes, X is greater than $1 + a$ times the mean.

By Chernoff bounds, the probability of aborting is: $\Pr[X > k] \leq \Pr[X > p'm \cdot (1+a)] \leq e^{\frac{-a^2 \cdot p' m}{2+a}}$. As $p' \neq 0$ by the definition of the ϕ function, the chance of aborting is negligible in m.

For the second part, rewriting m as $m = -(2+a)/(a^2 \cdot \phi(\frac{1}{2} - a))\ln(\varsigma)$, directly produces the required bound.

Proof. Each of the m columns of the matrix represents an independent trial in which 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. $p'm \leq$

$\frac{k}{1+a}$. The functionality will thus abort only if the actual number of successes, X is greater than $1+a$ times the mean.

By Chernoff bounds, the probability of aborting is: $\Pr[X > k] \leq \Pr[X > p'm \cdot (1+a)] \leq e^{\frac{-a^2 \cdot p'm}{2+a}}$. As $p' \neq 0$ by the definition of the ϕ function, the chance of aborting is negligible in m.

For the second part, rewriting m as $m = -(2+a)/(a^2 \cdot \phi(\frac{1}{2} - a)) \ln(\varsigma)$, directly produces the required bound.

As a corollary, for $m \geq \log^2 \lambda$, the above probability is negligible in λ.

5.5 Security Proof

In this section we show that our protocol realizes the ideal functionality of an STM. A core property is that the adversary is unable to create a valid certificate.

In the previous section we showed that the probability of an adversary with stake $\frac{1}{2} - a$ to achieve a quorum is negligible for appropriate values of k, m. To complete the proof we also need to show that the adversary cannot gain an advantage by means of breaking one of the primitives used in the protocol, or otherwise cause the simulation to fail.

Theorem 2. *Let $a < \frac{1}{2}$, $m \geq \log^2 \lambda$ and quorum parameter $k = m \cdot \phi(\frac{1}{2} + a)$. The protocol $\Pi.\mathsf{STM}$ of Section 5 realizes $\mathcal{F}^{\phi}_{\mathsf{STM}}(\mathcal{P}, m, k)$ against adversaries with stake at most $\frac{1}{2} - a$ in the $\mathcal{F}_{\mathsf{RS}}(\mathcal{P}), \mathcal{F}^{\psi_0}_{\mathsf{Kr}}(\mathcal{P})$-hybrid model, under the leveraged[5] 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:

- $\mathsf{Oracle\ Calls}$: 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 $\sigma = k_1^{(1/r)r'}$.
- $\mathsf{Register}$: The simulator runs the key generator $\mathsf{MSP.Gen}(\mathsf{Param})$ normally, returns the verification key vk_i and stores the private key sk_i.
- RegKey: The simulator runs the key verification algorithm $\mathsf{MSP.Check}$ and returns the output.
- $\mathsf{EligibilityCheck}$: The simulator can evaluate eligibility for all participants, by signing on behalf of each user and then sets ideal functionality accordingly. This distribution is the same as in real world, apart from potentially causing the functionality to abort, but that only occurs with only negligible probability.

[5] PS^C does not require leveraging, we include the assumption for uniformity with PS^B.

- CreateSig: For honest users the simulator creates signatures normally. For malicious ones, it uses random oracle programmability 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 "MSP forgery". If a signature verifies but the corresponding user is not eligible, the simulator fails with output "individual signature verification 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 proof verification 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 (due to Lemma 8). Thus, H_2 and H_3 are also statistically indistinguishable.

In H_4 the ideal functionality and simulator are used for 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 "MSP forgery" or "individual signature verification failure". In Lemma 10 we show that "MSP forgery" reduces to the co-CDH problem and in Lemma 9 we show that "individual signature verification 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 proof verification failure" but is otherwise identical to the previous execution. The output "aggregate proof verification failure" happens with negligible probability due to Lemma11. At this point, it suffices to point out that H_5 is identical to the environment interacting with the simulator and the ideal functionality.

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. We note that the concatenation proof system does not require this change, as the user identities are included.

5.6 Supporting Lemmas

Lemma 9. *The simulator outputs "individual signature verification failure" with negligible probability.*

Proof. The simulator only outputs the above message if an adversarial signature $\pi = (\sigma^*, {\sigma_{\mathsf{body}}}^*, reg_i^*, i, \boldsymbol{p}_i)$ 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.

Proof. The simulator only outputs the above message if an adversarial signature $\pi = (\sigma^*, {\sigma_{\mathsf{body}}}^*, reg_i^*, i, \boldsymbol{p}_i)$ 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 10. *The simulator outputs "MSP forgery" with negligible probability.*

Proof. The simulator only outputs "MSP forgery" if the environment provides a valid signature for an honest user without calling $\mathsf{CreateSig}$. We observer that as σ_{body} must be a signature on $\overline{\mathsf{topic}}||\mathsf{body}$, it is not possible for the environment to maul σ, σ_{body} from different topics into a new signature (except with negligible probability via a hash collision). Thus, at least one of the σ, σ_{body} values provided must be a "fresh" forgery. W.l.o.g we assume the forgery lies in σ.

We will show that we can adapt the simulation so that if "MSP forgery" 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 "MSP forgery", 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.

Proof. The simulator only outputs "MSP forgery" if the environment provides a valid signature for an honest $\overline{\text{user}}$ without calling CreateSig. We observer that as σ_{body} must be a signature on $\overline{\mathsf{topic}}||\mathsf{body}$, it is not possible for the environment to maul σ, σ_{body} from different topics into a new signature (except with negligible

probability via a hash collision). Thus, at least one of the σ, σ_{body} values provided must be a "fresh" forgery. W.l.o.g we assume the forgery lies in σ.

We will show that we can adapt the simulation so that if "MSP forgery" 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 quer-

ried. 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 "MSP forgery", 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 11. *The simulator outputs "Aggregate proof verification failure" with only negligible probability.*

Proof. We distinguish between two cases:

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

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 proof verification 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, as well as the e_i cofactors using $\boldsymbol{\sigma}$. 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.

Proof. We distinguish between two cases:

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

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 proof verification 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, as well as the e_i cofactors using $\boldsymbol{\sigma}$. 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.

6 Parameter Selection and Implementation Considerations

In the following, we investigate the effects of different parameter selections to the size of certificates. We also suggest a method of using multiple parameter sets concurrently (*concurrent hybrids*) so that the best feasible set is used by the aggregator. We also investigate the costs of producing, transmitting and verifying individual as well as aggregate signatures. We also investigate the tradeoffs between controlling eligibility by means of either the entirety of a message or a (potentially structured) segment of it.

6.1 Parameter Choices

Quorum parameters Due to Lemma 8, the probability of an adversarial minority achieving a quorum is negligible, while the probability of the honest majority forming one is overwhelming. For a probability of an adversarial quorum bounded by 2^{-128}, we use the lemma to obtain initial values of k, m and reduce them until they are tight, whilst maintaining $k = m \cdot \phi(\frac{1}{2})$. However, we are able to perform additional fine tuning: in many applications a forgery may be catastrophic whilst an isolated failure to sign may be recoverable (via retries, or redundancy in the application using the primitive). Towards this, we may vary the relation between k and m. Intuitively, $k = m \cdot \phi(\frac{1}{2})$ implies that a group holding $\frac{1}{2}$ fraction of the stake has a significant probability of signing with that probability quickly rising (or respectivelly falling) if the amount of stake held is more (or less) than $\frac{1}{2}$. By setting $k = m \cdot \phi(\frac{1}{2+\beta})$ for a positive safety margin $\beta \leq \frac{1}{2}$, we are able to sacrifice liveness in favor of smaller parameters. Following

	Adversarial Stake							
	40%				33%			
$\frac{k}{m}$	k	m	L-Abs	L-Par	k	m	L-Abs	L-Par
$\phi(.50)$	3684	34891	99.99 %	≈ 1	1129	10690	$1-2^{-30}$	≈ 1
$\phi(.55)$	1865	16144	99.99 %	≈ 1	769	6654	$1-2^{-30}$	≈ 1
$\phi(.60)$	1182	1182	48.67 %	≈ 1	576	4593	99.59 %	≈ 1
$\phi(.67)$	755	5434	LL	≈ 1	414	3050	47.75 %	≈ 1
$\phi(.75)$	526	3411	LL	$1-3\cdot10^{-12}$	326	2113	1.98%	$1-3\cdot10^{-8}$
$\phi(.80)$	441	2695	LL	$1-7\cdot10^{-7}$	286	1747	LL	99.99%

Table 3. Required values of k, m so that an adversarial quorum is formed with probability at most 2^{-128}. L-Abs and L-Par represent the probabilities to form a quorum (before any retries) when the adversarial stake abstains or participates respectively. LL describes probabilities $< 1\%$. The parameters can be meaningfully used in conjunction with an incentive scheme or as an auxiliary opportunistic parametrization where a less aggressive parametrization is used as a fallback. Values of ≈ 1 indicate a chance of failure $< 10^{-30}$.

the above intuition, when we set $\beta > \alpha$, the honest majority cannot reliably sign without the adversary being involved: the honestly held stake is $(\frac{1}{2} + \alpha)$ whereas the baseline is $(\frac{1}{2} + \beta)$ which is higher.

In Table 3, we calculate values for combinations of adversarial stake and quorum percentage $\frac{k}{m}$ and fixed $\phi(1) = \frac{1}{5}$. Reducing $\phi(1)$ would allow one to very slightly decrease k for a significant increase in m.

From the table, liveness can be challenging in certain parametrizations. This can be addressed in a number of ways: First, the probability of an honest quorum can be boosted by allowing retries (e.g. by attaching a short counter to the message). Second, if an incentive structure is in place, rational adversaries who cannot directly subvert the protocol will choose to participate in signing honest messages. Ideally, we would like to be able to use the more compact parameters until such time as liveness is at risk, and then fall back to higher (k, m) values. In the following paragraph, we describe a way to achieve this :

Concurrent Hybrids Our protocol is amenable to running parametrizations of multiple (k, m) concurrently with minimal impact to the adversary's chance of success. All other protocol parameters and data are shared.Thus, individual signatures are produced according to the maximal pair of (k, m) values, and aggregation chooses the smallest ones that form a quorum. This will increase communication costs by transmitting potentially unneeded individual signatures, but will choose the smallest feasible quorum size.

For ease of presentation, we present our findings for $\phi(1) = \frac{1}{5}$. Decreasing this value slightly reduces k while increasing m.

6.2 Signature and Proof Efficiency

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

Individual signatures For producing an individual signature, 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 a 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 concreteness, we assume that the 3 bit strings can be packed in 176 bits: path needs $\log k$ bits, index needs $\log m$ and stake can be limited to 128 bit precision.When assuming verifiers have the contents of AVK in memory, signatures can be reduced to 2 elements for $\sigma, \sigma_{\mathsf{body}}$ plus $\log N$ bits for i and $\log m$ for index, as ev can be computed from $\sigma, \mathsf{index}, \mathsf{topic}$.

The costs of the verifier are $\log N$ evaluations of H_p, a pairing check 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.

PS^C**aggregate signatures.** In Concatenation based aggregate signatures we batch the signatures on topic and aggregate the signatures on body. This replaces $2k$ pairing checks with k short exponentiations in $\mathbb{G}_1$ and $\mathbb{G}_2$, k multiplications in $\mathbb{G}_1$ and $\mathbb{G}_2$ and 2 pairing checks combinable to 3 pairings.

It is also possible to compress proofs. For $k = 424$ leaves, revealing the entirety of the 8th level of the tree can be accomplished by publishing 256 hashes. In turn, this reduces the length for each individual inclusion proof by 7 "steps": rather than giving a path to the root, inclusion proofs can terminate 7 levels early. This brings down the cost to the equivalent of $3k$ G_1 elements and $k(\log N - 6.38)$ hashes. Furthermore, as we don't need an embedded curve setting, we can opt for a 384 bit curve following [4], and use a symmetric 256 bit hash functions for the Merkle tree and the mapping.This produces a proof size of ca. 374 KiB.

6.3 Efficiency of 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 Hash evaluations for $e_{\boldsymbol{\sigma}}$.
- k short exponentiations in $\mathbb{G}_2$ to produce ivk.
- k short exponentiations in $\mathbb{G}_1$ to produce μ.
- $2k$ range checks with bound m (for index bounds, and index uniqueness).
- k Comparisons between ev and $\phi(\mathsf{stake})$.

Proof	Asymptotic	k=424	k=576	k=769
Single PS^B	$2G_1 + G_2 + H \log N + S + M$	1.9 KiB	1.9 KiB	1.9 KiB
Single PS^B (FN)	$2G_1 + M$	118 B	118 B	118 B
Single PS^C	$2G_1 + G_2 + H \log N + S + M$	1.1 KiB	1.1 KiB	1.1 KiB
Single PS^C (FN)	$2G_1 + M$	102 B	102 B	102 B
Aggregate PS^B	$G_1 + 2G_2 + O(\log(k \log q)) \cdot G_H$	4.1 KiB	4.6 KiB	4.6 KiB
Aggregate PS^C	$k(G_1 + G_2 + S + M) + G_1 + k(\log N - \log k + 1) \cdot H$	366 KiB	480 KiB	641 KiB
Aggregate $\mathsf{PS}^C(FN)$	$k(G_1 + M) + G_1$	22 KiB	30 KiB	41 KiB

Table 4. Proof sizes for the PS^B and PS^C proof systems. We represent $\mathbb{G}_i$ elements by G_i, hash outputs by H, stake by S and path & index metadata by M. Concrete values are based on the parameters on the text: $\log N = 30$, 446 bit base elements and hashes for the PS^B setting, 384 and 256 bits for elements and hashes in the PS^C setting, 128 bit stake and 48 bit metadata. Single PS^B are over an arity-8 tree. The k values were derived from Table 3. The indication (FN) is the setting where the verifier is a full node and hence certain metadata can be eliminated from the signature.

- 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 $\mathbb{G}_H$ for which our proof system is more efficient. The main outlier is 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 $O\left(\frac{k \log^4 q}{\log (k \log q)}\right)$ as verification is dominated by a multiexponentiation based on the circuit size.

Mapping Efficiency Verifying the elligator-based mapping M^E is somewhat involved. We point out that 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 verified. Fortunately, in the analysis of Section 7, the pre-image set has a size[6] 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. Given that, the cost of verifying a M^E evaluation is dominated by the range checks enforcing the correct ordering of pre-images.

Estimate for constraints We now give an estimation on the number of constraints required for our scheme, with a k value of 414, $m = 2980$, and $\log p = \log q = 446$ and $N = 2^{30}$. We assume $\mathbb{G}_1$ operations to require 12

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

Proof	Operations
Single PS^B	$\log_8 N H_p + 2H_s + 400F + M_1 + M_T + 3P$
Single PS^C	$(\log N + 1)H_s + M_1 + M_T + 3P$
Aggregate PS^B	$O(k \log q)E_H + M_1 + M_T + 3P$
Aggregate PS^C	$k \cdot (M_2 + E_1 + E_2) + M_1 + M_T + 3P$

Table 5. Verification complexity for the dominant operations and terms. M_x represent $\mathbb{G}_x$ multiplications, F field operations, P represent pairings and E_x represent $\mathbb{G}_x$ multi exponentiations. H_s and H_p represent symmetric and Poseidon hashes respectively.

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: the root hash is checked $k = 414$ 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 amortized cost per lookup is ca 3009 constraints. H_p evaluations for the leave contents can be performed at $4 : 1$ compression at a cost of 300 constraints. Comparisons between values cost 2^b, 3b constraints, amortized to 2b when values are used twice. Mapping representations involve 60 constraints plus 3 range checks.

In total we have:

- $k \cdot (3009 + 300)$ constraints for Merkle Tree lookups.
- $k \cdot 100$ constraints for $e_{\boldsymbol{\sigma}}$.
- $(k + 1) \cdot 48 \cdot 100$ constraints for multiplications in $\mathbb{G}_2$ for ivk.
- $(k + 1) \cdot 12 \cdot 100$ constraints for multiplications in $\mathbb{G}_1$ for μ.
- $4k \cdot \log m$ for range checks and comparisons with bound m.
- $3k \log q + 60k$ for representation function evaluations.

In total, we obtain $3k \log q + 4k \log m + 9329k \approx 2^{22}$ constraints. Extrapolating from [15, 37, 38], for k=414 this gives us a proof size of under 4KB with a batched verification time of ca. 100sec. 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 [14] 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.

6.4 Further PS Options

Our protocol is modular with regards to the proof system. The complexities in using a different system are using a dense mapping that does not pass user data through a random oracle (such as the Elligator-based one in Section 7) and any potential trust requirements on the reference string. There exists a number of alternative circuit-based proof systems. SNARKs, such as Plonk [30] and Sonic [48] offer constant verifier 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 significant optimizations may be possible, e.g with custom Plonk gates for Poseidon. STARKs, such as Redshift [40] and Aurora [7] offer similar verifier performance at the cost of large proofs. As zero knowledge is not a requirement, the size required makes them less attractive. Finally, recursive proof systems such as Halo [14] or Plonky2 [52] can also be explored: constant depth recursion can reduce proof sizes and verifier load. 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.

6.5 Dynamic Adversaries and Forward Security

We have modeled our functionality and scheme in a static corruption model. In proof of Stake applications, dynamic corruption greatly enhances the power of the adversary: the adversary waits to see which users are eligible for an action (e.g. to produce the next block) and then corrupts them. In our functionality, it is possible to make a tradeoff between security against dynamic adversaries and grinding: mandating the body field of each message to be empty (and the entire message be used as the topic) implies that eligibility is predicated on the message, and is independently distributed across different messages. That is, user P_1 being eligible for message mesg_1 is independent of user P_1 being eligible for message mesg_2. This defeats the strategy of corrupting a user after they have performed a particular action.

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 lottery with index index_0, then the adversary *can* determine that user's evaluation for every other index, but it is reasonable to assume that in most applications users will elect to either sign over all 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 [13, 10, 27, 54] 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, dynamic 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 second item of discussion is that the stake distribution may naturally drift over time. While this depends on the application, a natural solution would be to have users periodically register fresh keys to produce a new aggregate key in a regular basis. This further reduces the relative power of adaptivity.

Forward Security 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 (potentially over an old stake distribution). 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, generic constructions [47] can be used to add forward security while also maintaining the uniqueness property for a fixed point in time.

7 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 [38] 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 Ouroboros Crypsinous MUPRF [41] uses a similar technique, but the additional requirements on group structure do not provide us with curves compatible with the original Elligator [9] construction. Elligator squared [58] 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 [59] 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 [58]. We modify it so that it always terminates after a single iteration 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 [59]. 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 [38].

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 [58]. A key observation from the investigation of [58, 59] 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 [58], 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 12 (Lemma 5,[58]). *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 12, 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 3 ([58]). *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 3.

Lemma 13. *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 $Pr[M(y) \neq \perp] < \frac{1}{5}$.*

8 Applications

In this section we delve into 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 `mesg` of interest. Before we proceed, we remark that some care needs to be applied to ensure the integrity of STM sampling based on our security model, namely that user public-keys are fixed prior to messages being proposed for signing. Even though grinding attacks have a negligible probability to produce a forgery, cf. Lemma 8, an attacker who knows `topic` prior to the keys being finalized, can attempt to grind the probability of signing `topic` 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 a blockchain setting, this attack can be averted by storing the public-keys on chain and then including an unpredictable fresh nonce drawn from the blockchain itself as part of the message while also verifying this nonce during verification. 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, we assume this is implemented by default.

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.

In Bitcoin, balances are sent to a ScriptPubKey and are spendable 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 the ECDSA key controlling the balance; spending requires opening both keys & a 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 checked that `<RedeemScript>` verifies correctly with regard 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 mesg will be announced together with a threshold. Interested bitcoin owners reveal their `<RedeemScript>` values and issue an individual signature on mesg. The entirety of the above process can happen off-chain as a layer 2 type of coordination. When a sufficient number of those individual signatures are collected on mesg, they can be aggregated to issue an aggregate signature on behalf of 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 [51], with the challenge that in a PoS blockchain, e.g., [43], 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, we first 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 into periods; the length of each period is sufficient to allow ledger settlement. Let SD_i be a settled stakeholder distribution (i.e. all honest parties agree on it) during period i. SD_0 is the stakeholder distribution embedded at genesis; we assume parties are in agreement regarding SD_0.

When the distribution SD_i is derived from the blockchain, the message $\mathsf{mesg}_i = (i, C_i)$ 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 mesg_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, the resulting signature, denoted by chp_i can be computed and disseminated. The triple (i, C_i, chp_i) is 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.

Proofs of data availability. High performance consensus protocol design in the permissioned setting (e.g., [28, 22, 57]) heavily exploits a decoupling between the data that are to be agreed on and the consensus protocol itself which is running on short references to that data. Such decoupling naturally carries a potential risk: running consensus on references for which the corresponding data does not exist. It follows that proving such optimizations secure may require resolving this *data availability* consideration. Even though this problem is easy to tackle in the permissioned setting (as e.g., we may require a sufficient number of signatures so we ensure at least one honest party has seen the data) it is much more challenging to solve in the permissionless setting [56]. Mithril provides an immediate solution: if the underlying consensus protocol is run on references for which a Mithril signature exists, data availability is (cryptographically) guaranteed.

References

1. Agrawal, S., Neu, J., Tas, E.N., Zindros, D.: Proofs of proof-of-stake with sublinear complexity (2022), https://arxiv.org/abs/2209.08673
2. Attema, T., Fehr, S., Klooß, M.: Fiat-shamir transformation of multi-round interactive proofs. Cryptology ePrint Archive (2021)
3. Baldimtsi, F., Madathil, V., Scafuro, A., Zhou, L.: Anonymous lottery in the proof-of-stake setting. In: CSF (2020)
4. Barbulescu, R., Duquesne, S.: Updating key size estimations for pairings. Journal of Cryptology **32**(4) (2019)
5. Bellare, M., Garay, J.A., Rabin, T.: Fast batch verification for modular exponentiation and digital signatures. In: EUROCRYPT (1998)
6. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: CCS (1993)
7. Ben-Sasson, E., Chiesa, A., Riabzev, M., Spooner, N., Virza, M., Ward, N.P.: Aurora: Transparent succinct arguments for r1cs. In: Theory and applications of cryptographic techniques (2019)
8. Benhamouda, F., Gentry, C., Gorbunov, S., Halevi, S., Krawczyk, H., Lin, C., Rabin, T., Reyzin, L.: Can a public blockchain keep a secret? In: TCC (2020)
9. Bernstein, D.J., Hamburg, M., Krasnova, A., Lange, T.: Elligator: elliptic-curve points indistinguishable from uniform random strings. In: CCS (2013)
10. 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) (2006)
11. Boneh, D., Drijvers, M., Neven, G.: Compact multi-signatures for smaller blockchains. In: ASIACRYPT (2018)
12. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: ASIACRYPT (2001)
13. Boneh, D., Shparlinski, I.E.: On the unpredictability of bits of the elliptic curve diffie-hellman scheme. In: CRYPTO (2001)
14. 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)
15. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: Security and Privacy (2018)
16. Bünz, B., Kiffer, L., Luu, L., Zamani, M.: Flyclient: Super-light clients for cryptocurrencies. In: Security and Privacy (2020)
17. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: TCC. Springer (2007)
18. Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N., Peled, U.: UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In: CCS (2020)
19. Chaidos, P., Kiayias, A., Reyzin, L., Zinovyev, A.: Approximate lower bound arguments. Cryptology ePrint Archive (2023), https://eprint.iacr.org/2023/1655

20. Chen, J., Micali, S.: Algorand: A secure and efficient distributed ledger. Theor. Comput. Sci. **777** (2019)
21. Choudhuri, A.R., Goel, A., Green, M., Jain, A., Kaptchuk, G.: Fluid MPC: secure multiparty computation with dynamic participants. IACR Cryptol. ePrint Arch. **2020** (2020), https://eprint.iacr.org/2020/754
22. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and tusk: a dag-based mempool and efficient BFT consensus. In: EuroSys (2022)
23. Das, S., Camacho, P., Xiang, Z., Nieto, J., Bunz, B., Ren, L.: Threshold signatures from inner product argument: Succinct, weighted, and multi-threshold. Cryptology ePrint Archive, Paper 2023/598 (2023), https://eprint.iacr.org/2023/598
24. David, B., Gazi, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: EUROCRYPT (2018)
25. Desmedt, Y., Frankel, Y.: Shared generation of authenticators and signatures (extended abstract). In: CRYPTO (1991)
26. Dodis, Y., Yampolskiy, A.: A verifiable random function with short proofs and keys. In: Public Key Cryptography (2005)
27. Fazio, N., Gennaro, R., Perera, I.M., Skeith, W.E.: Hard-core predicates for a diffie-hellman problem over finite fields. In: CRYPTO 2013. Springer (2013)
28. Fitzi, M., Hirt, M.: Optimally efficient multi-valued byzantine agreement. In: PODC (2006)
29. 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)
30. 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
31. Ganesh, C., Orlandi, C., Tschudi, D.: Proof-of-stake protocols for privacy-aware blockchains. In: EUROCRYPT (2019)
32. Garg, S., Jain, A., Mukherjee, P., Sinha, R., Wang, M., Zhang, Y.: hints: Threshold signatures with silent setup. Cryptology ePrint Archive, Paper 2023/567 (2023), https://eprint.iacr.org/2023/567
33. Gaži, P., Kiayias, A., Zindros, D.: Proof-of-stake sidechains. In: Security and Privacy (2019)
34. Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Secure distributed key generation for discrete-log based cryptosystems. J. Cryptol. **20**(1) (2007)
35. Ghoshal, A., Tessaro, S.: Tight state-restoration soundness in the algebraic group model. In: CRYPTO. Springer International Publishing, Cham (2021)
36. Goldwasser, S., Ostrovsky, R.: Invariant signatures and non-interactive zero-knowledge proofs are equivalent. In: CRYPTO (1992)
37. Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: A new hash function for zero-knowledge proof systems. In: USENIX Security (2021)
38. Hopwood, D.: Pluto/eris half-pairing cycle of elliptic curves, https://github.com/daira/pluto-eris
39. Itakura, K., Nakamura, N.: A public-key cryptosystem suitable for digital multisignatures. NEC Research and Development **71** (October 1983)
40. Kattis, A., Panarin, K., Vlasov, A.: Redshift: Transparent snarks from list polynomial commitment iops. IACR Cryptol. ePrint Arch. **2019** (2019)
41. Kerber, T., Kiayias, A., Kohlweiss, M., Zikas, V.: Ouroboros crypsinous: Privacy-preserving proof-of-stake. In: Security and Privacy (2019)
42. Kiayias, A., Miller, A., Zindros, D.: Non-interactive proofs of proof-of-work. In: Financial Cryptography. Springer (2020)

43. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: CRYPTO (2017)
44. Leung, D., Suhl, A., Gilad, Y., Zeldovich, N.: Vault: Fast bootstrapping for the algorand cryptocurrency. In: NDSS (2019)
45. Li, C., Hwang, T., Lee, N.: Threshold-multisignature schemes where suspected forgery implies traceability of adversarial shareholders. In: EUROCRYPT (1994)
46. Lysyanskaya, A.: Unique signatures and verifiable random functions from the dh-ddh separation. In: CRYPTO (2002)
47. Malkin, T., Micciancio, D., Miner, S.: Efficient generic forward-secure signatures with an unbounded number of time periods. In: International Conference on the Theory and Applications of Cryptographic Techniques. Springer (2002)
48. Maller, M., Bowe, S., Kohlweiss, M., Meiklejohn, S.: Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In: CCS (2019)
49. Micali, S., Ohta, K., Reyzin, L.: Accountable-subgroup multisignatures: extended abstract. In: CCS (2001)
50. Micali, S., Reyzin, L., Vlachos, G., Wahby, R.S., Zeldovich, N.: Compact certificates of collective knowledge. In: Security and Privacy (2021)
51. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2008)
52. Polygon Zero Team: Plonky2: Fast recursive arguments with plonk and fri. https://github.com/mir-protocol/plonky2
53. Ristenpart, T., Yilek, S.: The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In: EUROCRYPT (2007)
54. Shani, B.: On the bit security of elliptic curve diffie–hellman. In: IACR International Workshop on Public Key Cryptography. Springer (2017)
55. Shoup, V.: Practical threshold signatures. In: EUROCRYPT (2000)
56. Smith, C., Beckett, A., Wackerow, P., AlehN: Data availability. https://ethereum.org/en/developers/docs/data-availability/
57. Spiegelman, N.G., Giridharan, N., Sonnino, A., Kokoris-Kogias, L.: Bullshark: DAG BFT protocols made practical. CCS (2022)
58. Tibouchi, M.: Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In: Financial Cryptography (2014)
59. 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 (2019)