Distributed Randomness using Weighted VRFs

Sourav Das[1], Benny Pinkas[2,3], Alin Tomescu[2], Zhuolun Xiang[2]

[1]University of Illinois at Urbana-Champaign
[2]Aptos Labs
[3]Bar-Ilan University

Abstract. Generating and integrating shared randomness into a blockchain can expand applications and strengthen security. We aim to have validators generating blockchain randomness autonomously, and fresh shared randomness is generated for each block. We focus on proof-of-stake blockchains, where each validator has a different amount of stake (aka weight). Such chains introduce a weighted threshold setting where subset authorization relies on the cumulative weight of validators rather than the subset size.

We introduce three cryptographic protocols to enable generating shared randomness in a weighted setting: A publicly verifiable secret sharing scheme (PVSS) which is weighted and aggregatable, a weighted distributed key generation protocol (DKG), and a weighted verifiable unpredictable function (VUF). Importantly, in the VUF protocol, which is the protocol that is run most frequently, the computation and communication costs of participants are independent of their weight. This feature is crucial for scalability.

We implemented our schemes on top of Aptos blockchain, which is a proof-of-stake blockchain deployed in production. Our micro-benchmarks demonstrate that the signing and verification time, as well as the signature size, are independent of the total weight of the parties, whereas the signing time and signature size of the baseline (BLS with virtualization) increase significantly. For instance, our VUF reduces the signature size by factors of $7\times$ and $34\times$ for total weights of 821 and 4053, respectively. We also demonstrate the practicability of our design via an end-to-end evaluation.

1 Introduction

A major limitation of existing blockchains is that they primarily support deterministic computation [Nak08,W+14]. This is a fundamental property as the security of blockchains crucially relies on the replicability of the blockchain state. However, many natural applications are inherently randomized. Few such examples are games, such as lotteries, card and action games, randomized Non-Fungible Token (NFT) minting and attribution, airdrops, and so on. Shared unpredictable randomness has usage cases beyond enabling applications that require randomized computation. It can also further strengthen the security of blockchain protocols. For example, having access to publicly verifiable random values allows leader-based blockchain networks to select leaders in a fair randomized fashion. Furthermore, integrating randomness into transaction ordering reduces opportunities for maximal extractable value (MEV) attacks by minimizing the advantages of targeting particular transaction positions within a block [DGK+20].

To support these applications, blockchains rely on shared unpredictable randomness, typically from a randomness beacon [CMB23,KWJ23,RG22]. Intuitively, the randomness beacon outputs unpredictable random values at periodic intervals on which the validators/miners of the blockchain agree. Whenever an application calls for randomness, it uses the agreed-upon output from the randomness beacon.

This paper describes methods for creating verifiable randomness by the validators which manage the blockchain. The process occurs in a weighted setting where each validator has a weight that is proportional to its stake in a proof-of-stake (PoS) blockchain. Randomness can only be generated by subsets of validators with a combined weight greater than a specific threshold. Furthermore, the output is verifiable, in the sense that everyone can verify its correctness and therefore no validator can manipulate the output. The performance of computing the random output is crucial, since new random values must be computed very frequently.

Limitations of using an external randomness beacon. One popular approach for supporting randomized computation in blockchains is to rely on an external randomness beacon such as Drand [Dra23] and Chainlink [cha24]. However, this approach has several security and efficiency limitations. It adds fault and trust assumptions, tying the blockchain's trust to the beacon's protocol. For instance, Drand's security currently depends on preventing corruption of 9 of its 18 members. Additionally, the blockchain's operation hinges on the beacon's continuous service. Any interruption, intentional or due to attacks or bugs, impacts the blockchain protocol using it.

Another significant issue with external randomness beacons is their latency. Beacons like Drand emit new random values every 30 seconds, causing a delay for protocols and transactions that need fresh randomness. This latency also compromises security, as random information only updates at the beacon interval end. This affects, for example, leader election and protection against MEV attacks, as the leader's identity and transaction order remain fixed and public until the next beacon output.

Lastly, using an external beacon complicates blockchain and application development. It forces each randomness call to navigate a two-transaction process, first commit to a future, undisclosed beacon output, and then consume the committed beacon. This introduces usability challenges, and also requires an additional mechanism to select the future output, with trade-offs: choosing an imminent output risks premature revelation and predictability, while a distant output can cause excessive latency.

On-chain randomness to the rescue. An alternative approach to support randomized applications on blockchains is to rely on shared randomness generated by the set of validators of that blockchain. The validators should generate fresh shared randomness for each block, ideally right after agreeing on the block. The random output must be unaffected by which authorized subset of validators computed it, and it must also be verifiable, meaning its validity can be confirmed by everyone, thereby preventing any corrupt validators from altering the output value. We refer to shared randomness with all these properties as the *on-chain* randomness. ("on-chain" also means that the output is generated in real time for each block published on the chain.)

On-chain randomness immediately addresses the above-mentioned issues of external randomness sources: Since the blockchain validators themselves generate the shared randomness, the fault-tolerant assumption and the availability guarantees of the blockchain remain intact. Moreover, the fact that a new shared randomness is output for each block, immediately addresses the latency and synchronization issues. Also, since the shared randomness is revealed only after the block is ordered, its value remains unpredictable until the block has been ordered. Lastly, blockchain or application developers can effortlessly utilize such randomness. This is achieved by simply retrieving the randomness seed embedded in the block, and, if necessary, subsequently generating additional randomness using a pseudorandom function.

1.1 Challenges

We study building blocks for constructing an on-chain randomness beacon protocol for BFT-based proof-of-stake blockchains. Briefly, in these blockchains, the validators are *weighted* where the weight of a validator is proportional to the amount of stake it contributes to secure the consensus protocol. Typically, these systems assume that malicious validators control at most $1/3$-rd of the stake. Many deployed blockchains such as Ethereum2.0 [BG17], Algorand [GHM$^+$17], Avalanche [Roc18] and Aptos [Apt22] are based on this threat model.

A natural approach for designing an on-chain randomness service is based on a threshold verifiable random function (VRF) [MRV99,Dod02]. The keys for computing the VRF are shared among the validators and a threshold number of the validators is required in order to compute it. The function is *verifiable* in the sense that anyone can verify that the output of the VRF is correct. Therefore, if the randomness assigned to a block is defined, for example, as the output of the VRF when the input is the block number, then no corrupt validator can change the output assigned to this block.

In a threshold VRF based protocol, validators first setup the VRF keys by running an distributed key generation (DKG) [GJKR07] and then collaboratively compute the VRF. The common approach is first to compute a verifiable *unpredicatable* function (VUF), for example by signing the same message, such as the block number. The VUF output has the property that it is unpredictable and it verifiable, but is not necessarily pseudo-random. The second step is to apply a hash function to the VUF output. If we model the hash function as a random oracle guarantees that the final output is random.

Implementing the above-mentioned approach directly for proof-of-stake (PoS) blockchains is inefficient because of the *weighted* nature of these blockchains. More precisely, in PoS blockchains the authorized subset of the validators does not depend on the number of validators in the subset but rather on their combined weight. To naively use an unweighted threshold VUF, the number of shares contributed by each validator to computing the VUF output must be proportional to its weight. Moreover, the DKG protocol must generate a number of shares that is proportional to the total stake, and give each validator a number of shares proportional to its weight. In existing blockchains, the total stake is large, on the order of billions, whereas existing VUF and DKG schemes scale only to a few hundred shares [GJKR07,DYX$^+$22]. Furthermore, there are large differences between the stake of different validators.[1] Assigning shares for rounded amount of stakes, such as giving the validator with the least stake a single share and then distributing proportionate shares to other validators, leads to rounding errors that require analysis, and still yields a high number of shares [GKR23,dST23]. (We discuss in Appendix B how to analyze and bound the error that is caused by rounding. Increasing the number of shares reduces rounding errors, but directly affects performance when using known threshold constructions.) This implies that additional techniques are required to support on-chain randomness in PoS blockchains. Looking ahead, we will use a weighted DKG and weighted threshold VUF to achieve on-chain randomness for PoS blockchains.

Other constraints: stake/weight changes. In addition to the scalability issues arising from the large number of shares, practical PoS blockchains introduce additional constraints. Existing PoS blockchains support changes to the stakes of validators. More precisely, these systems operate at periodic intervals known as *epochs*. The set of validators and their stake distribution remain fixed within each epoch, and can change across epoch boundaries. Moreover, the validators of the next epoch do not come online until the new epoch starts. This constraint motivates designing an on-chain randomness beacon scheme, where the validators of the outgoing epoch run a *non-interactive* weighted DKG protocol to share keys that will be used by the incoming validators. An alternate design is to let the incoming validators run the weighted DKG at the start of each epoch. However, such an approach is undesirable, as the blockchain stalls until the DKG is finished in the new epoch.

Looking ahead, we will indeed adopt an approach where validators of the outgoing epochs use a non-interactive weighted DKG, based on a aggregatable publicly-verifiable secret sharing (PVSS) scheme, to generate threshold VUF keys for the incoming validators.

More challenges in designing an on-chain randomness. Existing DKG protocols have the following typical structure. Participants use a verifiable secret sharing scheme to share a random secret with others, and the sum of all these secrets is used as a key for the VUF. We also adopt this approach, but we require two additional properties of the underlying secret sharing scheme: First, we require the secret sharing scheme to be *publicly verifiable* in the sense that the dealer can post a transcript of sharing all shares, and the correctness of this transcript can be verified by anyone without any additional interaction. (Other approaches that require interaction, such as participants complaining about receiving incorrect shares, are much harder to implement due to the need to achieve agreement among all participants given potential malicious failures of the network or participants.) Second, the secret sharing scheme must be *aggregatable* in the sense that the transcripts of different dealers can be aggregated into a single transcript that shares the combination of all secrets and is of about the same length as an original transcript. Looking ahead, we use the aggregation property to design a DKG protocol where participants agree on a *single valid aggregated transcript*, thereby eliminating the need to reach a consensus on the values and order of the transcripts of all dealers. The fact the agreement is only required for a single transcript rather than on n transcripts (for n dealers) is crucial for performance.

We must emphasize that current known *aggregatable* PVSS schemes, e.g. [GJM$^+$21a], can only share group elements, rather than field elements (e.g., when working in $\mathbb{Z}_p^*$ they can share elements of the form g^a, where $a \in [0, p)$, and not an element $a \in \mathbb{Z}_p^*$). This makes it much harder to use the key shared in these schemes as a key for a VUF, and required designing the VUF accordingly.

1.2 Our Contributions

We focus on the design of the following primitives:

[1] For example, as of January 22, 2024, the Aptos blockchain network had 126 validators. The validator with the highest stake had about 20X more stake than the validator with minimal stake. 60% of the validators had more than 2X or more the stake of the smallest validator (`https://explorer.aptoslabs.com/validators?network=mainnet`). The Solana network had more than 2000 validators. The validator with the highest stake had 1000X more stake than each of the 450 validator with lowest stake. (`https://solanabeach.io/validators`).

– A weighted non-interactive aggregatable publicly-verifiable secret sharing, which we denote as a weighted PVSS. This scheme shares group elements. (The non-interactivity, aggregation, and public-verfiability properties are crucial for a fast and non-interactive key generation procedure.)
– A weighted distributed key generation protocol – weighted DKG, which uses our weighted PVSS scheme
– A weighted verifiable unpredictable function – weighted VUF, which uses the keys that are shared in the weighted DKG. These keys are group elements. Each participant in the VUF computation sends a single message, independently of its weight. The output of the weighted VUF can be used for computing a weighted verifiable random function, by applying to it a hash function modeled as a random oracle.

The main advantage of our new weighted VUF protocol is that a validator that submits a share for computing an output of the VUF has communication and computation overheads that are *independent of the weight of the validator*. This is crucial for scalability and performance. In particular, the total communication is linear in the number of validators that contribute shares for the computation, and is independent of the threshold weight required for computing the function. The cost of the aggregator that computes the final output from the shares is linear in the threshold weight, but the number of pairings that it has to compute is only equal to the number of validators that submitted shares. We implement our schemes on top of Aptos blockchain [Apt22], a proof-of-stake blockchain deployed in production. From the micro-benchmarks, for instance, our scheme reduces the signature size by a factor $7\times$ and $34\times$ for total weight of 821 and 4053, respectively. We also demonstrate the practicability of our design via an end-to-end evaluation.

It is important to note that, while there are lower bounds and impossibility results for weighted secret sharing (see, e.g., [BTW05]), they do not apply to computing a VUF. In fact, the setup cost of the VUF for each validator depends on its weight. However, this setup is only run once, while the overhead of the validator in each invocation of the function is independent of the weight.

1.3 Related Work

We describe related work for constructing PVSS schemes, DKG protocols, and a weighted VUF. We finish with a discussion on recent developments in weighted threshold cryptography.

PVSS schemes for sharing group elements as secret keys. PVSS schemes have been studied for a long time, starting with [Sta96], and with origins in [CGMA85]. The performance of PVSS has been dramatically reduced to $O(n)$ total work with the SCRAPE scheme [CD17], and was further improved in constructions such as [CD20,CDGK23]. As we mentioned earlier, we need the PVSS to be aggregatable. Current PVSS schemes with this property can only share a secret key which is a group element: There is recent work on aggergatable PVSS [GJM$^+$21a], and aggergatable PVSS with adaptive security [BLL$^+$23,BL23], but these protocols do not easily transfer to the weighted setting.

The schemes closest to our work are the SCRAPE [CD17] construction and Groth's construction [Gro21]. More precisely, our PVSS scheme can be viewed as a combination of these two schemes. We note that the PVSS scheme in [Gro21] focuses on sharing field element secrets. Looking ahead, we tailor the PVSS scheme in [Gro21] to share group elements. For this reason, there is no longer a need to use the range proof needed in the PVSS scheme of [Gro21]. To best of our knowledge, no prior work has focused on designing concretely efficient aggregatable PVSS schemes with weighted recipients.

Previous work on DKG. As we mentioned earlier, in this paper we focus on a non-interactive weighted DKG to generate a sharing of a random group element secret. Numerous works have studied interactive DKG protocols under various network assumption, and there are relatively fewer non-interactive DKG constructions, such as [FS01,Gro21,KMM$^+$23,CD23,KGS23]. Non-interactive DKG protocols adopt the framework of nodes publishing PVSS transcripts for sharing random secrets using a broadcast channel. The DKG key is then an aggregated secret that is shared by a set of qualified participants. None of these prior non-interactive DKG schemes focus on the weighted setting.

Comparison to previous work on randomness beacons. Numerous works have studied decentralized randomness beacons under various threat models, networking constraints, and cryptographic assumptions. We refer readers to recent surveys on this subject [CMB23,RG22,KWJ23]. The Drand system deploys a threshold VRF-based randomness beacon and is the closest scheme to ours [Dra23]. Both Drand and our scheme run a DKG to set up threshold VRF keys and later use them to compute VRF evaluations. However, there are some crucial differences: First, we consider a weighted setting, whereas Drand implements an unweighted threshold. Second, we consider a setting where rekeying needs to be done quite frequently, say every couple of hours, due to changes in the stake of different validators. This requires using a very efficient, non-interactive DKG, and we adopt DKG schemes that set up group elements as the secret keys. On the other hand, Drand uses less frequent rekeying and, therefore, can use interactive DKG schemes to set up the BLS threshold signature signing keys. Concretely, Drand uses Pedersen's DKG (with complaints) as their DKG.

Comparison to existing on-chain randomness. Some blockchains (e.g. DFINITY [HMW18]) assume non-PoS security model for randomness, as a result, they can rely on a DKG and a threshold VRF (tVRF), rather than a weighted VRF. This threshold setting is much easier than our weighted setting because the total number of shares can be set to the number of validators (e.g., hundreds). External beacons like Drand [Dra23] also rely on implementing a DKG and a tVRF among a committee of servers. As a result, they are easier to implement in the threshold setting. Unfortunately, when a transaction consumes the generated randomness, it must now place external trust in the randomness beacon's security and availabilty. Furthermore, the randomness cannot be consumed instantly, but via a commit-and-reveal process, which makes development more cumbersome and access to randomness more delayed.

A very promising approach for producing randomness is via verifiable delay functions (VDFs) [BBBF18]. This approach has the advantage that unpredictability holds even if all validators are corrupted, under the assumption that nobody can evaluate the VDF faster than the delay it was originally set up with. Unfortunately, this approach cannot produce randomness quickly since it is inherently based on delaying the computation of the randomness. Furthermore, if blocks are produced more often than VDFs, then those blocks will not have access to instant randomness.

Lastly, some randomness designs are biasable or predictable by even a single malicious validator. In Flow's design [HBY23], validators cast votes and attach their VRF evaluations for a proposed block A, allowing the proposer of the next block B to aggregate votes and reveal the randomness of A. This enables bias by strategically not proposing block B, rendering block A an orphan block. Celo [cel24] lets the proposer of a block pick the randomness, commit to it and reveal it in a subsequent block. This opens avenues for refusing to reveal the randomness or making informed predictions to maximize profit based on expected transactions in the subsequent block. In Ethereum [Eth], each block proposer evaluates a VRF over the current epoch number. Then, the epoch's randomness is defined as a best-effort combination of the proposers' VRF evaluations. Unfortunately, this approach is prone to bias, as one or more colluding block proposers can choose not to mix their contributions in, if they do not like the outcome. Furthermore, this approach is very slow, as epochs only happen every 6.4 minutes.

2 Preliminaries

2.1 Notations and System Model

We use κ to denote the security parameter. Throughout the paper, we will use "$\leftarrow$" for probabilistic assignment and ":=" for deterministic assignment. We use $|S|$ to denote the size of a set S. Let $\mathbb{F}$ be a finite field of order q. For any integer a, we use $[a]$ to denote the ordered set $\{1, 2, \ldots, a\}$. Also, for two integers a and b where $a < b$, we use $[a, b]$ to denote the ordered set $\{a, a+1, \ldots, b\}$. For two vectors $\boldsymbol{X}$ and $\boldsymbol{Y}$ of equal length $k \in \mathbb{N}$, we use $\boldsymbol{X} \cdot \boldsymbol{Y}$ to denote a vector whose elements are the element-wise product of $\boldsymbol{X}$ and $\boldsymbol{Y}$, i.e., if $\boldsymbol{X} = [x_1, \ldots, x_k]$ and $\boldsymbol{Y} = [y_1, \ldots, y_k]$, then $\boldsymbol{X} \cdot \boldsymbol{Y} := [x_1 \cdot y_1, \ldots, x_k \cdot y_k]$.

Threat model and network assumption. We consider a system of n participants, representing the validators of a blockchain, with participant i having weight w_i. Let $W := \sum_{i \in [n]} w_i$ be the total weight. We assume no-restrictions on how these weights are distributed among the participants. We consider a *static* adversary $\mathcal{A}$ that can corrupt participants with combined weight of up to w, where w is typically set to be $1/3$ of the total weight W. (This is not a restriction of our PVSS or VUF protocols but rather a threshold set by a consensus protocol.) We assume a public key infrastructure (PKI) for our PVSS scheme, and work in the *random oracle model*, which we need due to our use of the Fiat-Shamir [FS87] heuristic to achieve non-interactivity in our weighted PVSS. For our weighted PVSS and DKG, we will assume that participants have access to a broadcast channel. For on-chain randomness, we will use the underlying blockchain as the broadcast channel.

We work with prime-order groups $(\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T)$ with scalar field $\mathbb{F}$ and with a bilinear pairing operation: $e : \mathbb{G} \times \hat{\mathbb{G}} \to \mathbb{G}_T$.

Definition 1 (Bilinear Pairing). *Let $\mathbb{G}, \hat{\mathbb{G}}$ and $\mathbb{G}_T$ be three prime order cyclic groups with scalar field $\mathbb{F}$. Let $g \in \mathbb{G}$ and $\hat{g} \in \hat{\mathbb{G}}$ be the generators. A pairing is an efficiently computable function $e : \mathbb{G} \times \hat{\mathbb{G}} \to \mathbb{G}_T$ satisfying the following properties.*

1. *bilinear: For all $u, u' \in \mathbb{G}$ and $\hat{v}, \hat{v}' \in \hat{\mathbb{G}}$ we have*

$$e(u \cdot u', \hat{v}) = e(u, \hat{v}) \cdot e(u', \hat{v}), \textit{ and}$$
$$e(u, \hat{v} \cdot \hat{v}') = e(u, \hat{v}) \cdot e(u, \hat{v}')$$

2. *non-degenerate: $g_T := e(g, \hat{g})$ is a generator of $\mathbb{G}_T$.*

We refer to $\mathbb{G}$ and $\hat{\mathbb{G}}$ as the source groups and $\mathbb{G}_T$ as the target group.

We base the security of our protocols on the bilinear Diffie-Hellman (BDH) assumption in $(\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T)$, which we formally define in Definition 2.

Definition 2 (Bilinear Diffie-Hellman (BDH)). *On input g, g^a, g^b, g^c, $\hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c$, where g is a generator of $\mathbb{G}$, $\hat{g}$ is a generator of $\hat{\mathbb{G}}$, and $a, b, c \leftarrow\!\$\ \mathbb{F}$, the BDH assumption states that it is computationally hard to compute $e(g, g)^{abc}$.*

2.2 Performance Considerations

When using pairings of the format $e : \mathbb{G} \times \hat{\mathbb{G}} \to \mathbb{G}_T$, the common basic operations that can be computed are exponentiations in $\mathbb{G}, \hat{\mathbb{G}}$ and $\mathbb{G}_T$, and the pairing e itself. When computing the multiplication of many exponentiations, namely $\prod_{i=1}^{n} {g_i}^{r_i}$, it is possible to use a known optimization of computing a multi-exponentiation, which is faster by an asymptotic factor of about $\log n$ compared to computing n individual exponentiations.

As an example, we can give the performance of BLS12-381 pairings, on a 10-core 2021 Apple M1 Max, as described in [Tom22]:

- Computing *exponentiations* in $\mathbb{G}, \hat{\mathbb{G}}$ and $\mathbb{G}_T$ takes 72, 136, and 500 microseconds, respectively.
- A *multi-exponentiation* of 256 elements in $\mathbb{G}$ takes 760 microseconds, i.e., 3 microseconds per exponentiation, which is $24\times$ faster than computing individual exponentiations. Computing a size-256 multi-exponentiation in $\hat{\mathbb{G}}$ takes 1.88 milliseconds, i.e., 7.33 microseconds per exponentiation. which is $18.8\times$ faster.
- Computing a *pairing* takes 486 microseconds.

Given these benchmarks, our goal is to minimize the number of pairings that need to be computed. In particular, we aim to replace a computation of n pairings by computing a single pairing and a multi-exponentiation of n items in $\mathbb{G}$ or $\hat{\mathbb{G}}$. If n is large then this optimization can improve the run time of this specific computation by two orders of magnitude.[2]

3 Security Definitions

We consider adversaries that control an unauthorized subset of the participants. In a threshold scheme these are subsets with size at most the threshold t. In a weighted scheme these are subsets whose total weight is at most the weight threshold w. We only consider *static* adversaries, which choose the subset of participants that they control before the protocol begins.

[2] We also note that when computing the multiplication of multiple pairings, it is possible to run an more efficient computation of a multi-pairing which computes a single final exponentiation for all pairings together [Tom22]. The improvement in the runtime is considerable, by a factor of about two, but much smaller than the improvement gained by replacing pairings with a multi-exponentiation.

3.1 Threshold VUF

For better exposition, we first define the security of threshold VUF with an idealized key generation algorithm. The advantages of this approach, rather than immediately considering VUF keying using a DKG, is that it lets us focus on the VUF security and avoid any nuances due to the key generation phase. Our security definitions are inspired from the threshold signature definitions in [BS23]. We begin with definitions for the unweighted threshold setting.

Definition 3 (Threshold VUF). *An (n, t)-threshold VUF scheme for a finite message space $\mathcal{M}$ is a tuple of polynomial time computable algorithms $\Sigma = (\mathsf{KeyGen}, \mathsf{ShareSign}, \mathsf{ShareVerify}, \mathsf{Verify}, \mathsf{Aggregate})$ with the following properties:*

1. $\mathsf{KeyGen}(n,t) \to \mathsf{pk}, \{\mathsf{pk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in[n]}$. *The key generation algorithm takes as input the total number of signers n and the threshold t, and the public parameters.* KeyGen *then outputs a public key pk, a vector of threshold public keys $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$, and a vector of secret signing keys $\{\mathsf{sk}_1, \ldots, \mathsf{sk}_n\}$. The j-th signer receives $(\mathsf{pk}, \{\mathsf{pk}_i\}_{i\in[n]}, \mathsf{sk}_j)$.*
2. $\mathsf{ShareSign}(m, \mathsf{sk}_i) \to \sigma_i$. *The partial signing algorithm is a possibly randomized algorithm that takes as input a message m and a secret key share sk_i. It generates a signature share σ_i.*
3. $\mathsf{ShareVerify}(m, \sigma_i, \mathsf{pk}_i) \to 0/1$. *The signature share verification algorithm is a deterministic algorithm that takes as input a message m, a public key share pk_i, and a signature share σ_i. It outputs 1 (accept) or 0 (reject).*
4. $\mathsf{Aggregate}(S, \{(\sigma_i, i)\}_{i\in S}) \to \sigma/\bot$. *The signature share combining algorithm takes as input the public key pk, a vector of public key shares $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$, a message m, and a set S of $t+1$ signature shares (σ_i, i) (with corresponding indices). It outputs either a signature σ or $\bot$.*
5. $\mathsf{Verify}(m, \mathsf{pk}, \sigma)$. *The signature verification algorithm is a deterministic algorithm that takes as input a public key pk, a message m, and a signature σ. It outputs 1 (accept) or 0 (reject).*
6. $\mathsf{Derive}(m, \mathsf{pk}, \sigma) \to \rho$. *The output derivation algorithm is a deterministic algorithm that takes as input a public key pk, a message m, and a signature σ, and outputs the VUF output ρ.*

We require the threshold VUF to ensure *correctness*, *uniqueness*, and *unforgeability* (also known as *unpredictability*). Next, we define these properties.

Correctness means that the output of the VUF protocol agrees with its definition.

Definition 4 (Correctness). *A threshold VUF scheme is correct if for all security parameters $\lambda \in \mathbb{N}$, all allowable thresholds $1 \le t+1 \le n$, all S such that $t+1 \le |S| \le n$, all messages $m \in \mathcal{M}$, and for $\mathsf{pk}, \{\mathsf{pk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in[n]} \leftarrow \mathsf{KeyGen}(n,t)$, the following holds:*

- $\Pr[\mathsf{ShareVerify}(m, \mathsf{ShareSign}(m, \mathsf{sk}_i), \mathsf{pk}_i) = 1] = 1$
- $\Pr[\mathsf{Verify}(m, \mathsf{Aggregate}(\{(\sigma_i, i)\}_{i\in S}), \mathsf{pk}) = 1 :$
  $\sigma_i = \mathsf{ShareSign}(m, \mathsf{sk}_i)\ \forall i \in S] \;=\; 1 - \mathsf{negl}(\lambda)$

Here, the probability is over the choice of randomness of the KeyGen *and the* $\mathsf{ShareSign}$ *algorithm.*

Looking ahead, since we seek to use the threshold VUF scheme in designing a randomness beacon, we require the threshold VUF to satisfy the following uniqueness property.

Definition 5 (Uniqueness). *A threshold VUF scheme ensures uniqueness if for all security parameters $\lambda \in \mathbb{N}$, all allowable thresholds $1 \le t+1 \le n$, all messages $m \in \mathcal{M}$, and for $\mathsf{pk}, \{\mathsf{pk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in[n]} \leftarrow \mathsf{KeyGen}(n,t)$, the following holds:*

$$\Pr[\mathsf{Verify}(m, \sigma, \mathsf{pk}) = 1 \wedge \mathsf{Verify}(m, \sigma', \mathsf{pk}) = 1 \wedge \sigma \neq \sigma'] = 0$$

Note that the uniqueness property is unconditional, and hence should hold even if $\mathcal{A}$ is computationally unbounded.

Unforgeability means that an adversary cannot predict the output of the VUF for any input on which the VUF was not queried earlier. We next define the corresponding security game. Let $\mathsf{H}_{\hat{\mathbb{G}}} : \mathcal{M} \to \hat{\mathbb{G}}$ be a hash function (modeled as a random oracle). We use q_{H} to denote the maximum number of queries to H.

Game 1 (Unforgeability Under Chosen Message Attack) *For a threshold VUF scheme Σ we define the* $\mathsf{UF\text{-}CMA}^{\mathcal{A},t}$ *game in the presence of an adversary $\mathcal{A}$ as follows:*

– **Setup.** *$\mathcal{A}$ specifies (n, t) and a set $\mathcal{C} \subset [n]$, $|\mathcal{C}| \le t$ of corrupt signers. Let $\mathcal{H} = [n] \setminus \mathcal{C}$ be the set of honest signers. Let $\mathcal{S}$ be the set of signers that $\mathcal{A}$ queries for partial signatures on the forged input. We require that $|\mathcal{C} \cup \mathcal{S}| \le t$.*
– **Key Generation**. *Run $\mathsf{KeyGen}(n, t)$ to generate the keys. Each signer i learns its signing key sk_i, along with the public key pk, and public keys $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$. Also, $\mathcal{A}$ learns sk_i for each $i \in \mathcal{C}$.*
– **Online Phase.** *During this phase, $\mathcal{A}$ can make the following queries.*
  - **Partial VUF Queries.** *$\mathcal{A}$ submits the tuple (i, m) for some $i \in \mathcal{H}$, and receives $\sigma_i \leftarrow \mathsf{ShareSign}(\mathsf{sk}_i, m)$.*
  - **Random Oracle Queries.** *$\mathcal{A}$ submits a query m to the random oracle $\mathsf{H}_{\hat{\mathbb{G}}}$. The functionality checks if $\mathsf{H}_{\hat{\mathbb{G}}}(m) = \perp$, and if so it sets $\mathsf{H}_{\hat{\mathbb{G}}}(m) \leftarrow\$ \hat{\mathbb{G}}$. It then returns $\mathsf{H}_{\hat{\mathbb{G}}}(m)$.*
– **Output Determination.** *$\mathcal{A}$ outputs a message m^* and a VUF output σ^*. Output 1 if $\mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*) = 1$ and $|\mathcal{C} \cup \mathcal{S}| \le t$, where $\mathcal{S}$ is the subset of signers $\mathcal{A}$ queried for a partial VUF query on the forged input m^*. Otherwise, output 0.*

Definition 6 (Unforgeability). *We say that Σ is $(\varepsilon, \tau, q_{\mathsf{H}}, q_s)$-unforgeable under chosen message attacks (UF-CMA) if for all adversaries $\mathcal{A}$ running in time at most τ, making at most q_{H} random oracle queries, and making at most q_s signing queries,* $\Pr[\mathsf{UF\text{-}CMA}^{\mathcal{A},t} = 1] \le \varepsilon$.

In this work we do not define the exact values of the parameters but rather set ε to be negligible and set $\tau, q_{\mathsf{H}}, q_s$ to be polynomial in the security parameter.

The security definition for the *weighted* VUF is similar, except that authorized subsets of shares refer to subsets of the signers with combined weight is greater than a threshold w, rather than subsets of more than t signers. Similarly, note that the special case of threshold VUF with $(n, t) = (1, 0)$ is a single-server VUF.

3.2 Threshold VUF Security with DKG

In Definition 3 of §3.1 we defined the security of threshold VUF with an idealized key generation. The definition with distributed key generation (DKG) is almost identical to Definition 3, except signers use an interactive DKG protocol to generate the VUF keys. More precisely, signers use a DKG protocol instead of KeyGen in Definition 3, where the DKG is defined as below:

$\mathsf{DKG}(n, t) \rightarrow \mathsf{pk}, \{\mathsf{pk}_i\}_{i \in [n]}, \{\mathsf{sk}_i\}_{i \in [n]}$: This is an interactive protocol among n signers, which all take as inputs some public parameters, a security parameter $\lambda \in \mathbb{N}$ as well as a pair of integers $n, t \in \mathsf{poly}(\lambda)$. At the end of the protocol, signers output a VUF public key pk and a vector of public keys $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$. Each signer i also outputs the secret key sk_i.

The Correctness, Uniqueness, and Unforgeability definitions of the threshold VUF with a DKG are identical to Definitions 4 to 6, respectively, except in the $\mathsf{UF\text{-}CMA}^{\mathcal{A},t}$ game where we run the $\mathsf{DKG}(n, t)$ protocol instead of the $\mathsf{KeyGen}(n, t)$ functionality.

The definition for *weighted VUF security with DKG* is similar, except that we change the requirement that the adversary controls at most t participants with the requirement that the adversary can control participants with combined weight is at most w.

4 A Weighted VUF

Our goal is to construct a weighted verifiable unpredictable function (VUF), in a setting where each participant might have a different weight. The functionality we want to implement is a special case of a threshold VUF with a w-out-of-W access structure, where participant i has w_i shares, and $W = \sum_{i=1}^{n} w_i$. The goal is that the computation and communication overhead of each participant will be sub-linear in its weight w_i (it will actually be constant).

Ideally, we would like to have a protocol where each participant has a single share, regardless of its weight. We do not achieve this goal. Instead, the number of shares that each participant has is proportional to its weight, but the contribution of each participant to the computation of the weighted VUF is of constant size. Also, each participant incurs a constant amount of computation costs to compute its contribution. In terms of aggregation, the computation of the weighted VUF requires a constant number of pairings per participant. The weight w_i of a participant only affects a multi-exponentiation of w_i values rather than computing w_i pairings.

```
$\mathsf{VUF.Setup}(h, \hat{g}) \to \mathsf{crs}$
**return** $(h, \hat{g}) \in \mathbb{G} \times \hat{\mathbb{G}}$

$\mathsf{VUF.KeyGen}(1^\kappa) \to (\mathsf{sk}, \mathsf{pk})$
$a \leftarrow\$ \mathbb{F}$;   $\mathsf{sk} := h^a$;   $\mathsf{pk} := \hat{g}^a$
**return** $(\mathsf{sk}, \mathsf{pk})$

$\mathsf{VUF.AugmentKeyPair}(\mathsf{sk}, \mathsf{pk}) \to (\mathsf{ask}, \mathsf{apk})$
$r \leftarrow\$ \mathbb{F}$;   $\pi := h^r$;   $\mathsf{rk} := \mathsf{sk}^r$ // $= (h^a)^r$
$\mathsf{ask} := r$;   $\mathsf{apk} := (\pi, \mathsf{rk})$
**return** $\mathsf{ask}, \mathsf{apk}$

$\mathsf{VUF.PubKeyVerify}(\mathsf{pk}, \mathsf{apk}) \to \{0, 1\}$
$(\pi, \mathsf{rk}) := \mathsf{apk}$
**assert** $e(\pi, \mathsf{pk}) = e(\mathsf{rk}, \hat{g})$
// $\Leftrightarrow e(h^r, \hat{g}^a) = e(h^{ar}, \hat{g})$

$\mathsf{VUF.Sign}(\mathsf{ask}, m) \to \sigma$:
$r := \mathsf{ask}$;   $\sigma := \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r}$
**return** $\sigma$

$\mathsf{VUF.Verify}(\mathsf{apk}, m, \sigma) \to \{0, 1\}$:
$(\pi, \cdot) := \mathsf{apk}$
// checks $\mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r}$ is computed correctly
**assert** $e(\pi, \sigma) = e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m))$
// $\Leftrightarrow e(h^r, \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r}) = e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m))$

$\mathsf{VUF.Derive}(\mathsf{apk}, \sigma) \to \mathsf{vuf}$:
$(\cdot, \mathsf{rk}) := \mathsf{apk}$
**return** $e(\mathsf{rk}, \sigma)$
// $= e(\mathsf{sk}^r, \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r}) = e(\mathsf{sk}, \mathsf{H}_{\hat{\mathbb{G}}}(m))$
```

Fig. 1: Single server VUF.

Since a VUF scheme corresponds to a signature scheme (see [MRV99] for a discussion on this subject), we often refer to the participants in the VUF protocol as *signers*, which compute *signature shares* that are combined to derive the final VUF output.

We outline the VUF construction in a series of steps. Initially, we present a construction where the key is only known only to a single server. Subsequently, we introduce an unweighted threshold construction, followed by a weighted threshold construction.

4.1 Setting the Ground - a Single-Server VUF

We start by describing VUF where a single participant knows the secret signing key and is responsible for signing. Hence this is a *single-sever* VUF. This VUF illustrates the core ideas that are used in our threshold and weighted VUFs. The single-server VUF is summarized in Figure 1, and is described next.

Setup and key generation. The public parameters are two generators $(h, \hat{g}) \in \mathbb{G} \times \hat{\mathbb{G}}$. For a uniform random $a \leftarrow\$ \mathbb{F}$, the secret and public keys of the server are $\mathsf{sk} := h^a$ and $\mathsf{pk} := \hat{g}^a$, respectively. The server uses the algorithm KeyGen to generate $(\mathsf{sk}, \mathsf{pk})$.

Next, the server uses AugmentKeyPair to generate an augmented public key $(\mathsf{ask}, \mathsf{apk})$, where $\mathsf{apk} := (\pi, \mathsf{rk}) = (h^r, \mathsf{sk}^r)$ and $\mathsf{ask} := r$ for some $r \leftarrow\$ \mathbb{F}$, only known to the server. This value will be used for verifying VUF outputs. The server publishes apk and stores ask privately. Any external entity with access to pk, can validate the correctness of apk using the PubKeyVerify algorithm.

VUF computation. Let $\mathsf{H}_{\hat{\mathbb{G}}} : \mathcal{M} \to \hat{\mathbb{G}}$ for message space $\mathcal{M}$ be a hash function modeled as a random oracle. The VUF output is defined as $e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m))^a$.

The VUF output cannot be verified directly. An initial signature of the input is computed using the algorithm VUF.Sign. This value can be verified. Then, the algorithm VUF.Derive derives from it the final VUF output. This separation help us design the threshold and weighted VUFs in a modular way.

In more detail, for a message m the server first calculates $\sigma := \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r}$ using VUF.Sign, and then calculates from it the VUF output using the VUF.Derive algorithm.

VUF verification. Any external verifier $\mathcal{V}$ with access to apk can verify the VUF signature σ using VUF.Verify algorithm. Precisely, VUF.Verify checks whether the exponents of σ and rk are multiplicative inverses of each other. Since VUF.Derive computes the VUF output as a *deterministic* function of σ, this is also a verification of the VUF output.

```
$\underline{\mathsf{VUF.Setup}(h, \hat{g})} \to \mathsf{crs}$
return $(h, \hat{g}) \in \mathbb{G} \times \hat{\mathbb{G}}$

$\underline{\mathsf{VUF.KeyGen}(a, t, n)} \to (\mathsf{sk}_i, \mathsf{pk}_i)_{i \in [n]}, \mathsf{pk}$:
$a \leftarrow\$ \mathbb{F}$
$(a_i)_{i \in [n]} \leftarrow \mathsf{ShamirSecretShare}(a, t, n)$
$(\mathsf{sk}_i, \mathsf{pk}_i) := (h^{a_i}, \hat{g}^{a_i})$
return $(\mathsf{sk}_i, \mathsf{pk}_i)_{i \in [n]}$

$\underline{\mathsf{VUF.AugmentKeyPair}(\mathsf{sk}_i, \mathsf{pk}_i)} \to (\mathsf{ask}_i, \mathsf{apk}_i)$
$r_i \leftarrow\$ \mathbb{F}$;   $\pi_i := h^{r_i}$;   $\mathsf{rk}_i := \mathsf{sk}^{r_i}$ // $= (h^{a_i})^{r_i}$
$\mathsf{apk}_i := (\pi_i, \mathsf{rk}_i)$;   $\mathsf{ask}_i := r_i$
return $(\mathsf{ask}_i, \mathsf{apk}_i)$

$\underline{\mathsf{VUF.PubKeyVerify}(\mathsf{pk}_i, \mathsf{apk}_i)} \to \{0, 1\}$
// As in the single-server case
$(\pi_i, \mathsf{rk}_i) := \mathsf{apk}_i$
assert $e(\pi_i, \mathsf{pk}_i) = e(\mathsf{rk}_i, \hat{g})$

$\underline{\mathsf{VUF.ShareSign}(\mathsf{ask}_i, m)} \to \sigma_i$:
// As in the single-server case
$r_i := \mathsf{ask}_i$;   $\sigma_i := \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r_i}$
return $\sigma_i$

$\underline{\mathsf{VUF.ShareVerify}(\mathsf{apk}_i, m, \sigma_i)} \to \{0, 1\}$:
// As in the single-server case
$(\pi_i, \cdot) := \mathsf{apk}_i$
assert $e(\pi_i, \sigma_i) = e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m))$

$\underline{\mathsf{VUF.Derive}(\mathsf{T}, \mathsf{m}, \sigma, \mathsf{apk})} \to \mathsf{vuf}$:
$(\sigma_i)_{i \in T} := \sigma$
assert $e(\pi_i, \sigma_i) = e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m)), \quad \forall i \in T$
$(\ell_i)_{i \in T} := \mathsf{LagrangeCoefficients}(\mathsf{T})$
$\forall i \in T$, recover $\mathsf{apk}_i$; set $(\cdot, \mathsf{rk}_i) := \mathsf{apk}_i$
return $\prod_{i \in T} e(\mathsf{rk}_i^{\ell_i}, \sigma_i)$
```

Fig. 2: Threshold VUF.

Security. Note that the single-server VUF is a degenrate case of threshold VUF with $n = 1$ and $t = 0$. Thus, similar the threshold VUF scheme (see Definition 3), we require the single-server VUF to satisfy the *correctness, uniqueness* and *unpredictability* properties.

4.2 An Unweighted Threshold VUF

The threshold VUF is described with n participants, where at least $t + 1$ participants are needed in order to compute the VUF output. The construction is summarized in Figure 2.

Setup and key generation. The public parameters of the threshold VUF are the same as that of the single-server VUF, i.e., generators $(h, \hat{g}) \in \mathbb{G} \times \hat{\mathbb{G}}$, and a hash function $\mathsf{H}_{\hat{\mathbb{G}}}$. The KeyGen algorithm computes the signing and public keys of all participants/signers. It first samples a random polynomial $a(x)$ of degree t. Let $a := a(0)$. The secret key of the threshold VUF is $\mathsf{sk} := h^a$, and the public key is $\mathsf{pk} := \hat{g}^a$. (The private key is kept hidden from all participants, and the public key is not explicitly used.) The output of the VUF on input m is defined as $e(\mathsf{sk}, \mathsf{H}_{\hat{\mathbb{G}}}(m)) = e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m))^a$.

Let $a_i := a(i)$. The signing key sk and public key pk of signer i are $(\mathsf{sk}_i, \mathsf{pk}_i) := (h^{a_i}, \hat{g}^{a_i})$, respectively.

After KeyGen, each signer i locally generates its augmented public-private key pair as follows. Signer i first samples an augmented secret key $\mathsf{ask} := r \leftarrow\$ \mathbb{F}$. The corresponding augmented public key is $\mathsf{apk}_i := (\pi_i, \mathsf{rk}_i) := (h^r, \mathsf{sk}^r)$. Each signer i then publishes its augmented public key apk_i. Also, upon receiving apk_j from signer j, each signer validates it using the $\mathsf{VUF.PubKeyVerify}$ algorithm, which verifies that π_j and rk_j have the same discrete logarithms with respect to h and sk_i, respectively.

VUF computation and verification. The per-signer signing algorithm is identical to that of single server VUF, except each signer i uses its private key ask_i to compute $\sigma_i := \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r_i}$. Similarly, to verify the VUF output σ_j from signer j, signers uses the $\mathsf{VUF.Verify}$ algorithm that is similar to that of the single-server VUF.

VUF output derivation. Any aggregator, upon receiving valid VUF outputs from a set of signers T with $|T| \geq t + 1$, computes the VUF output ρ as:

$$\rho := \prod_{i \in T} e(\mathsf{rk}_i^{\ell_i}, \sigma_i); \tag{1}$$

where ℓ_i is the Lagrange coefficient for the value $a(i)$ in the set T.

Verification. As in previous work on threshold VUF [GJM+21a], the final VUF output is not directly verifiable. Instead, it is possible to verify the signature shares that are used to derive the final output, by checking $e(\pi_i, \sigma_i) = e(h, \mathsf{H}_{\mathbb{G}}(m))$ holds for each $i \in T$.[3]

Analysis. In §5, we prove that this threshold VUF guarantees the correctness, uniqueness, and unforgeability properties, as defined in §3.1. Unforgeability is proved based on BDH hardness. In terms of performance, per-signer signing require one exponentiation, and per-signature verification requires two bilinear pairings. To derive the final output, an aggregator needs to perform $O(n \log^2 n)$ scalar operations to compute the Lagrange coefficients [TCZ+20], and $|T| + 1$ group exponentiations and bilinear pairings.

4.3 The Weighted VUF Protocol

Let w_i be the weight of signer i. Also, let $W := \sum_{i\in[n]} w_i$ be the total weight, and w be the VUF threshold. We summarize our (W, w) weighted VUF scheme in Figure 3, and give more details below.

Main differences from the threshold case. Unlike the unweighted threshold case, each signer i is assigned w_i shares. Namely, there is a polynomial $a(\cdot)$ of degree w, and signer i receives w_i secret shares of the form $h^{a(x)}$ for w_i different x values. The augmented public key of signer i contains $w_i + 1$ values, which include h^{r_i}, and each of the shares of this signer raised to the power of r_i. The signature share of signer i is still $\mathsf{H}_{\mathbb{G}}(m)^{1/r_i}$, exactly as in the threshold case. The fact that the size of the signature is independent of the weight is the major advantage of this construction. This is due to the fact that $e(h^{a(j)\cdot r_i}, \mathsf{H}_{\mathbb{G}}(m)^{1/r_i}) = e(h, \mathsf{H}_{\mathbb{G}}(m))^{a(j)}$ for all shares $h^{a(j)}$. Our final signature derivation is similar to the unweighted case, except we work with a polynomial of degree w. We next provide details about our weighted VUF construction.

Setup and key generation. The public parameters are the same as of our single-server VUF, i.e.,the generators $(h, \hat{g}) \in \mathbb{G} \times \hat{\mathbb{G}}$, and a hash function $\mathsf{H}_{\hat{\mathbb{G}}}$. Signer i with weight w_i receives w_i signing keys and has w_i corresponding public keys. Both signing and public keys correspond to the evaluations of a degree w polynomial $a(x)$ in the exponent. We use the notation $a_i := a(i)$. For signer i we define $s_i := \sum_{j=1}^{i-1} w_i$, and use the values $a_{s_i+1}, \dots, a_{s_i+w_i}$ as the exponents. As in the threshold setting, the secret key is $\mathsf{sk} := h^a$, and the public key is $\mathsf{pk} := \hat{g}^a$ (although they are not explicitly used) and the VUF output on input m is defined as $e(\mathsf{sk}, \mathsf{H}_{\hat{\mathbb{G}}}(m)) = e(h^a, \mathsf{H}_{\hat{\mathbb{G}}}(m))$.

Augmented public key, and its verification. Next, each signer i locally computes the augmented public-private keys $(\mathsf{apk}_i, \mathsf{ask}_i)$ using VUF.AugmentKeyPair, where $\mathsf{ask}_i := r_i \leftarrow\$ \mathbb{F}$, and apk_i consists of $\pi_i := h^{r_i}$ and w_i values $\mathsf{rk}_{i,j} := (\mathsf{sk}_{i,j})^{r_i}$, for each $j \in [w_i]$. Similar to threshold VUF, each signer i broadcasts its augmented public key apk_i. Upon receiving $\mathsf{apk}_j := (\pi_j, \{\mathsf{rk}_{j,k}\}_{k\in[w_j]})$ from signer j, each signer validates apk_j using the VUF.PubKeyVerify algorithm. This verification can be batched by sampling w_j random coefficients $(r_1, \dots, r_{w_j}) \leftarrow\$ \mathbb{F}^{w_j}$ and checking that,

$$e(\pi_j \ , \prod_{k\in[w_j]} \mathsf{pk}_{j,k}^{r_k}) \cdot e(\prod_{k\in[w_j]} \mathsf{rk}_{j,k}^{r_k} \ , \ \hat{g}^{-1}) = 1_{\mathbb{G}_T}. \tag{2}$$

Intuitively, Eq. (2) checks that all elements of the augmented public key have the same exponent, for the appropriate bases. The exponents in Eq. (2), can also be sampled from a smaller domain S, resulting in a failure probability of $1/|S|$. Further optimization is possible by batching verification across different signers.

VUF evaluation and verification. The VUF evaluation and signature verification are identical to the non-weighted case, and are *independent* of the signer's weight.

VUF derivation. Given a set T of signers, and their corresponding secrets $\{a_{s_i+1}, \dots, a_{s_i+w_i}\}_{i\in T}$, and assuming that their total weight satisfies $\sum_{i\in T} w_i > w$, there exist Lagrange coefficients $\{\ell_{i,j}\}_{i\in T; j\in[w_i]}$ such that the shared secret a can be expressed as $a = \sum_{i\in T} \sum_{j\in w_i} \ell_{i,j} a_{s_i+j}$.

[3] This pairing-based verification can be replaced by a more efficient verification based on a sigma-protocol: Each participant submits a signature share $\sigma_i = \mathsf{H}_{\mathbb{G}}(m)^{1/r_i}$ and attaches to it a Fiat-Shamir non-interactive proof that $\log_{\sigma_i} \mathsf{H}_{\mathbb{G}}(m) = \log_h \pi_i$. The signer can compute this proof since it knows the discrete log r_i. This approach requires sending three more group elements with the share σ_i, and using two exponentiations to verify the proof, but the computation will be faster than using pairings (see §2.2).

```
$\underline{\mathsf{VUF.Setup}(h, \hat{g}) \to \mathsf{crs}}$
$\textbf{return } (h, \hat{g}) \in \mathbb{G} \times \hat{\mathbb{G}}$

$\underline{\mathsf{VUF.KeyGen}(w, (w_i)_{i\in[n]})} \to ((\mathsf{sk}_i, \mathsf{pk}_i)_{i\in[n]}, \mathsf{pk})$:
    Let $W := \sum_{i\in[n]} w_i$;   $s_i := \sum_{j=1}^{i-1} w_i, \forall i \in [n]$
    $a \leftarrow\$ \mathbb{F}$
    $(a_j)_{j\in[W]} \leftarrow \mathsf{ShamirSecretShare}(a, w, W)$
    $\mathsf{sk}_i := (h^{a_{s_i+1}}, \dots, h^{a_{s_i+w_i}}) \overset{\mathsf{def}}{=} (\mathsf{sk}_{i,1}, \dots \mathsf{sk}_{i,w_i})$
    $\mathsf{pk}_i := (\hat{g}^{a_{s_i+1}}, \dots, \hat{g}^{a_{s_i+w_i}}) \overset{\mathsf{def}}{=} (\mathsf{pk}_{i,1}, \dots \mathsf{pk}_{i,w_i})$
    $\textbf{return } (\mathsf{sk}_i, \mathsf{pk}_i)_{i\in[n]}$

$\underline{\mathsf{VUF.AugmentKeyPair}(\mathsf{sk}_i, \mathsf{pk}_i)} \to (\mathsf{ask}_i, \mathsf{apk}_i)$
    $(h^{a_{s_i}}, \dots h^{a_{s_i+w_i-1}}) := \mathsf{sk}_i$
    $r_i \leftarrow\$ \mathbb{F}$;   $\mathsf{ask}_i := r_i$
    $\mathsf{apk}_i := (h^{r_i}, h^{r_i \cdot a_{s_i+1}}, \dots h^{r_i \cdot a_{s_i+w_i}}) \overset{\mathsf{def}}{=} (\pi_i, \mathsf{rk}_{i,1}, \dots \mathsf{rk}_{i,w_i})$
    $\textbf{return } (\mathsf{ask}_i, \mathsf{apk}_i)$

$\underline{\mathsf{VUF.PubKeyVerify}(\mathsf{pk}_i, \mathsf{apk}_i)} \to \{0, 1\}$
    $(\mathsf{pk}_{i,1}, \dots \mathsf{pk}_{i,w_i}) := \mathsf{pk}_i$
    $(\pi_i, \mathsf{rk}_{i,1}, \dots \mathsf{rk}_{i,w_i}) := \mathsf{apk}_i$
    $\textbf{assert } e(\pi_i, \mathsf{pk}_{i,j}) = e(\mathsf{rk}_{i,j}, \hat{g})$;   $\forall j \in [w_i]$
    // Batching reduces the number of pairings to 2
    // Details are in the paragraph on "Key verification"

$\underline{\mathsf{VUF.ShareSign}(\mathsf{ask}_i, m)} \to \sigma_i$:
    // As in the threshold and single-server cases
    $r_i := \mathsf{ask}_i$;   $\sigma_i := \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r_i}$
    $\textbf{return } \sigma_i$

$\underline{\mathsf{VUF.ShareVerify}(\mathsf{apk}_i, m, \sigma_i)} \to \{0, 1\}$:
    // As in the threshold and single-server cases
    $(\pi_i, \cdot) := \mathsf{apk}_i$
    $\textbf{assert } e(\pi_i, \sigma_i) = e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m))$

$\underline{\mathsf{VUF.Derive}(\mathsf{T}, \mathsf{m}, \sigma)} \to \mathsf{vuf}$:
    $(\sigma_i)_{i\in T} := \sigma$
    $\textbf{assert } e(\pi_i, \sigma_i) = e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m))$,   $\forall i \in T$
    $\forall i \in T$, recover $\mathsf{apk}_i$; set $(\cdot, \mathsf{rk}_{i,1}, \dots \mathsf{rk}_{i,w_i}) := \mathsf{apk}_i$
    $(\ell_{i,j})_{i\in T} := \mathsf{WeightedLagCoeffs}(T, (s_i, w_i)_{i\in T}, w)$
    $\textbf{return } \prod_{i\in T} e(\prod_{j\in[w_i]} \mathsf{rk}_{i,j}^{\ell_{i,j}}, \sigma_i)$
    // $= \prod_{i\in T} e(\prod_{j\in[w_i]} \mathsf{sk}_{i,j}^{r_i \cdot \ell_{i,j}}, \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r_i})$
    // $= e(\mathsf{sk}, \mathsf{H}_{\hat{\mathbb{G}}}(m))$
```

Fig. 3: Weighted VUF.

Any aggregator, upon receiving valid VUF outputs from a set of signers with combined weight greater than w, can therefore compute the VUF output as shown in VUF.Derive:

$$\prod_{i\in T} e(\prod_{j\in[w_i]} \mathsf{rk}_{i,j}^{\ell_{i,j}}, \sigma_i) = \prod_{i\in T} \prod_{j\in w_i} e(\mathsf{rk}_{i,j}^{\ell_{i,j}}, \sigma_i)$$
$$= \prod_{i\in T} \prod_{j\in w_i} e(\mathsf{sk}_{i,j}^{r_i \cdot \ell_{i,j}}, \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r_i}) = \prod_{i\in T} \prod_{j\in w_i} e(\mathsf{sk}_{i,j}^{\ell_{i,j}}, \mathsf{H}_{\hat{\mathbb{G}}}(m))$$
$$= \prod_{i\in T} \prod_{j\in w_i} e(h^{a_{s_i+j} \cdot \ell_{i,j}}, \mathsf{H}_{\hat{\mathbb{G}}}(m)) = e(h^a, \mathsf{H}_{\hat{\mathbb{G}}}(m)) = e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m))^a$$

Analysis. In §5, we prove that this weighted VUF guarantees the correctness, uniqueness, and unforgeability properties. With regards to performance, there is a one-time $O(W)$ communication and computation cost to send and verify the augmented keys. Following that step, the weighted VUF has several advantages over a naïve approach of running the threshold VUF with W participants: (i) The per-signer VUF evaluation cost and VUF output size are constant, independent of the weight of the signer. (ii) The cost of verifying a signature share sent by a signer is also constant, i.e., only two pairings. (iii) Overall, the aggregator computes only $O(|T|)$ pairings in order to verify all shares and derive the final output.[4]

[4] In a blockchain setting, the size of the set T is typically small, as the Nakamoto coefficient of most blockchains is smaller than 20 (`https://nakaflow.io/` as of Jan. 26, 2024).

Input: $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c) \in \mathbb{G}^3 \times \hat{\mathbb{G}}^4$.

Setup and Key Generation:

1. Set $\hat{g} = \hat{g}$ and $h = g^b$. Send $(h, \hat{g})$ to $\mathcal{A}$.
2. Let $\mathsf{pk} = \hat{g}^a$ and $\mathsf{sk} = h^a = g^{ab}$. Note that $\mathcal{A}_{\mathrm{BDH}}$ cannot compute sk, assuming hardness of BDH.
3. Sample $u \leftarrow\$ \mathbb{F}$. We have $u = br$ for some $r \in \mathbb{F}$ unknown to $\mathcal{A}_{\mathrm{BDH}}$. Compute $\pi = g^u = g^{br} = h^r$ and $\mathsf{rk} = (g^a)^u = g^{abr} = h^{ar} = \mathsf{sk}^r$. The augmented public key is then $\mathsf{apk} = (\pi, \mathsf{rk})$. Send $(\mathsf{pk}, \mathsf{apk})$ to $\mathcal{A}$.

Simulating signing and random oracle queries:

1. Let q_{H} be a upper bound on the number of random oracle queries that $\mathcal{A}$ makes.
2. Let $k' \leftarrow\$ [q_{\mathsf{H}}]$. On the k-th random oracle query on a message m_k, if $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) \neq \perp$, return $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. Otherwise, if $k \neq k'$, then sample $x_k \leftarrow\$ \mathbb{F}$, set $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^{x_k}$ and store (m_k, x_k), and return $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. When $k = k'$, return $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^c$.
3. On k-th signing query on message m_k, if $m_k \neq m_{k'}$, output $\sigma = \hat{g}^{bx_k/u}$. Otherwise, abort.

Let ρ be $\mathcal{A}$'s VUF output for the message $m_{k'}$, then $\mathcal{A}_{\mathrm{BDH}}$ outputs ρ as the BDH output.

Fig. 4: Single server VUF security reduction

5 Security Analysis of Running the VUF with Trusted Key Generation

This section contains three parts, proving the security of the single-signer VUF, the threshold VUF, and the weighted VUF. We reiterate that in this section we prove security assuming that keys are generated by a trusted key generation algorithm. We analyze the security of our weighted VUF when instantiated with our DKG in §9.

5.1 Single Server VUF Security

The *correctness* property can be verified by observation. The *uniqueness* property follows from the following three arguments: First, for a fixed VUF public key g^a and for any given message m, the *unique* VUF output is $\rho = e(\mathsf{H}_{\hat{\mathbb{G}}}(m), \hat{g}^a)$. Second, for any augmented public key $\mathsf{apk} = (\pi, \mathsf{rk})$, there exists a *unique* σ which satisfies the validity check $e(\sigma, \pi) = e(\mathsf{H}_{\hat{\mathbb{G}}}(m), \hat{h})$. Third, a valid (σ, apk) pair always generates the unique VUF output ρ.

Next, we prove he unforgeability property assuming the hardness of the computational Bilinear Diffie-Hellman (BDH) assumption (Definition 2) in the random oracle model, where we model the hash function $\mathsf{H}_{\hat{\mathbb{G}}}$ as a random oracle.

Theorem 2 (Single Server VUF). *Assuming the hardness of computational bilinear Diffie-Hellman (BDH), the single-server VUF protocol of Figure 1 is existentially unforgeable as per Definition 6.*

Proof. We show that if there exists an adversary $\mathcal{A}$ that can break the security of our single signer VUF with probability $\varepsilon_{\mathrm{VUF}}$ then there exists an adversary $\mathcal{A}_{\mathrm{BDH}}$ which can use $\mathcal{A}$ to break the BDH assumption with the probability $\varepsilon_{\mathrm{BDH}} \geq \varepsilon_{\mathrm{VUF}}/q_{\mathsf{H}}$. Here, q_{H} is the upper bound on the number of random oracle queries that $\mathcal{A}$ makes. In the reduction $\mathcal{A}_{\mathrm{BDH}}$ receives an input $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c) \in \mathbb{G}^3 \times \hat{\mathbb{G}}^4$ and interacts with $\mathcal{A}$ to break BDH, as described in Figure 4.

First, suppose that $\mathcal{A}$ successfully forged the VUF output for the message $m_{k'}$. Let ρ be the value $\mathcal{A}$ outputs as the VUF output on $m_{k'}$. Note that ρ is the unique VUF output such that $\rho = e(h^a, \mathsf{H}_{\hat{\mathbb{G}}}(m_{k'}))$. Since, $\mathcal{A}_{\mathrm{BDH}}$ programs $\mathsf{H}_{\hat{\mathbb{G}}}(m_{k'}) = \hat{g}^c$ and $h = g^b$, this implies that $\rho = e(g, \hat{g})^{abc}$.

Observe that in Figure 4 $\mathcal{A}_{\mathrm{BDH}}$ produces $\mathsf{pk}, \mathsf{apk}$ with the same distribution as in the real run, and outputs uniform random group elements for every unique random oracle query. Also, for every message $m_k \neq m_{k'}$, $\mathcal{A}_{\mathrm{BDH}}$ outputs a σ which satisfy the validity check $e(\pi, \sigma) = e(h^r, \hat{g}^{bx_k/r}) = e(h, \hat{g}^{bx_k}) = e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m))$. This implies that, conditioned on the event that $\mathcal{A}_{\mathrm{BDH}}$ correctly guesses the forged message $m_{k'}$, the view of $\mathcal{A}$ during its interaction with $\mathcal{A}_{\mathrm{BDH}}$ is identically distributed as $\mathcal{A}$'s view real-protocol execution. More precisely, let F_{real} and F_{sim} be the event that $\mathcal{A}$

Input: $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c) \in \mathbb{G}^\times \hat{\mathbb{G}}^4$.

Setup and Key Generation:

1. Set $h = g^b$. Send $(h, \hat{g})$ to $\mathcal{A}$.
2. Let $\mathcal{M} \subset [n]$ with $|\mathcal{M}| \leq t$ be the set of malicious participants. Let $\mathcal{H} = [n] \setminus \mathcal{M}$ be the set of honest participants.
3. Let $\mathsf{pk} = \hat{g}^a$. This implies that $\mathsf{sk} = h^a = g^{ab}$. Note that $\mathcal{A}_{\text{BDH}}$ cannot explicitly compute sk.
4. For each malicious participant $j \in \mathcal{M}$, sample its secret key $a_j \leftarrow\$ \mathbb{F}$. Let $a(\cdot) \in \mathbb{F}[x]^t$ be a degree t polynomial such that $a(0) = a$ and $a(j) = a_j$ for $j \in \mathcal{M}$. (If $|\mathcal{M}| = t$ then this polynomial is uniquely defined. Otherwise, choose at random $t - |\mathcal{M}|$ additional values $a(j)$ for values $j > n$ in order to define the polynomial.) Compute g^{a_i} for each $i \in [n]$ by interpolating in the exponent.
5. Send $\mathsf{pk} = \hat{g}^a$, $\{\mathsf{pk}_i = \hat{g}^{a_i}\}_{i\in[n]}$, $\{h^{a_j}\}_{j\in\mathcal{M}}$ to $\mathcal{A}$.
6. For each honest participant $i \in \mathcal{H}$, sample $u_i \leftarrow\$ \mathbb{F}$. It holds that $u_i = b \cdot r_i$ for some $r_i \in \mathbb{F}$ unknown to $\mathcal{A}_{\text{BDH}}$. Compute $\pi_i = g^{u_i} = h^{r_i}$. Then, compute g^{a_i} using interpolation in the exponent, and compute $\mathsf{rk}_i = g^{a_i u_i} = g^{a_i b r_i} = h^{a_i r_i} = \mathsf{sk}_i^{r_i}$. The augmented public key is then $\mathsf{apk}_i = (\pi_i, \mathsf{rk}_i)$. Send $\{\mathsf{apk}_i\}_{i\in\mathcal{H}}$ to $\mathcal{A}$.
7. Receive a valid apk_i for each $i \in \mathcal{M}$ from $\mathcal{A}$.

Simulating signing and random oracle queries:

1. Let q_{H} be an upper bound on the number of random oracle queries that $\mathcal{A}$ makes.
2. Sample a random index $k' \leftarrow\$ [q_{\mathsf{H}}]$.
3. On the k-th random oracle query on a message m_k, if $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) \neq \perp$, return $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. Otherwise, if $k \neq k'$, sample $x_k \leftarrow\$ \mathbb{F}$, set $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^{x_k}$, store (m_k, x_k), and return $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. When $k = k'$, return $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^c$.
4. On k-th partial signature query (i, m_k) for signer $i \in \mathcal{H}$, if $m_k \neq m_{k'}$, output $\sigma = (\hat{g}^b)^{x_k/u_i} = \mathsf{H}_{\hat{\mathbb{G}}}(m_k)^{b/u_i} = \mathsf{H}_{\hat{\mathbb{G}}}(m_k)^{1/r_i}$. Otherwise, abort.

Let ρ be $\mathcal{A}$'s VUF output for the message $m_{k'}$, then $\mathcal{A}_{\text{BDH}}$ outputs ρ as the BDH output.

Fig. 5: Threshold VUF security reduction

outputs a successful forgery in real protocol execution and during its interaction with $\mathcal{A}_{\text{BDH}}$, respectively. Also, let E be the event tha $\mathcal{A}_{\text{BDH}}$ correctly guesses the forged message, then we get:

$$\varepsilon_{\text{BDH}} \geq \Pr[F_{\mathsf{sim}}] \geq \Pr[F_{\mathsf{real}} \wedge E] \tag{3}$$

$$= \Pr[F_{\mathsf{real}} \mid E] \cdot \Pr[E] \geq \varepsilon_{\text{VUF}} \cdot \frac{1}{q_{\mathsf{H}}} \tag{4}$$

$$\implies \varepsilon_{\text{BDH}} \geq \frac{\varepsilon_{\text{VUF}}}{q_{\mathsf{H}}} \tag{5}$$

Here, Eq. (4) follows from the fact that the F_{real} is independent of the event E and $\Pr[E]$ is at least $1/q_{\mathsf{H}}$. (This reduction, as well as the following ones, can be made tighter using techniques such as in [Cor00].)

5.2 Threshold VUF Security

The *correctness* property can be easily verified through observation. The *uniqueness* property follows from the following arguments:

– The public key verification in VUF.PubKeyVerify checks that $e(\pi_i, \mathsf{pk}_i) \stackrel{?}{=} e(\mathsf{rk}_i, \hat{g})$. Since $\mathsf{pk}_i = \hat{g}^{a_i}$, verification only succeeds if for some value r_i it holds that $\pi_i = h^{r_i}$ and $\mathsf{rk}_i = (h^{a_i})^{r_i} = \mathsf{sk}^{r_i}$.
– Share verification in the function VUF.ShareVerify checks whether $e(\pi_i, \sigma_i) \stackrel{?}{=} e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m))$. Since $\pi_i = h^{r_i}$ and therefore $e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m)) = e(h, \mathsf{H}_{\hat{\mathbb{G}}}(m))^{r_i \cdot 1/r_i} = e(h^{r_i}, \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r_i}) = e(\pi_i, \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r_i})$, this check succeeds only if $\sigma_i = \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r_i}$. This is the only share value that participant i can provide for m.

– Output derivation in the function $\mathsf{VUF.Derive}$ returns the value $\prod_{i\in T} e(\mathsf{rk}_i, \sigma_i^{\ell_i})$. It holds that all the shares of $i \in T$ passed the check in $\mathsf{VUF.ShareVerify}$. Therefore the function output is as follows and is always equal to $e(\mathsf{sk}, \mathsf{H}_{\hat{\mathbb{G}}}(m))$.

$$\begin{aligned}\prod_{i\in T} e(\mathsf{rk}_i^{\ell_i}, \sigma_i) &= \prod_{i\in T} e((\mathsf{sk}_i^{r_i})^{\ell_i}, \mathsf{H}_{\hat{\mathbb{G}}}(m)^{1/r_i}) = \prod_{i\in T} e(\mathsf{sk}_i^{\ell_i}, \mathsf{H}_{\hat{\mathbb{G}}}(m)) \\ &= e(\prod_{i\in T} \mathsf{sk}_i^{\ell_i}, \mathsf{H}_{\hat{\mathbb{G}}}(m)) = e(\prod_{i\in T} g^{a_i\ell_i}, \mathsf{H}_{\hat{\mathbb{G}}}(m)) \\ &= e(g^{\sum_{i\in T} a_i\ell_i}, \mathsf{H}_{\hat{\mathbb{G}}}(m)) = e(\mathsf{sk}, \mathsf{H}_{\hat{\mathbb{G}}}(m))\end{aligned}$$

The *unforgeability* proprety is based on the computational Bilinear Diffie-Hellman (BDH) assumption in the random oracle model.

Theorem 3 (Threshold VUF security). *Assuming the hardness of bilinear Diffie-Hellman (BDH), the threshold VUF protocol of Figure 2 is existentially unforgeable as per Definition 6.*

Proof. The proof shows that, if an adversary $\mathcal{A}$ which corrupts up to t out of n participants succeeds in forging a VUF output of the threshold VUF scheme with probability $\varepsilon_{\mathrm{VUF}}$, then it is possible build an adversary $\mathcal{A}_{\mathrm{BDH}}$ which uses $\mathcal{A}$ to break the BDH assumption with the probability $\varepsilon/q_{\mathsf{H}}$. Here, q_{H} is the upper bound on the number of random oracle queries that $\mathcal{A}$ makes.

Let $g \in \mathbb{G}$ and $\hat{g} \in \hat{\mathbb{G}}$ be the generators of the groups. $\mathcal{A}_{\mathrm{BDH}}$ is given an input $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c) \in \mathbb{G}^3 \times \hat{\mathbb{G}}^4$. It interacts with $\mathcal{A}$ to break the BDH assumption, as in Figure 5. Let ρ be the VUF output on the message $m_{k'}$. The uniqueness property of our threshold VUF implies that $\rho = e(h^a, \mathsf{H}_{\hat{\mathbb{G}}}(m_{k'})) = e((g^b)^a, \hat{g}^c) = e(g, \hat{g})^{abc}$.

Let $\varepsilon_{\mathrm{tVUF}}$ be the probability with which an $\mathcal{A}$ forges a VUF output in the real-execution of our threshold VUF protocol. Then, using an similar argument as in Theorem 2, $\mathcal{A}$ forges a VUF output during its interaction with $\mathcal{A}_{\mathrm{BDH}}$ with probability at least $\varepsilon_{\mathrm{tVUF}}/q_{\mathsf{H}}$, i.e., $\varepsilon_{\mathrm{BDH}} \geq \varepsilon_{\mathrm{tVUF}}/q_{\mathsf{H}}$.

5.3 Weighted VUF Security

The *correctness* and *uniqueness* properties of the weighted VUF follow from arguments similar to the uniqueness property of the threshold VUF. Also, as in the threshold setting, the unforgeability of the weighted VUF scheme is proved based on the computational Bilinear Diffie-Hellman (BDH) assumption. The proof is in the random oracle model, where the hash function $\mathsf{H}_{\hat{\mathbb{G}}}$ is modeled as a random oracle.

Theorem 4 (Weighted VUF security). *Assuming the hardness of bilinear Diffie-Hellman (BDH), the weighted VUF protocol of Figure 3 is existentially unforgeable as per Definition 6 applied to the weighted setting as described in Section 3.1.*

Proof. Let $\mathcal{A}$ be the adversary which corrupts signers with combined weight of up to w and forges a VUF output with probability $\varepsilon_{\mathrm{wVUF}}$. Then, in Figure 6, we illustrate how $\mathcal{A}_{\mathrm{BDH}}$ can use $\mathcal{A}$ to break the BDH assumption. Again, given this simulation and using an argument similar to the proof of Theorem 2, $\mathcal{A}_{\mathrm{BDH}}$ breaks BDH assumption with probability at least $\varepsilon_{\mathrm{wVUF}}/q_{\mathsf{H}}$.

6 Weighted PVSS for Group Elements

The weighted VUF we described in §4 assumed that all signing keys of the signers generated by a trusted key generation functionality. Since our end goal is a weighted VUF in a decentralized setting, such centralized key generation is undesirable. In §8, we will illustrate distributed key generation protocol for the VUF. Note that for weighted VUF, the DKG needs to be weighted as well. A crucial component of our weighted DKG is a non-interactive aggregatable weighted publicly verifiable secret sharing (PVSS) scheme.

This section describes a new weighted PVSS scheme. As a building block, and in order to illustrate our main ideas, we first describe the scheme assuming participants are unweighted. The PVSS scheme is non-interactive, and enables

Input: $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c) \in \mathbb{G}^3 \times \hat{\mathbb{G}}^4$.

Setup and Key Generation:

1. Set $h = g^b$. Send $(h, \hat{g})$ to $\mathcal{A}$.
2. Let $\mathcal{M} \subset [n]$ be the set of malicious participants with combined weight $\leq w$. Let $\mathcal{H} = [n] \setminus \mathcal{M}$ be the set of honest participants.
3. Let $\mathsf{pk} = \hat{g}^a$. This implies that $\mathsf{sk} = h^a = g^{ab}$. Note that $\mathcal{A}_{\mathrm{BDH}}$ cannot explicitly compute sk.
4. For each malicious participant $i \in \mathcal{M}$ with weight w_i, sample its secret keys $a_{i,j} \leftarrow\$\ \mathbb{F}$ for each $j \in w_i$. Let $a(\cdot) \in \mathbb{F}[x]^w$ be a degree w polynomial such that $a(0) = a$ and $a(s_i + j) = a_{i,j}$. (If $\sum_{i \in \mathcal{M}} w_i = w$ then this polynomial is uniquely defined. Otherwise, choose at random $w - \sum_{i \in \mathcal{M}} w_i$ additional values $a(j)$ for values $j > W$ in order to define the polynomial.) Compute $\hat{g}^{a_{i,j}}$ for each $i \in [n]$ and each $j \in w_i$ by interpolating in the exponent.
5. Send $\mathsf{pk} = \hat{g}^a, \{\mathsf{pk}_i = \{\hat{g}^{a_{i,j}}\}_{j \in w_i}\}_{i \in [n]}, \{h^{a_{i,j}}\}_{i \in \mathcal{M}, j \in w_i}$ to $\mathcal{A}$.
6. For each honest participant $i \in \mathcal{H}$, sample $u_i \leftarrow\$\ \mathbb{F}$. It holds that $u_i = b \cdot r_i$ for some $r_i \in \mathbb{F}$ unknown to $\mathcal{A}_{\mathrm{BDH}}$. Compute $\pi_i = g^{u_i} = h^{r_i}$. Then, compute $g^{a_{i,j}}$ for each $j \in w_i$ using interpolation in the exponent, and finally compute $\mathsf{rk}_{i,j} = g^{a_{i,j} u_i}$. $g^{a_{i,j} b r_i} = h^{a_{i,j} r_i}$.
7. Send $\{\mathsf{apk}_i = (\pi_i, \{\mathsf{rk}_{i,j}\}_{j \in w_i})\}_{i \in \mathcal{H}}$ to $\mathcal{A}$.
8. Receive valid apk_i for each $i \in \mathcal{M}$ from $\mathcal{A}$.

Simulating signing and random oracle queries.

1. Let q_{H} be a upper bound on the number of random oracle queries that $\mathcal{A}$ makes.
2. Sample a random index $k' \leftarrow\$\ [q_{\mathsf{H}}]$.
3. On the k-th random oracle query on a message m_k, if $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) \neq \perp$, return $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. Otherwise, if $k \neq k'$, then sample $x_k \leftarrow\$\ \mathbb{F}$, set $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^{x_k}$ and store (m_k, x_k), and return $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. When $k = k'$, return $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^c$.
4. On k-th partial signature query (i, m_k) for signer $i \in \mathcal{H}$, if $m_k \neq m_{k'}$, output $\sigma = (\hat{g}^b)^{x_k/u_i} = \mathsf{H}_{\hat{\mathbb{G}}}(m_k)^{b/u_i} = \mathsf{H}_{\hat{\mathbb{G}}}(m_k)^{1/r_i}$. Otherwise, abort.

Let ρ be $\mathcal{A}$'s VUF output for the message $m_{k'}$, then $\mathcal{A}_{\mathrm{BDH}}$ outputs ρ as the BDH output.

Fig. 6: Weighted VUF security reduction, where we highlight the difference compared to Figure 5 in gray.

a dealer to share a secret $s \in \hat{\mathbb{G}}$ (rather than a secret in $\mathbb{F}$). To verify the PVSS transcript, a verifier needs to perform only 3 pairings and $O(1)$ n-wide multi-exponentiations. Furthermore, the PVSS transcript is aggregatable, meaning that it is possible to aggregate the transcripts of multiple dealers into a single transcript that shares a secret that is the multiplication of the secrets shared by all dealers. The aggregated transcript is also publicly verifiable. The only added verification cost is that of verifying one additional proof-of-knowledge per PVSS transcript that was aggregated into the final transcript.

In§6.4 we describe how to change the unweighted PVSS to a weighted one, by having each participant register as many public keys as its weight. This approach is not always possible. For example, the exact weight of a participant might be constantly changing and might only become known shortly before the PVSS is run (for example, shortly before an epoch change in a blockchain). This makes it hard to run the PVSS, since all participants must first reach a consensus about the public keys of each participant. Also, there is a considerable difference in the implementation complexity between assigning a single key per participant and assigning to it a changing number of keys, in particular when modifying an existing system which already uses a single key per participant. To overcome this issue, we present in §6.5 a weighted PVSS protocol where each participant has a single key, independently of its weight. The overhead of this protocol is slightly higher, and its security proof is different than that of the protocol for the unweighted setting (see §7).

6.1 PVSS Definitions

We define PVSS for the unweighted threshold setting. A secret sharing scheme lets a dealer D share a secret s among a set of n participants with an (n, t) threshold access structure. We focus on non-interactive PVSS. Informally, the main property we seek from a PVSS is that any subset of t or fewer shares does not reveal any information (other than some public commitment) about the secret s, but any subset of $t+1$ or more shares can uniquely determine the secret s. Additionally, any external verifier $\mathcal{V}$ should be able to check that the dealer D has correctly shared a secret among n parties without learning any information about the shares or the secret, hence the name *publicly verifiable*.

Definition 7 (Non-interactive PVSS). *An (n,t) non-interactive PVSS is a tuple of polynomial-time computable algorithms (*KeyGen, Share, Verify, Recon*) as defined below. We will assume that all these algorithms implicitly take as input public parameters* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa)$*, generated using appropriate* Setup *algorithms.*

– $\mathsf{KeyGen} \rightarrow \{\mathsf{ek}_i, \mathsf{dk}_i\}_{i\in[n]}$. KeyGen *is a non-interactive algorithm, where each participant i publishes an encryption key* ek_i *and keeps the corresponding decryption key* dk_i *private.*
– $\mathsf{Share}(n, t, \{\mathsf{ek}_i\}_{i\in[n]}, s) \rightarrow (\mathsf{com}, \{c_i, \pi_i\}_{i\in[n]})$. *The dealer D non-interactively generates shares $s_1, \ldots, s_n$ for a randomly chosen secret s. It encrypts each share s_i with the encryption key* ek_i *of signer i to obtain the ciphertext c_i, along with proofs π_i that the c_i values are encryptions of valid shares of the same secret. The dealer additionally computes a commitment* com *to the secret shares.*
– $\mathsf{Verify}(\mathsf{com}, \{c_i, \pi_i\}_{i\in[n]}) \rightarrow 0/1$. *On input a PVSS transcript, anyone can non-interactively use* Verify *to validate that* com *is a commitment to valid shares of some secret s and the c_i values consist of encryptions of valid shares of the same secret.*
– $\mathsf{Recon}(S, \mathsf{com}, \{c_i, \mathsf{dk}_i\}_{i\in S}) \rightarrow s/\perp$. *In this step, each participant $i \in S$ decrypts c_i using its secret key* dk_i *to get its share $\tilde{s}_i$. It publishes $\tilde{s}_i$ together with a NIZK proof $\tilde{\pi}_i$ that $\tilde{s}_i$ is a correct decryption of c_i. Anyone with* $(\mathsf{com}, \{\mathsf{ek}_i, \pi_i\})$ *can validate tese proofs. Lastly, $t+1$ or more valid decrypted shares can be deterministically combined to recover the original secret s shared by the dealer. If the number of valid decrypted shares is $\leq t$,* Recon *outputs $\perp$.*

A PVSS scheme must ensure *correctness, verifiability*, and *secrecy*. We define these properties next. Informally, *correctness* means that if the dealer is honest all checks succeed and the secret can be reconstructed. *Verifiability* means that any attempts of the dealer to cheat in dealing the shares, and any attempts of the participants to use the wrong shares, are detected (except with negligible probability). *Secrecy* means that the view of any t corrupt participants can be simulated by a simulator which only sees a commitment to the shared secret.[5] In addition, we require the PVSS scheme to be *aggregatable*, meaning that it is possible to aggregate the PVSS transcripts of multiple dealers to a single transcript whose length is of the same order as that of a single transcript.

Definition 8 (Correctness). *A PVSS scheme is correct if for all $\lambda \in \mathbb{N}$, all allowable thresholds $1 \leq t+1 \leq n$, all S such that $t+1 \leq |S| \leq n$, all secret $s \in \mathbb{F}$, for all subsets S of participants with $|S| \geq t+1$, and for* $\{\mathsf{ek}_i, \mathsf{dk}_i\}_{i\in[n]} \leftarrow \mathsf{KeyGen}()$*, the following properties hold:*

– $\Pr[\mathsf{Verify}(\mathsf{Share}(n, t\{\mathsf{ek}_i\}, s), \{\mathsf{ek}_i\}_{i\in[n]}) = 1] = 1$
– $\Pr[\mathsf{Recon}(S, \mathsf{com}, \{c_i, \mathsf{dk}_i\}_{i\in S}) = s : (\mathsf{com}, \{c_i, \pi_i\}_{i\in[n]}) \leftarrow \mathsf{Share}(n, t, \{\mathsf{ek}_i\}_{i\in[n]}, s)] = 1$

Here, the probability is over the choices of randomness of the KeyGen *and the* Share *algorithm.*

Definition 9 (Verifiability). *If* Verify *accepts a PVSS transcript* $(\mathsf{com}, \{c_i, \mathsf{dk}_i\}_{i\in S}$*, then, with overwhelming probability, the c_i's are encryptions of valid shares of some secret. If the check in the reconstruction step passes, then the communicated shares $\tilde{s}_i$ are the shares created by the dealer.*

We define the secrecy property in terms of simulatability: We would like to require that for every probabilistic polynomial-time adversary $\mathcal{A}$ that corrupts up to t signers, there exists a PPT simulator, such that on input of a commitment com to a uniformly random secret s, the simulator produces a view which is indistinguishable from $\mathcal{A}$'s view of a honestly generated PVSS transcript with s as the secret.

[5] We use this definition for secrecy since the shared secert is used as a key for a VUF, and the verifiability property of the VUF enables to verify that VUF outputs were computed using the right key. This means that we cannot require that the output of the PVSS is indistinguishable from a uniformly random value, since given a potential value for the private key of the VUF it is possible to verify whether it is correct.

While this definition of secercy is somewhat informal, we next provide a concrete security definition (Definition 10), which uses the parameters and commitment used in the PVSS protocol that we presented in this work. We note that there are alternative ways to define the secrecy property for a PVSS scheme (see, for example, [CD17,DXT$^+$23]). We define secrecy as above based on how the secrecy property affects the overall security of the threshold VUF scheme. In particular, the verifiability property of the VUF requires that it is possible to verify that VUF outputs were computed using the right key. That key is shared by the PVSS. This means that we cannot require that the output of the PVSS is indistinguishable from a uniformly random value, since given a potential value for the private key of the VUF it is possible to verify whether it is correct. Instead, we require that an adversary corrupting up to t participants sees a view which cannot be distinguished from a view which is generated by a simulator that is only given a commitment to the value shared by the PVSS.

Game 5 (PVSS Secrecy) *Let $\kappa \in \mathbb{N}$ be a security parameter. Let $\mathcal{A}$ be the adversary. Let $\mathcal{C}$ be the challenger.*

- **Setup.** *Adversary $\mathcal{A}$ specifies (n, t) with $t < n$ to a challenger $\mathcal{C}$. Let $\mathcal{M} \subset [n]$ with $|\mathcal{M}| \leq t$ be the set of malicious participants. $\mathcal{A}$ sends $\mathcal{M}$ to $\mathcal{C}$. Let $\mathcal{H} = [n] \setminus \mathcal{M}$ be the set of honest participants.*
- *$\mathcal{C}$ samples a uniformly random bit $b \in \{0, 1\}$. Depending upon the bit, $\mathcal{C}$ interacts with $\mathcal{A}$ as follows:*
  - *If $b = 0$, $\mathcal{C}$ generates an honest PVSS transcript as follows:*
    - *Sample uniformly random generators $(g, h, \hat{g}) \in \mathbb{G}^2 \times \hat{\mathbb{G}}$ and send them to $\mathcal{A}$.*
    - *For each honest participant $i \in \mathcal{H}$, sample $\mathsf{dk}_i \leftarrow\$ \mathbb{F}$ and let $\mathsf{ek}_i = g^{\mathsf{dk}_i}$. For each of the corresponding encryption keys ek_i, honestly generate its proof-of-knowledge π_i. Send $\{\mathsf{ek}_i, \pi_i\}_{i \in \mathcal{H}}$ to $\mathcal{A}$.*
    - *$\mathcal{A}$ responds with $\{\mathsf{ek}_j, \pi_j\}_{j \in \mathcal{M}}$.*
    - *Let $\mathbf{ek} = [\mathsf{ek}_i]_{i \in [n]}$ be the vector of encryption keys of all participants. Send $\mathsf{PVSS.Deal}_{\mathsf{pp}}(\mathbf{ek}, a)$ to $\mathcal{A}$.*
  - *If $b = 1$, $\mathcal{C}$ runs a simulator which interacts with $\mathcal{A}$ on behalf of $\mathcal{S}$. That simulator generates a PVSS transcript given a commitment to the shared secret but without knowing the shared secret itself (for the unweighted threshold PVSS construction, this is the simulator $\mathcal{S}$ from Figure 10 in §7).*

Definition 10 (PVSS Secrecy). *For any adversary $\mathcal{A}$, let $\mathsf{View}_{\mathsf{real}}$ and $\mathsf{View}_{\mathsf{sim}}$ be the view of $\mathcal{A}$ in Game 5 with $b = 0$ and $b = 1$, respectively. A PVSS scheme is secure, if for any adversary PPT adversary $\mathcal{A}$ that corrupts up to t participants, $\mathsf{View}_{\mathsf{real}}$ is computationally indistinguishable from $\mathsf{View}_{\mathsf{sim}}$.*

Secret reconstruction. While we define and describe the secret reconstruction algorithm of the PVSS, it is only described for completeness. In our target scenario the PVSS is used as part of a DKG to generate shared keys for a VUF. The shared key is never reconstructed.

6.2 Public-Key Registration

All participants the in the PVSS, including participants that only receive shares, must provide a proof-of-knowledge (PoK) of the private key corresponding to their public key.

The need for a PoK. The attached PoK prevents the adversary from launching rogue-key attacks [RY07,BDN18]. If PoK's are not present, participants could set their public keys as a function of the public keys of other participants. This enables even a single participant to set its key so that it can reconstruct the secret all by itself! As a simple example, suppose that the PVSS protocol described in Figure 8 implements a 2-out-of-n secret sharing with a linear polynomial $p(x) = bx + a$. The two first shares are $h^{p(1)} = h^{b+a}$; $\quad h^{p(2)} = h^{2b+a}$. The secret is $h^a = h^{2p(1)-p(2)}$. If Participant 2 is not required to provide a PoK of its private key, it can set its public key to be $\mathsf{ek}_2 = {\mathsf{ek}_1}^2$. According to the protocol, the dealer sends $C_1 = h^{p(1)}{\mathsf{ek}_1}^r$ and $C_2 = h^{p(2)}{\mathsf{ek}_2}^r = h^{p(2)}{\mathsf{ek}_1}^{2r}$. Participant 2 can then decrypt the secret all by itself by computing $C_1^2/C_2 = h^{2p(1)-p(2)}\mathsf{ek}^{2r-2r} = h^a$. Similar attacks are possible against any threshold, and enable even a single participant that sets its public key as a function of the public keys of other participants, to reconstruct the secret.

Looking ahead, we will crucially use the PoK to prove the security of our PVSS scheme. Concretely, we use the non-interactive variant of the Schnorr identification scheme as the PoK [Sch90]. Each participant must provide this proof once, when it joins the system (and also whenever it changes its public key). We present the PoK's in Figure 7, separately from the PVSS protocol.

```
$\underline{\mathsf{PVSS.PKsetup}(i)} \to (\mathsf{dk}_i, (\mathsf{ek}_i, \mathsf{pok}_i))$
// Participant $i$ generating and proving its key
$\mathsf{dk}_i \leftarrow_\$ \mathbb{F}$;   $\mathsf{ek}_i := g^{\mathsf{dk}_i}$
return $(\mathsf{dk}_i, (\mathsf{ek}_i, \mathsf{PoK.Dlog}(g, \mathsf{ek}_i, \mathsf{dk}_i)))$

$\underline{\mathsf{PVSS.PKver}(\mathsf{ek}_i, \mathsf{pok}_i)} \to \{0, 1\}$
// Verifying the key of participant $i$
$(\cdot, y, \cdot, \cdot) := \mathsf{pok}_i$
assert $y = \mathsf{ek}_i$ // verify that proof is for $\mathsf{ek}_i$
assert $\mathsf{PoK.DlogVer}(\mathsf{pok}_i)$

$\underline{\mathsf{PoK.Dlog}(g, y, \alpha)} \to \mathsf{pok}$
// Schnorr proof of knowledge of $\alpha = \log_g y$
$r \leftarrow_\$ \mathbb{F}$;   $u := g^r$;   $c := \mathsf{H}(g, y, u) \in \mathbb{F}$
$z := r + c \cdot \alpha$
return $(g, y, u, z)$

$\underline{\mathsf{PoK.DlogVer}(\mathsf{pok})} \to \{0, 1\}$
$(g, y, u, z) := \mathsf{pok}$;   $c := \mathsf{H}(g, y, u)$
assert $g^z = u \cdot y^c$
```

Fig. 7: Key generation by participants, and verification that each participant knows its private key. The proof of knowledge of the private key must be provided by each participant when it provides a new public key, and must be verified by all other participants. Here, we model the hash function H as a random oracle.

6.3 Unweighted Threshold PVSS

A downside of existing aggregatable PVSS schemes such as SCRAPE [CD17] is that they require using $2n$ pairings to verify the correctness of the aggregated transcript. (Here n is the total number of shares, which in our *weighted* PVSS is denoted by W.) This is a considerable overhead, since we expect n to be in the order of thousands.

We describe a new PVSS protocol where the verification of the aggregated transcript requires only three pairings, two n-wide multi-exponentiations in $\mathbb{G}$, and one in $\hat{\mathbb{G}}$. Recall from §2.2 that computing an n-wide multi-exponentiation is $O(\log n)$ faster than computing n exponentiations, which is faster than computing n pairings. The total gain in performance can be two orders of magnitude.

Our PVSS protocol combines ideas from SCRAPE PVSS protocol [CD17] for group elements and Groth's PVSS for field elements. Note that although Groth's PVSS [Gro21] is for field elements, our PVSS scheme is for secret-sharing group elements. We summarize the PVSS scheme in Figure 8, and describe its details next.

Setup and Notations. Let g, h be two uniformly random and independent generators of $\mathbb{G}$. Also, let $\hat{g}$ be a random generator of $\hat{\mathbb{G}}$. Let $(\mathsf{dk}_i, \mathsf{ek}_i := g^{\mathsf{dk}_i})$ be the secret and public keys of participant i. Let $\mathbf{ek} := [\mathsf{ek}_1, \ldots, \mathsf{ek}_n]$ be the vector of the public keys of all participants. We use ElGamal encryption to encrypt the shares.

Transcript generation ($\mathsf{PVSS.Deal}$)**.** Let $p(x)$ be a random degree t polynomial that the dealer wants to deal. The share of participant i is $h^{p(i)}$. The dealer first computes two commitment vectors:

$$\boldsymbol{V} := [V_0, V_1, V_2, \ldots, V_n] := [g^{p(0)}, g^{p(1)}, g^{p(2)}, \ldots, g^{p(n)}]$$
$$\hat{\boldsymbol{V}} := [\hat{V}_0, \hat{V}_1, \hat{V}_2, \ldots, V_n] := [\hat{g}^{p(0)}, \hat{g}^{p(1)}, \hat{g}^{p(2)}, \ldots, \hat{g}^{p(n)}]$$

Additionally, the dealer adds a standard PoK of the discrete logarithm of $\hat{V}_0$ to the base $\hat{g}$, i.e., a proof-of-knowledge of the shared secret. This added proof prevents malicious participants from sharing secrets which depend on the secrets shared by other participants (e.g., canceling previous secrets). This PoK is only for the shared secret and not for all evaluations of the polynomial.

The dealer then encrypts all shares using the ElGamal encryption scheme with a randomness $r \leftarrow_\$ \mathbb{F}$ that is used for all encryptions. Let $\boldsymbol{C}$ be the resulting ciphertext, then

$$\begin{aligned}\boldsymbol{C} &:= [C_0, C_1, C_2, \ldots, C_n] := [g^r, h^{p(1)}\mathsf{ek}_1^r, h^{p(2)}\mathsf{ek}_2^r, \ldots, h^{p(n)}\mathsf{ek}_n^r] \\ &= [g^r, h^{p(1)}g^{\mathsf{dk}_1 \cdot r}, h^{p(2)}g^{\mathsf{dk}_2 \cdot r}, \ldots, h^{p(n)}g^{\mathsf{dk}_n \cdot r}]\end{aligned}$$

The PVSS transcript is $(\hat{R} = \hat{g}^r, \mathbf{pok}, \boldsymbol{V}, \hat{\boldsymbol{V}}, \boldsymbol{C})$. The PoK in the transcript is of the shared PVSS secret.

Transcript verification. The PVSS transcript verifier $\mathcal{V}$ first validates the PoKs included in $\mathbf{pok}$. For a non-aggregated transcript, $\mathbf{pok}$ contains exactly one proof. Then, $\mathcal{V}$ checks that $\boldsymbol{V}$ commits to evaluations of a polynomial of degree

```
$\underline{\mathsf{PVSS.Deal}_{\mathsf{pp}}(\mathbf{ek}, a_0)} \to \mathsf{trx}$
// Sample a degree $t$ random polynomial
// and commit to its evaluations
$p(X) := \sum_{i=0}^{t} a_i X^i$, where $(a_1, \ldots, a_t) \leftarrow\$ \mathbb{F}^t$
$\hat{V}_0, \hat{V}_1, \ldots, \hat{V}_n := \hat{g}^{p(0)}, \hat{g}^{p(1)}, \ldots, \hat{g}^{p(n)}$
$V_0, V_1, \ldots, V_n := g^{p(0)}, g^{p(1)}, \ldots, g^{p(n)}$
// PoK of the secret, using protocol from Fig. 7
$\mathsf{pok} = \mathsf{PoK.Dlog}(\hat{V}_0, \hat{g}, p(0))$
$\mathbf{pok} := \mathsf{pok}$    // Store as a vector of length 1
$r \leftarrow\$ \mathbb{F}$    $\hat{R} := \hat{g}^r$
$C_0, C_1, \ldots, C_n := g^r, h^{p(1)}\mathsf{ek}_1^r, \ldots, h^{p(n)}\mathsf{ek}_n^r$
return $(\hat{R}, \mathbf{pok}, \boldsymbol{V}, \hat{\boldsymbol{V}}, \boldsymbol{C})$.

$\underline{\mathsf{PVSS.Verify}_{\mathsf{pp}}(\mathbf{ek}, \mathsf{trx})} \to \{0, 1\}$
$(\hat{R}, \mathbf{pok}, \boldsymbol{V}, \hat{\boldsymbol{V}}, \boldsymbol{C}) := \mathsf{trx}$
assert $\mathsf{SCRAPE.LowDegreeTest}(\boldsymbol{V}, t, n) = 1$
assert $e(C_0, \hat{g}) = e(g, \hat{R})$
$\forall \mathsf{pok}_i \in \mathbf{pok}$, assert $\mathsf{PoK.DlogVer}(\mathsf{pok}_i)$
// $V_0^{(j)}$ refers to $V_0$ of $\mathsf{pok}_j \in \mathbf{pok}$
assert $V_0 = \prod_{j \in \mathbf{pok}} V_0^{(j)}$
$\forall i \in [0, n]$,   assert $e(g, \hat{V}_i) = e(V_i, \hat{g})$
$\forall i \in [n]$,   assert $e(h, \hat{V}_i) \cdot e(\mathsf{ek}_i, \hat{R}) = e(C_i, \hat{g})$
// See the paragraph "Optimizing transcript verification"
// on how to batch verify with only 3 pairings

$\underline{\mathsf{PVSS.Aggregate}(\mathsf{trx}_1, \mathsf{trx}_2)} \to \mathsf{trx}$
$(\hat{R}_1, \mathbf{pok}_1, \boldsymbol{V}_1, \hat{\boldsymbol{V}}_1, \boldsymbol{C}_1) := \mathsf{trx}_1$
$(\hat{R}_2, \mathbf{pok}_2, \boldsymbol{V}_2, \hat{\boldsymbol{V}}_2, \boldsymbol{C}_2) := \mathsf{trx}_2$
$\hat{R} := \hat{R}_1 \cdot \hat{R}_2$
$\mathbf{pok} := \mathbf{pok}_1 || \mathbf{pok}_2$    // concatenation
$\boldsymbol{V} := \boldsymbol{V}_1 \cdot \boldsymbol{V}_2$, $\hat{\boldsymbol{V}} := \hat{\boldsymbol{V}}_1 \cdot \hat{\boldsymbol{V}}_2$, and $\boldsymbol{C} := \boldsymbol{C}_1 \cdot \boldsymbol{C}_2$,
return $(\hat{R}, \mathbf{pok}, \boldsymbol{V}, \hat{\boldsymbol{V}}, \boldsymbol{C})$

$\underline{\mathsf{PVSS.DecryptShare}_{\mathsf{pp}}(\mathsf{trx}, i, \mathsf{dk}_i)} \to (\mathsf{sk}_i, V_i, \mathsf{pk})$
$\left(\cdot, \cdot, \hat{\boldsymbol{V}}, \boldsymbol{C}\right) := \mathsf{trx}$
$\mathsf{sk}_i := C_i / (C_0)^{\mathsf{dk}_i}$
$\mathsf{pk} := \hat{V}_0$
return $(\mathsf{sk}_i, \hat{V}_i, \mathsf{pk})$

$\underline{\mathsf{PVSS.Reconstruct}_{\mathsf{pp}}(S, (\mathsf{sk}_i, \hat{V}_i)_{i \in S})} \to \mathsf{sk}$
// find subset $Q$ of $\geq t + 1$ valid shares
assert $\exists Q \subseteq S, |Q| \geq t + 1$ such that:
    $e(\mathsf{sk}_i, \hat{g}) = e(h, \hat{V}_i)$,   $\forall i \in Q$
// Compute Lagrange coeff. $\ell_{Q,i}(0) = \prod_{j \in Q, j \neq i} \frac{0-j}{i-j}$
$\mathsf{sk} \leftarrow \prod_{i \in Q} \mathsf{sk}_i^{\ell_{Q,i}(0)}$
return $\mathsf{sk}$    // $= h^{p(0)}$

$\underline{\mathsf{SCRAPE.LowDegreeTest}(\boldsymbol{V}, t, n)} \in \{0, 1\}$
// Random degree $d = n - t$ polynomial
assert $n = |\boldsymbol{V}| - 1$
$d := n - t$
$f(X) := \sum_{i=0}^{d} f_i X^i$, where $(f_0, \ldots, f_d) \leftarrow\$ \mathbb{F}^d$
$f(1), \ldots, f(n) := \mathsf{FFT}_{\mathbb{F}}(f)$
$\ell' := 1 / \prod_{j \in [1,n]} (0 - j)$
for all $i \in [1, n]$:
    $\ell_i := 1 / \left(i \cdot \prod_{j \neq i, j \in [1,n]} (i - j)\right)$
assert $V_0^{\ell' f(0)} \prod_{i \in [0,n)} (V_i)^{\ell_i \cdot f(i)} = 1_{\hat{\mathbb{G}}}$
```

Fig. 8: A threshold PVSS scheme, including the dealing, verification and aggregation algorithms. The dealt secret is $\mathsf{sk} = h^{p(0)} \in \mathbb{G}$, and the corresponding public commitment $\mathsf{pk} = \hat{g}^a \in \hat{\mathbb{G}}$. $\mathsf{pp} = (g, h, \hat{g}) \in \mathbb{G}^2 \times \hat{\mathbb{G}}$ are the public parameters of the scheme. For any $i \in [n]$ $\mathsf{ek}_i = g^{\mathsf{dk}_i} \in \mathbb{G}$ is the i-th encryption key, with a corresponding decryption key dk_i.

at most t using the low-degree test from [CD17]. (This test can also be applied to $\hat{\boldsymbol{V}}$, but it is more efficient to run it in $\mathbb{G}$ rather than in $\hat{\mathbb{G}}$.) The low-degree test requires one FFT and one n-wide multi-exponentiation.

Next, $\mathcal{V}$ checks that the encryptions are valid, i.e., that C_i encrypts the share of participant i, and also that $\boldsymbol{V}$ and $\hat{\boldsymbol{V}}$ encode the same values. Later on we describe an optimized protocol in which $\mathcal{V}$ checks all ciphertexts using a batched protocol that uses only four pairings and four n-wide multi-exponentiations. However, to illustrate the idea, we first describe the simple approach where $\mathcal{V}$ uses 3 pairing to validate each single ciphertext in $\boldsymbol{C}$. $\mathcal{V}$ checks that:

1. C_0 is well formed, i.e., the exponents of C_0 and $\hat{R}$ are the same. Namely $e(C_0, \hat{g}) = e(g, \hat{R}) \Leftrightarrow e(g^r, \hat{g}) = e(g, \hat{g}^r)$.
2. For each $i \in [1, n]$:

$$\begin{aligned} e(h, \hat{V}_i) \cdot e(\mathsf{ek}_i, \hat{R}) &= e(C_i, \hat{g}) \Leftrightarrow \\ e(h, \hat{g}^{p(i)}) \cdot e(\mathsf{ek}_i, \hat{g}^r) &= e(h^{p(i)}\mathsf{ek}_i^r, \hat{g}) \Leftrightarrow \\ e(h^{p(i)}, \hat{g}) \cdot e(\mathsf{ek}_i^r, \hat{g}) &= e(h^{p(i)}, \hat{g}) e(\mathsf{ek}_i^r, \hat{g}) \end{aligned}$$

3. For each $i \in [0, n]$, $e(g, \hat{V}_i) = e(V_i, \hat{g})$.

The value $\hat{V}_0 = \hat{g}^{p(0)}$ is the commitment to the secret itself, and is never encrypted. So there is nothing to verify there. (Instead, $\hat{V}_0$ is verified as part of the low degree test.)

On using $\varSigma$-protocols. It is possible to verify the PVSS transcript using the Chaum-Pedersen discrete logarithm equality test (as in [CP92]) instead of using a bilinear pairing. The advantage is a faster verification time as a result of not using pairings. We do not use this approach is as it is based on a proof-of-knowledge that is generated by the dealer who knows the shared secret. On the other hand, an aggregated transcript of multiple dealers shares a secret that no one knows, and therefore a single proof-of-knowledge cannot be generated. Our end goal is to verify an *aggregated* transcript and therefore we use a pairing-based verification.

Optimizing transcript verification. The PVSS transcript verification procedure we describe so far requires $5n + 2$ pairings. This cost can be reduced to four pairings and four n-wide multi-exponentiations, using the standard random linear combination technique to verify all n ciphertexts and commitments at once. To do so, $\mathcal{V}$ samples $2n+1$ uniform random field elements $\rho_1, \ldots, \rho_n, \rho'_0, \rho'_1, \ldots, \rho'_n \leftarrow\$ \mathbb{F}^{2n+1}$, and then checks that:

$$e(h, \prod_{i\in[n]} \hat{V}_i^{\rho_i}) \cdot e(g \prod_{i\in[n]} \mathsf{ek}_i^{\rho_i}, \hat{R}) \cdot e(g, \prod_{i\in[0,n]} \hat{V}_i^{\rho'_i}) = e(C_0 \prod_{i\in[n]} C_i^{\rho_i} \prod_{i\in[0,n]} V_i^{\rho'_i}, \hat{g})$$

Intuitively, in this equation the verifier checks that a random linear combination of the ciphertexts is valid with respect to the same random linear combination of the commitments and of the encryption keys, and that that $e(C_0, \hat{g}) = e(g, \hat{R})$. The randomized check ensures that if there exists an i such that $e(h, \hat{V}_i) \cdot e(\mathsf{ek}_i, \hat{R}) \neq e(C_i, \hat{g})$ or $e(g, \hat{V}_i) \neq e(V_i, \hat{g})$, then the check in the equation fails with probability at least $1 - 1/|\mathbb{F}|$.

Verifying the aggregated transcript. The aggregated transcript, as well, can be verified using these exact same checks, along with checking correctness of each $\mathsf{pok} \in \mathbf{pok}$.

On using the commitments. Note that the commitments $V_0, \ldots, V_n$ are not directly used in the PVSS protocol. They are included since they are required for the proof of security of the VUF protocol that uses the PVSS scheme for generating its keys.

6.4 Weighted PVSS using Virtualization

A straightforward transformation of the unweighted PVSS scheme into a setting where parties are weighted is the folklore virtualization approach, which assigns a virtual party, with its own public key, for each share. We describe it in next.

The weighted PVSS scheme treats each party i with weight w_i as w_i virtual parties. Let $W := \sum_{i\in[n]} w_i$ be the total weight, and let $w + 1$ with $1 \leq w < W$ be the threshold. Then, the dealer shares its secret using the unweighted PVSS scheme with (W, w)-threshold secret sharing. Each party then receives a number of shares that is equal to its weight. More precisely, let $s_i := \sum_{j=1}^{i-1} w_j$ be the sum of the weights of the first $i - 1$ parties. Then, party i receives the shares whose indexes are in the range $(s_i, s_{i+1}]$.

In this transformation to weighted PVSS, each party i with weight w_i must use w_i independent encryption keys, i.e.,

$$\mathsf{dk}_i := \{\mathsf{dk}_{i,1} \ldots, \mathsf{dk}_{i,w_i}\} \leftarrow\$ \mathbb{F}^{w_i}; \text{ and } \mathsf{ek}_i := \{\mathsf{ek}_{i,1}, \ldots, \mathsf{ek}_{i,w_i}\}.$$

Party i publishes $\{\mathsf{ek}_{i,1}, \ldots, \mathsf{ek}_{i,w_i}\}$ and proves knowledge of the corresponding private keys. (It is possible to use a single proof of knowledge for all keys, by using batching and proving knowledge of the discrete log of $\prod_{j\in[w_i]} \mathsf{ek}_{i,j}^{r_j}$, with random exponents $r_1, \ldots, r_{w_i}$.)

Security and performance analysis. Since each party i uses w_i independent encryption keys, the scheme is identical to running a (W, w) unweighted scheme. Hence, the security of this weighted PVSS scheme follows directly from the security of the unweighted PVSS scheme.

In terms of performance, the PVSS sharing and verification costs are proportional to the total weight W. Additionally, each party i needs to decrypt w_i ciphertexts. The number of pairings needed for verification is constant, using the optimizations listed for the unweighted case. This overhead is reasonable since the PVSS is run rather infrequently, e.g. at the beginning of each epoch of the blockchain, whereas the VUF is running much more frequently, say in each block produced by the blockchain.

Remark. The virtualization approach for weighted secret sharing requires each party to to assign a public key for each unit of its weight. This requirement could be challenging, as the weight of a validator may vary over different epochs, and since all validators must reach a consensus about the keys of each validator. With such constraints it might be preferable to use a PVSS where each party has a single public key, regardless of its weight. Such a PVSS is slightly more complicated and is less efficient, and we describe it only in the full version of this paper.

6.5 Weighted PVSS using a Single Public Key per Participant

There might be system requirements that require the usage of a single public key per participant (for example, if the number of keys of a participant varies depending on its weight, it must broadcast these keys and other parties need to reach a consensus on the values of these keys). Figure 9 describes a weighted PVSS protocol in which participants have a single public key, independent of their weight.

We use the following notation in this protocol: The shares are indexed $1, \ldots, W$, where participant j has w_j shares. We use the notation $u(i)$ to denote the index of the participant who should receive the i-th share. (Participant 1 should receive the first w_1 shares and therefore for $i = 1, \ldots, w_1$ it holds that $u(i) = 1$. Participant 2 should receive the next w_2 shares, etc.)

The main difficulty in proving security in this setting is that the security proof uses a simulator which must "cheat" in generating the encryptions of the shares of the honest parties. This requires having $W - w$ degrees of freedom, as the number of shares of honest parties (which is equal to $2W/3$ in the consensus setting). In the unweighted threshold PVSS, the values of the encryptions are set by choosing appropriate values for the public keys of the honest parties, but in this weighted PVSS the number of these keys is smaller than the number of shares. The new protocol solves this problem by using a separate r_i value per share, rather than a single r value for all shares. Setting these values appropriately allows the simulator to simulate the encryptions of the shares of the honest parties.

Optimizations and performance We first discuss the computation overhead. The main overhead is computing the pairings in PVSS.Verify. A naïve implementation requires $7W$ pairings. The main obstacle for improving the overhead using batching is in batching the computation of the many invocations of $e(\mathsf{ek}_{u(i)}, \hat{R}_i)$, since these pairings do not have a common input as their first or second argument. However, all the shares associated with node ℓ use the same public key ek_ℓ (for these nodes $u(i) = \ell$). Therefore it is possible to batch all pairings that use this argument. Batching all these W pairings can therefore be done using n pairings and n multi-exponentiations of widths $w_1, \ldots, w_n$. **The total number of pairings is $n + 3$.**

In terms of communication, the naïve protocol has a transcript of size $5W$ group elements, which is quite large. There are two optimizations that can be applied:

– Each of the vectors $\boldsymbol{V}$ and $\hat{\boldsymbol{V}}$ includes W values of a polynomial of degree w. Instead of sending these values explicitly it is possible to send the coefficients of the polynomial in the exponent, namely $g^{a_0}, \ldots, g^{a_w}$ and $\hat{g}^{a_0}, \ldots, \hat{g}^{a_w}$. *The length of each of these vectors is* $w+1$. The vectors $\boldsymbol{V}, \hat{\boldsymbol{V}}$ can be computed using interpolation in the exponent.

```
$\underline{\mathsf{PVSS.Deal}_{\mathsf{pp}}(\mathbf{ek}, a_0)} \to \mathsf{trx}$
// Sample a degree $w$ random polynomial
$p(X) \leftarrow \sum_{i=0}^{w} a_i X^i$, where $(a_1, \ldots, a_t) \leftarrow_\$ \mathbb{F}^w$
$\hat{V}_0, \hat{V}_1, \ldots, \hat{V}_W \leftarrow \hat{g}^{p(0)}, \hat{g}^{p(1)}, \ldots, \hat{g}^{p(W)}$
$V_0, V_1, \ldots, V_W \leftarrow g^{p(0)}, g^{p(1)}, \ldots, g^{p(W)}$
// PoK of the secret, using protocol from Fig. 7
$\mathsf{pok} = \mathsf{PoK.Dlog}(V_0, \hat{g}, p(0))$
$\mathbf{pok} \leftarrow \mathsf{pok}$    // Store as a vector of length 1
$r_1, \ldots, r_W \leftarrow_\$ \mathbb{F}^W$
$R_1, \ldots, R_W \leftarrow g^{r_1}, \ldots, g^{r_W}$
$\hat{R}_1, \ldots, \hat{R}_W \leftarrow \hat{g}^{r_1}, \ldots, \hat{g}^{r_W}$
// $u(i)$ is the index of the participant receiving share $i$
$C_1, \ldots, C_W \leftarrow h^{p(1)} \mathsf{ek}_{u(1)}^{r_1}, \ldots, h^{p(W)} \mathsf{ek}_{u(W)}^{r_W}$
return $(\boldsymbol{R}, \hat{\boldsymbol{R}}, \mathbf{pok}, \boldsymbol{V}, \hat{\boldsymbol{V}}, \boldsymbol{C})$.

$\underline{\mathsf{PVSS.Verify}_{\mathsf{pp}}(\mathbf{ek}, \mathsf{trx})} \to \{0,1\}$
$(\boldsymbol{R}, \hat{\boldsymbol{R}}, \mathbf{pok}, \boldsymbol{V}, \hat{\boldsymbol{V}}, \boldsymbol{C}) \leftarrow \mathsf{trx}$
assert $\mathsf{SCRAPE.LowDegreeTest}(\boldsymbol{V}, w, W) = 1$
$\forall \mathsf{pok}_i \in \mathbf{pok}$, assert $\mathsf{PoK.DlogVer}(\mathsf{pok}_i)$
// $V_0^{(j)}$ refers to $V_0$ of $\mathsf{pok}_j \in \mathbf{pok}$
assert $V_0 = \prod_{j \in \mathbf{pok}} V_0^{(j)}$
$\forall i \in [0, W]$,   assert $e(g, \hat{V}_i) = e(V_i, \hat{g})$
$\forall i \in [W]$,   assert $e(R_i, \hat{g}) = e(g, \hat{R}_i)$
$\forall i \in [W]$,   assert $e(h, \hat{V}_i) \cdot e(\mathsf{ek}_{u(i)}, \hat{R}_i) = e(C_i, \hat{g})$
// The number of pairings in batch verification is $n + 3$,
// where $n$ is the number of recipients

$\underline{\mathsf{PVSS.Aggregate}(\mathsf{trx}_1, \mathsf{trx}_2)} \to \mathsf{trx}$
$(\boldsymbol{R}_1, \hat{\boldsymbol{R}}_1, \mathbf{pok}_1, \boldsymbol{V}_1, \hat{\boldsymbol{V}}_1, \boldsymbol{C}_1) \leftarrow \mathsf{trx}_1$
$(\boldsymbol{R}_2, \hat{\boldsymbol{R}}_2, \mathbf{pok}_2, \boldsymbol{V}_2, \hat{\boldsymbol{V}}_2, \boldsymbol{C}_2) \leftarrow \mathsf{trx}_2$
$\mathbf{pok} \leftarrow \mathbf{pok}_1 || \mathbf{pok}_2$    // concatenation
$\hat{\boldsymbol{R}} \leftarrow \hat{\boldsymbol{R}}_1 \cdot \hat{\boldsymbol{R}}_2$,   $\boldsymbol{R} \leftarrow \boldsymbol{R}_1 \cdot \boldsymbol{R}_2$
$\boldsymbol{V} \leftarrow \boldsymbol{V}_1 \cdot \boldsymbol{V}_2$,   $\hat{\boldsymbol{V}} \leftarrow \hat{\boldsymbol{V}}_1 \cdot \hat{\boldsymbol{V}}_2$,   $\boldsymbol{C} \leftarrow \boldsymbol{C}_1 \cdot \boldsymbol{C}_2$
return $(\boldsymbol{R}, \hat{\boldsymbol{R}}, \mathbf{pok}, \boldsymbol{V}, \hat{\boldsymbol{V}}, \boldsymbol{C})$

$\underline{\mathsf{PVSS.DecryptShare}_{\mathsf{pp}}(\mathsf{trx}, i, \mathsf{dk}_{u(i)})} \to (\mathsf{sk}_i, V_i, \mathsf{pk}_i)$
$\left(\boldsymbol{R}, \cdot, \hat{\boldsymbol{V}}, \boldsymbol{C}\right) \leftarrow \mathsf{trx}$
$\mathsf{sk}_i \leftarrow C_i / (R_i)^{\mathsf{dk}_{u(i)}}$
$\mathsf{pk}_i \leftarrow \hat{V}_i$
return $(\mathsf{sk}_i, \hat{V}_i, \mathsf{pk}_i)$

$\underline{\mathsf{PVSS.Reconstruct}_{\mathsf{pp}}(S, (\mathsf{sk}_i, \hat{V}_i)_{i \in S})} \to \mathsf{sk}$
// Same as Figure 8 except with a polynomial of degree $w$

$\underline{\mathsf{SCRAPE.LowDegreeTest}(\boldsymbol{V}, w, W)} \in \{0,1\}$
// Same as Figure 8 except with a polynomial of degree
// $w$ and with $W + 1$ points
```

Fig. 9: A weighted PVSS protocol using a single public key per participant. Participant i has weight w_i, where the total weight is $W = \sum_{i=1}^{n} w_i$ and the threshold is w.

Inputs:

1. The parameters n, t of the PVSS scheme.
2. $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b) \in \mathbb{G}^3 \times \hat{\mathbb{G}}^3$ where $a, b \leftarrow\$ \mathbb{F}$.
3. Let $\mathcal{M} \subset [n]$ with $|\mathcal{M}| \leq t$ be the set of corrupt participants. Let $\mathcal{H} = [n] \setminus \mathcal{M}$ be the set of honest participants.
4. $(\mathsf{dk}_i, \mathsf{ek}_i) \in \mathbb{F} \times \mathbb{G}$ for each $i \in \mathcal{M}$, be the encryption and decryption of the malicious participants. We assume that the simulator $\mathcal{S}$ can extract the decryption keys of malicious participants using the proof-of-knowledge extractor.

Transcript and Encryption Key Generation:

5. The public parameters are $(g, \hat{g}, h = g^b)$.
6. Let g^a and $\hat{g}^a$ be the PVSS public commitment, i.e., $V_0 = g^a, \hat{V}_0 = \hat{g}^a$. This means that $h^a = g^{ab}$ is the shared secret. (Note that, $\mathcal{S}$ cannot directly compute g^{ab}).
7. Let $a(\cdot) \in \mathbb{F}[x]$ be a polynomial of degree t such that $a(0) = a$ and the PVSS secret of each participant i is $h^{a(i)}$. Compute $a(\cdot)$ as follows. For each malicious participant $i \in \mathcal{M}$, sample $a(i) \leftarrow\$ \mathbb{F}$, and compute $V_i = g^{a(i)}, \hat{V}_i = \hat{g}^{a(i)}$. (If $|\mathcal{M}| < t$ then choose at random $t - |\mathcal{M}|$ additional values $a(j)$ for values $j > n$ in order to define the polynomial.)
8. Compute $V_i = g^{a(i)}, \hat{V}_i = \hat{g}^{a(i)}$ for each $i \in \mathcal{H}$ using interpolation in the exponent. (Note that, unlike computing these values, $\mathcal{S}$ cannot compute the share $h^{a(i)}$ since it does not know $h^{a(0)}$).
9. For each honest participant $i \in \mathcal{H}$, sample $\theta_i \leftarrow\$ \mathbb{F}$, and use $\mathsf{ek}_i = g^{\theta_i - a(i)}$. Compute the required proof-of-knowledge (PoK) for the decryption key $\mathsf{dk}_i = \theta_i - a(i)$, using the NIZK simulator of the PoK protocol.
10. Compute the ciphertexts as follows.
    (a) Sample $r' \leftarrow\$ \mathbb{F}$ and set $R = g^b g^{r'}$ and $\hat{R} = \hat{g}^b \hat{g}^{r'}$. This implicitly sets $r = \log_g R = b + r'$.
    (b) For each malicious participant $i \in \mathcal{M}$, compute the ciphertexts as per the protocol specification, i.e., $C_i = h^{a(i)} \mathsf{ek}_i^r$. Use knowledge of $a(i)$ and dk_i for $i \in \mathcal{M}$, to compute these ciphertexts as $C_i = h^{a(i)} R^{\mathsf{dk}_i}$.
    (c) For each honest participant $i \in \mathcal{H}$, compute its ciphertext C_i:
    $$C_i = R^{\theta_i} g^{-a(i)r'} = g^{\theta_i r - a(i)r'} = g^{(\theta_i - a(i))r + b \cdot a(i)}$$
    using $g^{a(i)}$, r' and θ_i. Since $\mathsf{ek}_i = g^{\theta_i - a(i)}$, C_i is equal to $\mathsf{ek}_i^r \cdot g^{b \cdot a(i)} = \mathsf{ek}_i^r \cdot h^{a(i)}$.
11. Output: $\hat{R}$, pok, $[V_i]_{i \in [0,n]}$, $[\hat{V}_i]_{i \in [0,n]}$, $[C_i]_{i \in [n]}$ as the PVSS transcript.

Fig. 10: PVSS simulator $\mathcal{S}$ for the threshold setting.

This change saves the cost of running the low-degree test. In addition, the aggregation of two transcripts can be done by multiplying these values, instead of multiplying the evaluations of the polynomial.

– Of the W values in the vector $\boldsymbol{R}$, the simulation needs to set only the $W - w$ values corresponding to the shares of the honest participants. Therefore instead of sending the full vector it is possible to define a polynomial of degree $W - w - 1$ which has the right values for the shares of the honest participants, and arbitrary values for the other shares. This polynomial can be sent instead of $\boldsymbol{R}$, and the vector $\boldsymbol{R}$ can be computed from it using interpolation in the exponent. The same is also true for $\hat{\boldsymbol{R}}$. *The length of each of these polynomials is $W - w$.*

With these optimization, the transcript is $3W$ elements, i.e., W elements for the ciphertexts, w elements for each of $\boldsymbol{V}$ and $\hat{\boldsymbol{V}}$, and $W - w$ elements for each of $\hat{\boldsymbol{R}}$ and $\boldsymbol{R}$.

7 PVSS Security Analysis

7.1 Security of Threshold PVSS

We first prove the secrecy property of the unweighted threshold PVSS of Protocol 8. This immediately proves the security of the weighted PVSS described in §6.4, which is based on virtualization.

Theorem 6. *Protocol 8 satisfies the secrecy property as per Definition 10.*

Proof. The simulation in Figure 10 generates a transcript which has a distribution that has a negligible statistical difference to the distribution of the transcript generated by the PVSS protocol in Figure 8. There are some issues that need to be noted:

- The difference in the distributions of the simulation and the real execution can be caused by extraction errors incurred by the simulator when attempting to extract the private keys of the participants controlled by $\mathcal{A}$. This happens with negligible probability.
- Step 4 of the simulation extracts the private keys of all malicious participants using the PoK extractor. Since knowledge extraction needs to rewind the execution, it might cause issues in extracting in parallel the keys of the multiple malicious participants. This issue can be circumvented through using Fischlin's efficient conversion from a Sigma protocol to a zero-knowledge protocol with non-rewinding extraction [Fis05].
- One might be worried that a small difference between the distributions of the simulation and the real-protocol execution might be caused by the following issue: In the simulated world $\mathcal{S}$ needs to program the random oracle in order to successfully simulate the proofs of knowledge. However, $\mathcal{S}$ cannot do this programming on points at which $\mathcal{A}$ has already queried the random oracle. The probability of hitting such points is negligibly small, but this issue can be avoided altogether by having the simulator first observe which random oracle queries it needs to program to cheat in the proofs of knowledge, program their outputs accordingly, and only then let $\mathcal{A}$ send queries to the random oracle.

Hence, this PVSS scheme ensures Secrecy as per Definition 10.

7.2 Security of Weighted PVSS with a Single Key per Participant

We prove here the security of the weighted PVSS scheme with a single key per participant described in Figure 9 of Section 6.5. The proof follows the proof for the unweighted case. The simulator that is presented there in Figure 10 is replaced with a new simulator which is described in Figure 11. The main difficulty is that in the unweighted case the simulator generates encryptions of the shares of the honest participants by setting their private keys to values that agree with the checks performed by the protocol. This was possible since each share was encrypted with a different public key. The difference in the weighted case is that a single public key is used for encrypting multiple shares, and therefore the key cannot be set to a value which agrees with all these encryption. Instead, we use the fact that a different r_i value is used for each encryption of a share. The simulator sets these r_i values for the checks to succeed.

Theorem 7. *The weightes PVSS protocol of Figure 9 is secure as per Definition 10 (using the simulator of Figure 11).*

Proof. The simulator in Figure 11 has the same distribution which is statistically close, up to a negligible difference, to the PVSS transcript in the protocol of Figure 9. Differences between the simulation and execution distributions can only be attributed by extraction errors that occur when the simulator attempts to extract the private keys from A's controlled participants when they run the PoKs. This happens with negligible probability.

The public keys of the honest participants are uniformly distributed both in the protocol and in the simulation. The vectors $\boldsymbol{V}, \hat{\boldsymbol{V}}$ are generated in the simulation in exactly the same way as in the real protocol (except for interpolation being done in the exponent). The proofs of knowledge are generated by a NIZK simulator and therefore also have the same distribution. The vector $\boldsymbol{R}$ in both the protocol and the simulation are uniformly distributed, and the vector $\hat{\boldsymbol{R}}$ is uniquely defined by $\boldsymbol{R}$. The vector $\boldsymbol{C}$ is composed of elements C_i which are uniquely defined by the vector $a()$ (which is defined by $\boldsymbol{V}$), $\boldsymbol{R}$ and the public keys. Since $\boldsymbol{V}$, $\boldsymbol{R}$ and the public keys have the same distribution in the real run and in the simulation, so does the vector $\boldsymbol{C}$.

8 Distributed Key Generation (DKG)

8.1 DKG Definition

Definition 11. *A distributed key generation protocol for a weighted (W, w) VUF protocol amounts to secret sharing a uniformly random value $z = h^a \in \mathbb{G}$ and making public the value $y = \hat{g}^a$. Assume there are n parties with weights $w_1, \ldots, w_n$, s.t. $\sum_{i=1,\ldots,n} w_i = W$ and $s_i = \sum_{j=1,\ldots,i-1} w_j$. Let $p(\cdot) \in \mathbb{F}[x]$ be a polynomial of degree w such that*

Inputs:

1. The parameters W, w of the PVSS scheme, as well as the weights $w_1, \ldots, w_n$ of the participants.
2. $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b) \in \mathbb{G}^3 \times \hat{\mathbb{G}}^3$ where $a, b \leftarrow\$ \mathbb{F}$.
3. Let $\mathcal{M} \subset [n]$ be the set of corrupt participants, whose total weight $\sum_{i \in \mathcal{M}} w_i$ is at most w. Let $\mathcal{H} = [n] \setminus \mathcal{M}$ be the set of honest participants.
4. $(\mathsf{ek}_i, \mathsf{dk}_i) \in \mathbb{F} \times \mathbb{G}$ for each $i \in \mathcal{M}$, being the encryption and decryption of the malicious participants. (Unlike the unweighted setting, this simulator does not need to extract the decryption keys of the malicious participants.)

Transcript and Encryption Key Generation:

5. The public parameters of the PVSS are $(g, \hat{g}, h = g^b)$.
6. Let g^a and $\hat{g}^a$ be the PVSS public commitment. Namely, $\mathcal{S}$ sets $V_0 = g^a, \hat{V}_0 = \hat{g}^a$. This means that $h^a = g^{ab}$ is the shared secret. However, $\mathcal{S}$ does not have access to g^{ab}.
7. Let $a(\cdot) \in \mathbb{F}[x]$ be the polynomial of degree w such that $a(0) = a$ and PVSS secret of each participant i is $h^{a(i)}$. $\mathcal{S}$ computes the polynomial $a(\cdot)$ in the following way: For each share i which belongs to a malicious participant in $\mathcal{M}$, $\mathcal{S}$ samples $a(i) \leftarrow\$ \mathbb{F}$, and computes $V_i = g^{a(i)}, \hat{V}_i = \hat{g}^{a(i)}$. If $\sum_{i \in \mathcal{M}} w_i < w$ then $\mathcal{S}$ chooses at random $w - \sum_{i \in \mathcal{M}} w_i$ additional values $a(j)$ for values $j > w$ in order to define the polynomial.)
8. $\mathcal{S}$ then computes $V_i = g^{a(i)}, \hat{V}_i = \hat{g}^{a(i)}$ for each index i of a share belonging to the participants in $\mathcal{H}$. This is done using interpolation in the exponent.
9. For each honest participant $\ell \in \mathcal{H}$, the simulator $\mathcal{S}$ samples $\theta_\ell \leftarrow\$ \mathbb{F}$, and computes $\mathsf{ek}_\ell = g^{b \cdot \theta_\ell}$. $\mathcal{S}$ computes the required proof-of-knowledge for the decryption key $\mathsf{dk}_i = b \cdot \theta_\ell$, using the NIZK simulator of the PoK protocol.
10. $\mathcal{S}$ computes the ciphertexts as follows.
    (a) For each $i \in [W]$, $\mathcal{S}$ sets R_i and $\hat{R}_i$ in the following way:
      – If share i belongs to a participant $\ell \in \mathcal{M}$, then $\mathcal{S}$ chooses r_i at random and sets $R_i = g^{r_i}, \hat{R}_i = \hat{g}^{r_i}$.
      – Otherwise, if share i belongs to a participant $\ell \in \mathcal{H}$, $\mathcal{S}$ samples $d_i \leftarrow\$ \mathbb{F}$ and sets $R_i = (g^{a(i)})^{-1/\theta_\ell} \cdot g^{d_i}$ and $\hat{R}_i = (\hat{g}^{a(i)})^{-1/\theta_\ell} \cdot \hat{g}^{d_i}$. This implicitly sets $r_i = -a(i)/\theta_\ell + d_i$.
    (b) For each malicious participant $\ell \in \mathcal{M}$ and a share i belonging to this participant, $\mathcal{S}$ computes the ciphertexts as per the protocol specification. More precisely, $\mathcal{S}$ uses its knowledge of $a(i)$ and r_i to compute $C_i = h^{a(i)} \mathsf{ek}_\ell^{r_i}$. (Unlike the unweighted setting there is no need for $\mathcal{S}$ to extract dk_ℓ in order to compute this ciphertext as $C_i = h^{a(i)} R_i^{\mathsf{dk}_\ell}$.)
    (c) For each honest participant $\ell \in \mathcal{H}$, recall that the ciphertext should be of the form $h^{a(i)} \cdot \mathsf{ek}_\ell^{r_i}$. This value is equal to $g^{b \cdot a(i)} \cdot (g^{b \cdot \theta_\ell})^{-a(i)/\theta_\ell + d_i} = g^{b \cdot a(i) - b \cdot a(i) + b \cdot \theta_\ell \cdot d_i} = g^{b \cdot \theta_\ell \cdot d_i}$. $\mathcal{S}$ uses its knowledge of θ_ℓ, d_i and g^b to compute the ciphertext as $C_i = (g^b)^{\theta_\ell \cdot d_i}$.
11. $\mathcal{S}$ then outputs: $\boldsymbol{R}, \hat{\boldsymbol{R}}, \mathsf{pok}, [V_i]_{i \in [0,n]}, [\hat{V}_i]_{i \in [0,n]}, [C_i]_{i \in [n]}$ as the PVSS transcript.

Fig. 11: PVSS simulator $\mathcal{S}$ for the weighted setting, where each participant has a single public key.

$z := h^{p(0)}$. At the end of the protocol party i outputs the w_i shares $h^{p(s_i+1)}, \ldots, h^{p(s_i+w_i)}$ of the secret z. We require our DKG protocol to satisfy the following correctness *properties in the presence of an adversary $\mathcal{A}$ that corrupts parties with up to w shares.*

(C1) All subsets of $w + 1$ shares provided by honest parties define the same unique secret key $z = h^a$.

(C2) All honest parties output the same public key $y = \hat{g}^a$ where a is the the discrete log to the base h of the unique secret guaranteed by (C1).

Applications of DKG *such as threshold signatures and threshold encryption require that in addition to y, threshold public keys of all parties are also publicly known. So we add a fourth requirement.*

(C3) All honest parties agree on and output the public keys of all parties. The public key of party i is $y_i = \hat{g}^{p(s_i+1)}, \ldots, \hat{g}^{p(s_i+w_i)}$.

We want to note that we do not require our DKG protocol to satisfy notions of secrecy, such as the secret key being uniformly random or some simulatability-based secrecy definition as required by many existing DKG protocols [FS01,GJKR07,NBBR16,GJM+21b,SBKN21,DYX+22]. Instead, we directly prove that our weighted VUF scheme when combined with our weighted DKG scheme is secure assuming hardness of bilinear Diffie-Hellman (BDH) assumption. A similar approach was used in [GJKR07,CGRS23] to directly prove the combination of Pedersen's DKG scheme [Ped91] with many existing threshold cryptosystems. We present our combined proof in Section 9.

8.2 Designing a Weighted DKG

Given a non-interactive weighted PVSS scheme, the natural approach for designing a weighted DKG is the following: First, each party, as a dealer, computes a PVSS transcript for a random secret and broadcasts the transcript to all using a total order broadcast channel (we recall the definition of total order broadcast in Definition 12). Then, each party locally validates all the PVSS transcripts it receives from the broadcast channel and discards the invalid ones. Finally, each party derives its share of the DKG key by decrypting its share from the valid PVSS transcripts and aggregating them locally.

This approach has appeared in the literature many times [FS01,SJSW19,Gro21,KMM$^+$23]. Apart from its simplicity, it has additional advantages: It uses the total order broadcast channel in a black-box manner. In addition, this approach can use more efficient non-aggregatable PVSS schemes, such as [CD17,CDGK23].

On the other hand, this approach has some disadvantages: First, it requires all PVSS transcripts to be sent over the broadcast channel, which can be prohibitively expensive. This approach also has higher latency, as it might not be possible to fit PVSS transcripts from all dealers into a single message (block) in a blockchain.

We adopt a different approach that leverages the aggregation property of our PVSS scheme. It addresses the above-mentioned concerns in the common-case operation (i.e., with no or few active corruptions), and is thus more appropriate for our use case of on-chain randomness for PoS blockchains. The main advantage of this approach is that the relatively expensive broadcast channel is used to agree on only a single valid aggregated transcript, rather than on the PVSS transcripts of all parties.

The public parameters for our DKG scheme consist of the public parameters of the PVSS scheme, the vector $\mathbf{ek}$ of encryption keys of all the parties, and a vector of the verification keys from the signature/verification key-pairs of all parties. With these public parameters, our DKG scheme works in three simple phases: *Sharing, Agreement*, and *Key derivation*. We summarize our DKG scheme in Figure 12, and describe each phase next.

Sharing phase. During the sharing phase, each party i samples a uniformly random secret $s_i \leftarrow\$ \mathbb{F}$ and computes the PVSS transcript $\mathsf{trx}_i := (\mathsf{pok}_i, \cdot) \leftarrow \mathsf{PVSS.Deal}_{\mathsf{pp}}(\mathbf{ek}, s_i)$. Party i then signs its proof-of-knowledge pok_i. Let σ_i be this signature. Party i then sends the message $\langle \mathsf{SHARE}, \sigma_i, \mathsf{trx}_i \rangle$ to all parties over a peer-to-peer channel (rather than over a broadcast channel). Note that party i signs only pok_i and not the whole transcript trx_i. As we discuss later, this is intentional and necessary.

Agreement phase. During the agreement phase, each party i also locally maintains an aggregated PVSS transcript trx initialized as $\mathsf{trx} := \mathsf{trx}_i$, and a set of signatures $\boldsymbol{\sigma}$, initialized as $\boldsymbol{\sigma} := \{\sigma_i\}$. Upon receiving a message $\langle \mathsf{SHARE}, \sigma_j, \mathsf{trx}_j \rangle$ from party j, party i validates that σ_j is a valid signature on the PoK included in trx_j, and uses $\mathsf{PVSS.Verify}$ to verify that trx_j is a valid PVSS transcript. If both checks are successful, it aggregates trx_j and trx using $\mathsf{PVSS.Aggregate}$, and updates $\boldsymbol{\sigma}$ as $\boldsymbol{\sigma} := \boldsymbol{\sigma} \cup \{\sigma_j\}$. Party i also maintains the sum of the weights of the dealers of PVSS transcripts it has aggregated so far.

Next, we choose an arbitrary party as a broadcaster of the aggregated transcript. This choice is arbitrary (for example, in a blockchain setting the broadcaster can be the proposer of the next block), and need not be agreed upon by all parties. The broadcaster waits until it aggregates PVSS transcripts from dealers with a combined weight greater than w. IT then publishes the aggregated transcript $(\boldsymbol{\sigma}, \mathsf{trx})$ using a total order broadcast channel.

Every other party waits to receive $(\boldsymbol{\sigma}, \mathsf{trx})$ on the broadcast channel. For a broadcast output $(\boldsymbol{\sigma}, \mathsf{trx} = (\mathbf{pok}, \cdot))$, let T be the set of parties whose signatures are in $\boldsymbol{\sigma}$. Each recipient uses the $\mathbf{pok}$ in trx to locally check that: (i) $\forall k \in T$, $\sigma_k \in \boldsymbol{\sigma}$ is a valid signature on $\mathsf{pok}_k \in \mathbf{pok}$; (ii) trx is the aggregation of the PVSS transcripts of parties in T. (iii) The combined weight of dealers in T is $> w$.

If all these checks are successful, each party outputs trx as the aggregated transcript for the DKG and proceeds to the key-derivation phase. In case multiple valid aggregated transcripts are sent over the broadcast channel, each party outputs the first valid aggregated transcript received on the broadcast channel as the transcript for DKG. If the check fails, we choose a different party j (say the next block proposer in a blockchain), and let party j broadcast its locally aggregated transcript. We continue this process, until the first valid aggregated transcript is output by the total order broadcast.

```
PUBLIC PARAMETERS:
1: PVSS public parameters $\mathsf{pp} = (\cdot, \mathbb{F})$
2: Weights of all parties $\hat{w} = [w_1, \ldots, w_n]$
3: Encryption keys $\mathbf{ek}$ of all parties
4: Signature verification keys of all parties
5: Decryption key $\mathsf{dk}_i$ and a signature key

SHARING PHASE:
6: Let $s_i \leftarrow\!\$\ \mathbb{F}$
7: Let $\mathsf{trx}_i := (\mathsf{pok}_i, \cdot) \leftarrow \mathsf{PVSS.Deal}_{\mathsf{pp}}(\mathbf{ek}, s_i)$
8: Let $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{pok}_i)$
9: send $(\sigma_i, \mathsf{trx}_i)$ to all.

AGREEMENT PHASE:
10: Let $w_+ := w_i$, $\mathsf{trx} := \mathsf{trx}_i$, and $\boldsymbol{\sigma} = \{\sigma_i\}$
11: upon receiving $(\sigma_j, \mathsf{trx}_j)$ from party $j$ do
12:     if $\mathsf{PVSS.Verify}_{\mathsf{pp}}(\mathbf{ek}, \mathsf{trx}_j) = 1$ and $\sigma_j$ is valid then
13:         $\mathsf{trx} := \mathsf{PVSS.Aggregate}(\mathsf{trx}, \mathsf{trx}_j)$
14:         $\boldsymbol{\sigma} := \boldsymbol{\sigma} \cup \{\sigma_j\}$ and $w_+ := w_+ + w_j$
15:         if $w_+ \geq w$ then
16:             break
    // Repeat until the first honest party outputs
17: if chosen as a broadcaster then
18:     broadcast $(\boldsymbol{\sigma}, \mathsf{trx})$ using a total order broadcast
19: upon receiving $(\boldsymbol{\sigma}, \mathsf{trx})$ from the broadcast channel do
20:     Let $T$ be the indices of signers with signatures in $\boldsymbol{\sigma}$
21:     assert $\sigma_k \in \boldsymbol{\sigma}$ for each $k \in T$ are valid
22:     assert $\mathsf{trx}$ is valid for the set $T$
23:     assert $\sum_{i \in T} w_i \geq w$
24:     if all checks are successful then
25:         output $\mathsf{trx}$ and go to the key-derivation phase

KEY DERIVATION PHASE:
26: Let $\mathsf{trx} = (\cdot, \hat{\boldsymbol{V}}, \cdot)$ be the output of the agreement phase.
27: Let $\mathsf{sk}_i := \mathsf{PVSS.DecryptShare}(\mathsf{trx}, i, \mathsf{dk}_i)$
28: Let $\mathsf{pk} := \hat{\boldsymbol{V}}[0]$ and $\mathsf{pk}_j := \hat{\boldsymbol{V}}[s_{j-1} : s_j]$ for each $j \in [n]$
29: return $\mathsf{sk}_i, \mathsf{pk}, \{\mathsf{pk}_j\}_{j \in [n]}$
```

Fig. 12: Weighted DKG protocol for party i

We emphasize that the security of this weighted DKG and the resulting VUF is unaffected by the choice of the broadcaster and the aggregated transcript that it chooses. Intuitively, this is since every aggregated transcript contains a contribution from at least one honest party, and this contribution is unknown to an adversary.

Key-derivation phase. Let trx be the aggregated transcript that parties agree on during the agreement phase. During the key-derivation phase, each party i locally derives its VUF secret signing key sk_i by decrypting its share from the aggregated transcript trx using $\mathsf{PVSS.DecryptShare}(\mathsf{trx}, i, \mathsf{dk}_i)$, where dk_i is the private decryption key for the encryption key ek_i. Party i then extracts the public key pk and the threshold public keys $\{\mathsf{pk}_j\}_{i \in [n]}$ from the $\hat{\boldsymbol{V}}$ vector of the transcript.

8.3 Security Analysis

The correctness of our DKG follows from the correctness of the PVSS scheme and the security guarantees of the total order broadcast protocol (see Definition 12). In Section 9, we prove the security of our weighted VUF scheme when the DKG protocol generates the VUF keys. Again, similar to the security of VUF with trusted key generation, our security proof of VUF with DKG relies on the hardness of BDH in the random oracle model.

9 Security Analysis of keying the VUF with the DKG

This section proves the security of the VUF when the keys are generated using the distributed key generation of the DKG construction, according to the security definition in Section 3.2. The proof is in the random oracle model, and is based on the computational Bilinear Diffie-Hellman (BDH) assumption.

Theorem 8. *The unweighted threshold VUF protocol of Figure 2 satisfies the unforgeability property when key generation is done using a DKG of Figure 12, that is based on the PVSS protocol of Figure 8.*

Proof. $\mathcal{A}_{\mathrm{BDH}}$, which is described in Figure 13, is given a BDH challenge, simulates the environment with which $\mathcal{A}$ interacts, and given $\mathcal{A}$'s VUF forgery generates an answer for the BDH challenge.

First, let us prove that $\mathcal{A}$'s view in the simulation that $\mathcal{A}_{\mathrm{BDH}}$ generates has the same distribution as $\mathcal{A}$'s view in a real run of the protocol. The simulation has three parts: encryption key registration, DKG simulation, and VUF signature simulation. The encryption key registration happens once, when the system is set up. To simplify the simulation we assume that DKG simulation happens once, when the VUF key is generated. The VUF signature simulation covers many VUF invocations that use the key generated in the DKG. In a real system the DKG might be run periodically, and after each DKG invocation there is a phase of VUF signature simulation which uses the key generated in the recent DKG. This does not change the success probability of the simulation, since the encryption keys that are generated in the encryption key registration phase are independent of the keys generated in the DKG phase. The only "guess" that the simulator has to make is about which honest dealer will be chosen to participate in the PVSS aggregation in the phase in which $\mathcal{A}$ is able to forge a VUF output. This always happens with probability greater than $1/n$.

Encryption key registration: The simulation of the encryption key registration phase is identical to the simulation in Figure 10 of the PVSS protocol and therefore generates a view that has the same distribution as in the real protocol.

DKG simulation: The simulation of the DKG phase generates PVSS transcripts and augmented public keys for all participants. The generation of the PVSS transcripts for all dealers but i' is as in the real run of the protocol, and the transcript of i' is generated as in the PVSS simulation of Figure 10 and thus has the same distribution as in the real protocol (up to a negligible error). The augmented public keys are generated as in the VUF simulation of Figure 5 and thus have the same distribution as in the real protocol.

VUF signature simulation: The simulation of the VUF signature is the same as the simulation in Figure 5 of the VUF protocol, hence it has an identical distribution to that in the real protocol.

Next, let us verify that the *final output* of $\mathcal{A}_{\mathrm{BDH}}$ indeed solves the BDH challenge. Let ρ be the value that $\mathcal{A}$ outputs as the VUF output. If $\mathcal{A}$ is correct then $\rho = e(h^{a(0)}, \mathsf{H}_{\hat{\mathbb{G}}}(m_{k'})) = e((g^b)^{a(0)}, \hat{g}^c) = e(g, \hat{g})^{b\cdot c\cdot a(0)}$. $\mathcal{A}_{\mathrm{BDH}}$ then outputs

$$\begin{aligned}&\rho \cdot e((g^b)^{a_H + a_M}, \hat{g}^{-c})\\ &= e(g,\hat{g})^{b\cdot c\cdot a(0)} \cdot e(g,\hat{g})^{(-b\cdot c)(a_H+a_M)} = e(g,\hat{g})^{b\cdot c\cdot a}\end{aligned}$$

This is indeed the required output for the BDH challenge.

As for the *success probability* of $\mathcal{A}_{\mathrm{BDH}}$, conditioned on $i' \in Q$ (step 12) it is the same as the success probability of $\mathcal{A}$ in the VUF forgery game. Given that $\Pr(i \in) > 1/n$, the success probability of $\mathcal{A}_{\mathrm{BDH}}$ is smaller than that of $\mathcal{A}$ by a factor at most $1/n$.

9.1 The Weighted Setting

Next we prove the security of the weighted VUF when the keys are generated using the distributed key generation of the DKG construction. We show this proof both for the case where the PVSS uses virtualization and each participant has as many public keys as its weight (Section 6.4, and for the PVSS which uses one public key per participant, regardless of its weight (Figure 9). As in the unweighted case, the proof is in the random oracle model and is based on the computational Bilinear Diffie-Hellman (BDH) assumption.

Theorem 9. *The weighted VUF protocol of Figure 3 satisfies the unforgeability property when key generation is done using a DKG that is based on the PVSS protocol of Figure 8 and the virtualization paradigm of Section 6.4.*

Proof. $\mathcal{A}_{\text{BDH}}$, which is described in Figure 14, is given a BDH challenge, simulates the environment with which $\mathcal{A}$ interacts, and given $\mathcal{A}$'s VUF forgery generates an answer for the BDH challenge.

As in the proof for the unweighted case (Theorem 8), we prove that $\mathcal{A}$'s view in the simulation that $\mathcal{A}_{\text{BDH}}$ generates has the same distribution as $\mathcal{A}$'s view in a real run of the protocol. The simulation has three parts: encryption key registration, DKG simulation, and VUF signature simulation. The encryption key registration happens once, when the system is set up. To simplify the simulation we assume that DKG simulation happens once, when the VUF key is generated. The VUF signature simulation covers many VUF invocations that use the key generated in the DKG. In a real system the DKG might be run periodically, and after each DKG invocation there is a phase of VUF signature simulation which uses the key generated in the recent DKG. This does not change the success probability of the simulation, since the encryption keys that are generated in the encryption key registration phase are independent of the keys generated in the DKG phase. The only "guess" that the simulator has to make is about which honest dealer will be chosen to participate in the PVSS aggregation in the phase in which $\mathcal{A}$ is able to forge a VUF output. This always happens with probability greater than $1/n$.

Encryption key registration: As in the unweighted case, the simulation of the encryption key registration phase is identical to the simulation in Figure 10 of the PVSS protocol and therefore generates a view that has the same distribution as in the real protocol.

DKG simulation: The simulation of the DKG phase generates PVSS transcripts and augmented public keys for all participants. The generation of the PVSS transcripts for all dealers but i' is as in the real run of the protocol, and the transcript of i' is generated as in the PVSS simulation of Figure 10 and thus has the same distribution as in the real protocol. The augmented public keys are generated as in the weighted VUF simulation of Figure 6 and thus have the same distribution as in the real protocol.

VUF signature simulation: The simulation of the VUF signature is the same as the simulation in Figure 6 of the VUF protocol, hence it has an identical distribution to that in the real protocol.

Next, let us verify that the *final output* of $\mathcal{A}_{\text{BDH}}$ indeed solves the BDH challenge. Let ρ be the value that $\mathcal{A}$ outputs as the VUF output. If $\mathcal{A}$ is correct then $\rho = e(h^{a(0)}, \mathsf{H}_{\hat{\mathbb{G}}}(m_{k'})) = e((g^b)^{a(0)}, \hat{g}^c) = e(g, \hat{g})^{b \cdot c \cdot a(0)}$. $\mathcal{A}_{\text{BDH}}$ then outputs

$$\begin{aligned} &\rho \cdot e((g^b)^{a_H + a_M}, \hat{g}^{-c}) \\ &= e(g, \hat{g})^{b \cdot c \cdot a(0)} \cdot e(g, \hat{g})^{(-b \cdot c)(a_H + a_M)} = e(g, \hat{g})^{b \cdot c \cdot a} \end{aligned}$$

This is indeed the required output for the BDH challenge.

As for the *success probability* of $\mathcal{A}_{\text{BDH}}$, conditioned on $i' \in Q$ (step 12) it is the same as the success probability of $\mathcal{A}$ in the VUF forgery game. Given that $\Pr(i \in) > 1/n$, the success probability of $\mathcal{A}_{\text{BDH}}$ is smaller than that of $\mathcal{A}$ by a factor at most $1/n$.

Next, we state the security of the weighted VUF protocol when key generation is done using the DKG protocol, when the PVSS uses a single public key per participant, independently of its weight.

Theorem 10. *The weighted VUF protocol of Figure 3 satisfies the unforgeability property when key generation is done using a DKG that is based on the PVSS protocol of Figure 9.*

Proof. The proof is similar to that of the unweighted case (Theorem 8) and in based on the simulator in Figure 15. The only difference is in the simulation of the PVSS transcript of the honest participant which is chosen to deal the secret h^a which it does not know. This transcript is simulated as in the PVSS simulator of Figure 11, which is for the weighted case where each participant has a single key, rather than according to the PVSS simulator of Figure 10 which was previously used.

10 Evaluation

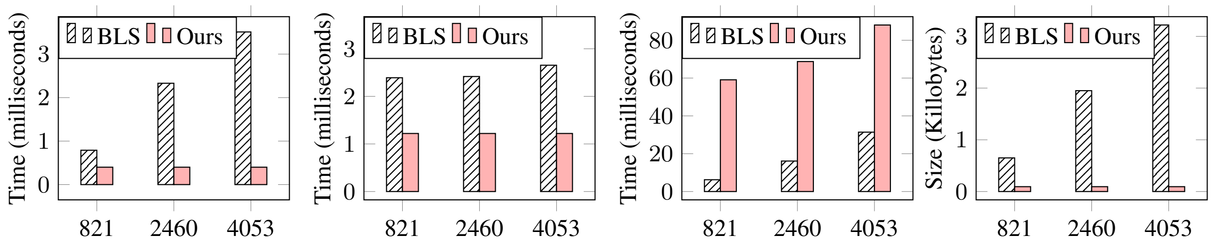
We implement our on-chain randomness in Rust atop of the open source implementation of Aptos blockchain [Apt22], a proof-of-stake blockchain (see `https://github.com/aptos-labs/aptos-core`). Our implementation includes all parts of our system, i.e., a weighted PVSS, a weighted DKG and a weighted VUF. We will make our implementation publicly available. For cryptography, our implementation uses the `blstrs` library [Sup24], which implements efficient finite field and elliptic curve arithmetic. We also use (for both our implementation and the baselines) multi-exponentiation of group elements using Pippenger's method [Ber02, §4].

10.1 Micro-benchmarks

Metrics. For our microbenchmark, we evaluate the *signing time*, *partial signature verification time*, *VUF derivation time*, *aggregation time*, and *signature size*. The signing time refers to the time it takes a signer to sign a message. The partial signature verification time measures the time the aggregator takes to verify a single partial signature. The VUF derivation time measures the time an aggregator takes to compute the VUF output given a set of valid partial signautres. The aggregation time measures the total time an aggregator takes to verify a subset of partial signatures necessary to compute the VUF output and the time the aggregator takes to derive the VUF output. More precisely, if partial signatures from k signers are needed to derive the VUF output, then the aggregation time is the sum total of k partial signature verification times and the VUF deriviation time. The signature size is the size of a partial signature of a signer.

For the abovementioned metrics, we compare our VUF scheme with Boldyreva's BLS threshold signatures with virtualization to support weights. While measuring the partial signature verification time of the BLS virtualization scheme, we implement the optimization where the verifier verifies all w_i partial signatures of signer i in a batch using only two pairings and two multi-exponentiations.

Results. We microbenchmark the computation costs using a t2d-standard-32 Google Cloud virtual machine, with 32 vCPUs, and 128 GBs of memory. We report our results in Table 16. The BLS signatures were computed over BLS12-381, corresponding to 128-bit security. Signature verification was done using batching, with random exponents that were chosen as random 256-bit scalars. Through our evaluation, we seek to illustrate that our scheme improves over the virtualization based approach of BLS threshold signatures.



(a) Signing time (b) Verification time (c) VUF derivation time (d) Signature size

Fig. 16: Micro-benchmarks of BLS virtualization and our scheme. Here, x-axis denotes the total weight of the system.

Signing time. As expected, the average per signer signing time in BLS virtualization grows linearly with the total weight of the system (from 0.79ms to 3.51ms), where in our VUF it is constant.

Partial signature verification time. The verification times of our scheme remain constant (1.22ms) with the total weight, and are about $2\times$ faster than those of BLS virtualization.

VUF derivation time. As observed, the VUF derivation time of BLS virtualization is proportional to the total weight. In our scheme, the aggregator needs to compute $O(n)$ additional pairings compared to the BLS virtualization approach, where n is the number of parties whose VUF shares are aggregated. For a smaller total weight W, these pairing computations amount to a non-trivial fraction of the VUF derivation time. Hence, our VUF derivation time is higher than the baseline. However, with increasing total weight, the time spent computing the multi-exponentiations and the Lagrange coefficients for BLS increases. This also explains a slower growth of the aggregation of our VUF. With even higher total weights, this gap will become narrower and insignificant. In addition, as our VUF derivation time depends on the number of validators, it will be smaller if the aggregator chooses to aggregate the final output from fewer validators with higher individual weights.

Aggregation time. We report the aggregation time of our scheme and BLS virtualization in Figure 17. For all three weight distributions we consider, we need to combine partial signatures from 74 signers on average. Thus, for both BLS virtualization and our approach, the aggregation time is the sum total of 74 partial signature verification time and the VUF derivation time. As expected, although the VUF derivation time of our scheme is higher than the BLS virtualization, the total time an aggregator will spend to compute the VUF output in our scheme is smaller. More precisely, the aggregation time of our VUF scheme is approximately 78% of that of the BLS virtualization approach. Furthermore, as we expect the gap between the VUF derivation time of our approach and the BLS virtualization to reduce with higher total weight, we expect our aggregation to also improve compared to BLS virtualization with increasing total weight.

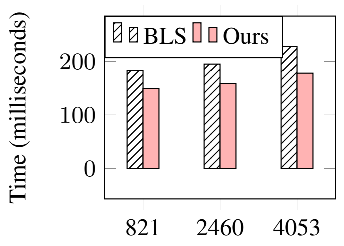


Fig. 17: Aggregation time

Total weights	DKG latency (sec)		On-chain randomness latency (ms)
	Sharing	Agreement	
821	1.4	18.6	160
2460	2.3	40.7	181
4053	3.0	61.0	203

Table 2: End-to-end latency with 112 validators

Partial signature size. As expected, in BLS virtualization the average per party signature size grows linearly with the number of shares (from 667.93 to 3297.35 bytes), and is constant (96 bytes) in ours. Even for the smallest total weight that we consider, 821, our scheme reduces the partial signature size by a factor $7\times$. The reduction is $34\times$ for total weight of 4053.

To summarize, even though the signature verification time is slower in our VUF, the overall aggregation time of our VUF is better than that of BLS threshold signatures. Furthermore, the communication overhead of our VUF is dramatically better than that of BLS signatures. This is particularly important, since computation can be easily parallelized, whereas communication cannot.

10.2 End-to-end Evaluation

Before we describe the end-to-end evaluation of on-chain randomness, we provide some background on the Aptos blockchain architecture. The blockchain proceeds in incremental epochs as described in §1, where each epoch lasts about two hours, and the validators and their stake distribution can change only during epoch changes.

Implementation. In our implementation, before every epoch change, validators run the weighted DKG to generate VUF keys for the next epoch. At the beginning of the new epoch, new validators decrypt their key pairs from the weighted DKG transcript. For any given block B, we use the hash of the VUF output on the epoch number and the block height at which B is committed, as the on-chain randomness for block B. To generate the VUF output for block B, validators wait until B is committed in the blockchain, and then exchange their VUF shares to generate on-chain randomness for block B. Each validator, upon receiving enough valid shares, locally generate the VUF output and hash it to derive the on-chain randomness for block B. We provide more details about our on-chain randomness implementation in Appendix C.

Note that it is straightforward to implement an independent randomness beacon, akin to Drand [Dra23] and Dfinity [HMW18], with our weighted DKG and weighted VUF. Moreover, our scheme facilitates the creation of efficient randomness beacons within the PoS setting.

Evaluation Setup. We ran experiments on Google Cloud, using 112 t2d-standard-32 type virtual machines spread equally across four simulated regions: us-central, eu-west, ap-northeast, sa-east. We report the simulated inter-region latencies in Table 3. They range between 100ms to 255ms for the average round trip time (RTT). The simulated intra-region round-trip latency is 100 ms. Each virtual machine has 32 vCPUs, 128 GBs of memory, and can provide up to 32 Gbps of network bandwidth.

Sending Region	**Receiving Region**	**AvgRTT (ms)**
us-central	eu-west	103.435
us-central	ap-northeast	133.996
us-central	sa-east	145.483
sa-east	us-central	145.703
sa-east	eu-west	176.894
sa-east	ap-northeast	255.289
eu-west	us-central	104.169
eu-west	sa-east	176.813
eu-west	ap-northeast	198.555
ap-northeast	us-central	128.999
ap-northeast	eu-west	198.539
ap-northeast	sa-east	255.323

Table 3: Simulated Round Trip Delays in E2E Evaluation

We use a weight distribution among the validators following the real-world stake distribution from 112 validators of the Aptos network as on Oct 18, 2023. We use three different total weight : 821, 2460 and 4053, and the corresponding weight distributions can be found below. We use 67% of the total weight as the reconstruction threshold for randomness beacon, to showcase the performance of our implementation even under high reconstruction threshold.

- Total weight = 821, weight distribution = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 6, 6, 6, 7, 7, 7, 9, 10, 14, 14, 15, 15, 15, 15, 15, 15, 16, 16, 16, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 20, 20, 20, 20].
- Total weight = 2460, weight distribution = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 5, 5, 5, 5, 5, 5, 6, 7, 7, 10, 11, 11, 11, 11, 11, 13, 14, 14, 15, 18, 18, 20, 20, 20, 22, 28, 31, 42, 44, 44, 44, 45, 46, 46, 46, 47, 47, 48, 50, 51, 51, 51, 51, 52, 54, 54, 54, 54, 54, 54, 54, 54, 54, 54, 54, 54, 54, 54, 54, 57, 57, 60, 60, 60, 60].
- Total weight = 4053, weight distribution = [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 8, 8, 8, 8, 8, 9, 11, 11, 12, 16, 18, 18, 18, 18, 19, 22, 23, 23, 25, 29, 30, 32, 33, 34, 36, 46, 51, 69, 72, 72, 73, 73, 76, 76, 76, 77, 78, 79, 82, 84, 84, 84, 84, 86, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 93, 94, 98, 98, 98, 98].

Results Metrics. We measure latency as our primary end-to-end performance metric. For DKG, we measure the latencies of the sharing phase and agreement phase, as the key derivation phase (a few milliseconds) is negligible compared to other phases. For on-chain randomness, the latency measures the time to generate randomness for each block, i.e., from the time when the block is finalized by consensus, until the time when the VUF is computed and the randomness of the block is generated.

Results. We summarize the evaluation results of DKG and on-chain randomness in Table 2. The latency of the DKG depends, almost linearly, in the total weight. This aligns with expectations, given that the communication and computation costs of the weighted PVSS are linear in the total weight.

The latency of on-chain randomness only marginally increases with the growth in the total weights. This can be due to the communication costs being unaffected by the total weight. The computation increases with the total weight, but it is easily parallelizable and not a bottleneck.

11 Discussion and Conclusion

In this paper, we presented a weighted VUF protocol with constant computation and communication costs for each party, regardless of their weight. In combination with the PVSS protocol presented in this work, this provides an efficient solution for generating randomness on-chain, in proof-of-stake blockchains.

12 Acknowledgements

We would like to thank Dan Boneh, Rex Fernando, Zekun Li, Zhoujun Ma, Alexander Spiegelman, and Michael Straka, for their help to this work.

References

Apt22. Aptos. The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure. 2022. Accessed: 2023-02-19.

BBBF18. Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.

BDN18. Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part II*, pages 435–464. Springer, 2018.

Ber02. Daniel J. Bernstein. Pippenger's exponentiation algorithm. 2002. URL: `https://cr.yp.to/papers/pippenger.pdf`.

BG17. Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

BL23. Renas Bacho and Julian Loss. Adaptively secure (aggregatable) PVSS and application to distributed randomness beacons. In *ACM CCS 2023*, pages 1791–1804. ACM, 2023.

BLL$^+$23. Renas Bacho, Christoph Lenzen, Julian Loss, Simon Ochsenreither, and Dimitrios Papachristoudis. Grandline: Adaptively secure dkg and randomness beacon with (almost) quadratic communication complexity. Cryptology ePrint Archive, Paper 2023/1887, 2023. `https://eprint.iacr.org/2023/1887`. URL: `https://eprint.iacr.org/2023/1887`.

Bra87. Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

BS23. Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography, v. 06.* January 2023.

BTW05. Amos Beimel, Tamir Tassa, and Enav Weinreb. Characterizing ideal weighted threshold secret sharing. In *Theory of Cryptography,*, volume 3378 of *Lecture Notes in Computer Science*, pages 600–619. Springer, 2005.

CD17. Ignacio Cascudo and Bernardo David. SCRAPE: scalable randomness attested by public entities. In *Applied Cryptography and Network Security - ACNS 2017*, volume 10355 of *Lecture Notes in Computer Science*, pages 537–556. Springer, 2017.

CD20. Ignacio Cascudo and Bernardo David. ALBATROSS: publicly attestable batched randomness based on secret sharing. In *ASIACRYPT 2020*, volume 12493 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.

CD23. Ignacio Cascudo and Bernardo David. Publicly verifiable secret sharing over class groups and applications to dkg and yoso. *Cryptology ePrint Archive*, 2023.

CDGK23. Ignacio Cascudo, Bernardo David, Lydia Garms, and Anders Konring. Yolo yoso: Fast and simple encryption and secret sharing in the yoso model. In *Advances in Cryptology – ASIACRYPT 2022*, page 651–680. Springer-Verlag, 2023.

cel24. Celo randomness documentation, 2024. URL: `https://docs.celo.org/protocol/randomness`.

CGMA85. Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In *26th Annual Symposium on Foundations of Computer Science, 1985*, pages 383–395. IEEE Computer Society, 1985.

CGRS23. Hien Chu, Paul Gerhart, Tim Ruffing, and Dominique Schröder. Practical schnorr threshold signatures without the algebraic group model. *Cryptology ePrint Archive*, 2023.

cha24. Chainlink vrf documentation, 2024. URL: `https://chain.link/vrf`.

CL02. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

CMB23. Kevin Choi, Aathira Manoj, and Joseph Bonneau. Sok: Distributed randomness beacons. *Cryptology ePrint Archive*, 2023.

Cor00. Jean-Sébastien Coron. On the exact security of full domain hash. In *Advances in Cryptology - CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 229–235. Springer, 2000.

CP92. David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *Annual international cryptology conference*, pages 89–105. Springer, 1992.

DGK$^+$20. Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927. IEEE, 2020.

Dod02. Yevgeniy Dodis. Efficient construction of (distributed) verifiable random functions. In *Public Key Cryptography—PKC 2003: 6th International Workshop on Practice and Theory in Public Key Cryptography Miami, FL, USA, January 6–8, 2003 Proceedings 6*, pages 1–17. Springer, 2002.

Dra23. Drand. Drand-a distributed randomness beacon daemon. `https://drand.love/`, 2023. Accessed: 2023-02-19.

dST23. Luciano Freitas de Souza and Andrei Tonkikh. Swiper and dora: efficient solutions to weighted distributed problems. *CoRR*, abs/2307.15561, 2023. URL: `https://doi.org/10.48550/arXiv.2307.15561`, `arXiv:2307.15561`, `doi:10.48550/ARXIV.2307.15561`.

DXT$^+$23. Sourav Das, Zhuolun Xiang, Alin Tomescu, Alexander Spiegelman, Benny Pinkas, and Ling Ren. A new paradigm for verifiable secret sharing. *Cryptology ePrint Archive*, 2023.

DYX$^+$22. Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *IEEE Security and Privacy (SP)*, 2022.

Eth. Ethereum. Ethereum RANDAO Specifications. URL: `https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md#randao`.

Fis05. Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In *Crypto '05*, pages 152–168, 2005.

FS87. Amos Fiat and Adi Shamir. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

FS01. Pierre-Alain Fouque and Jacques Stern. One round threshold discrete-log key generation without private channels. In *International Workshop on Public Key Cryptography*, pages 300–316. Springer, 2001.

GHM$^+$17. Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

GJKR07. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.

GJM$^+$21a. Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. In *EUROCRYPT 2021 Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 147–176. Springer, 2021. URL: `https://eprint.iacr.org/2021/005`.

GJM$^+$21b. Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 147–176. Springer, 2021.

GKR23. Peter Gaži, Aggelos Kiayias, and Alexander Russell. Fait accompli committee selection: Improving the size-security tradeoff of stake-based committees. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 845–858, 2023.

Gro21. Jens Groth. Non-interactive distributed key generation and key resharing. *Cryptology ePrint Archive*, 2021.

HBY23. Alex Hentschel and Tarak Ben Youssef. Flow random number generator, 2023. URL: `https://forum.flow.com/t/secure-random-number-generator-for-flow-s-smart-contracts/5110`.

HMW18. Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

KGS23. Chelsea Komlo, Ian Goldberg, and Douglas Stebila. A formal treatment of distributed key generation, and new constructions. *IACR Cryptol. ePrint Arch.*, page 292, 2023. URL: `https://eprint.iacr.org/2023/292`.

KMM$^+$23. Aniket Kate, Easwar Vivek Mangipudi, Pratyay Mukherjee, Hamza Saleem, and Sri Aravinda Krishnan Thyagarajan. Non-interactive vss using class groups and application to dkg. *Cryptology ePrint Archive*, 2023.

KWJ23. Alireza Kavousi, Zhipeng Wang, and Philipp Jovanovic. Sok: Public randomness. *Cryptology ePrint Archive*, 2023.

MRV99. Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.

Nak08. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008.

NBBR16. Wafa Neji, Kaouther Blibech, and Narjes Ben Rajeb. Distributed key generation protocol with a new complaint management strategy. *Security and communication networks*, 9(17):4585–4595, 2016.

Ped91. Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 522–526. Springer, 1991.

RG22. Mayank Raikwar and Danilo Gligoroski. Sok: Decentralized randomness beacon protocols. In *Australasian Conference on Information Security and Privacy*, pages 420–446. Springer, 2022.

Roc18. Team Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. *Available [online].[Accessed: 4-12-2018]*, 2018.

RY07. Thomas Ristenpart and Scott Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In *Advances in Cryptology-EUROCRYPT 2007: 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007. Proceedings 26*, pages 228–245. Springer, 2007.

SBKN21. Nibesh Shrestha, Adithya Bhat, Aniket Kate, and Kartik Nayak. Synchronous distributed key generation without broadcasts. *Cryptology ePrint Archive*, 2021.

Sch90. Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Advances in Cryptology—CRYPTO'89 Proceedings 9*, pages 239–252. Springer, 1990.

SJSW19. Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. Ethdkg: Distributed key generation with ethereum smart contracts. *Cryptology ePrint Archive*, 2019.

Sta96. Markus Stadler. Publicly verifiable secret sharing. In *EUROCRYPT '96*, volume 1070 of *Lecture Notes in Computer Science*, pages 190–199. Springer, 1996.

Sup24. Supranational. BLST: Bls signatures. `https://github.com/supranational/blst`, 2024.

TCZ$^+$20. Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. Towards scalable threshold cryptosystems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 877–893. IEEE, 2020.

Tom22. Alin Tomescu. Pairings or bilinear maps. `https://alinush.github.io/2022/12/31/pairings-or-bilinear-maps.html`, 2022. Accessed: 2023-02-19.

W$^+$14. Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

A Backgroun Material

Definition 12 (Total Order Broadcast). *In a distributed system with n parties $1, 2, \dots, n$, where each party can broadcast and deliver messages, a total order broadcast ensures the following properties:*

– **Agreement:** *If an honest party delivers a message m, then all honest parties eventually deliver m.*
– **Integrity:** *Honest parties delivers each message at most once.*
– **Validity:** *If a honest party broadcasts a message m, then all honest parties eventually deliver m.*
– **Total Order***: For any two messages m and m', if m is delivered before m' by any honest party, then m is delivered before m' by all honest party.*

B Rounding the Stake to Smaller Integer Weights

Let n be the number of validators, let $(w_1, \dots, w_n)$ be the stake per validator, and $W = \sum_{i=1}^{n} w_i$ be the total stake. For a set S of validators we use the notation $W(S) = \sum_{i \in S} w_i$.

The PVSS, DKG and VUF constructions assume that participants have integer weights w_i, and distribute a total of W shares. Proof-of-stake blockchains have a very large total weight (say, about a billion tokens), which makes it completely impractical to use one share per token unit. In this appendix, we describe methods for using a much smaller number of shares while analyzing and limiting the potential rounding errors that might be caused by this process.

A similar problem to ours was investigated in [dST23], although from a different angle: The starting point for both works is a set $(w_1, \dots, w_n)$ of weights, and a threshold w. The goal is to replace the original weights with much smaller weights, as well as compute a new threshold, so two properties are satisfied: (1) for almost all subsets S of participants, the sum of their new weights is greater than the new threshold if and only if $\sum_{i \in S} w_i > w$, and (2) the total sum of the new weights is minimal. Most likely, the accuracy property (property (1)) will not hold for all subsets, due to the usage of smaller weights. The work in [dST23] takes as input an allowable gap in accuracy and then assigns weights to satisfy this condition while minimizing the total weight. That paper shows that the assignment of weights can be done by heuristically solving a knapsack problem, and works very well in practice. We use a different approach, taking as input a target number of shares, or a small range for the number of shares, and computing the maximal disruption to accuracy that is possible with this number of shares. This approach also works well in practice. In other words, we replace the original threshold w with two new thresholds $w - \Delta_D, w + \Delta_U$, while ensuring that any original subset with weight smaller than $w - \Delta_D$ will not achieve the new threshold, whereas any original subset with weight greater than $w + \Delta_U$ will achieve the new threshold. The gap is accuracy, $\Delta_D + \Delta_U$ is typically small in practice, around 1%-2% for common stake distributions. We also describe a worst-case upper-bound on this gap.

B.1 How to Round

We set a parameter B so that a share is given for every B amount of stake. We limit our focus to share allocation which use rounding, and where a validator with w stake receives either $\lfloor w/B \rfloor$ or $\lceil w/B \rceil$ shares. To explore the entire range of rounding possibilities, we use a parameter $0 \leq c < 1$, where a validator with w stake receives $\lfloor w/B + c \rfloor$ shares. This method for rounding is depicted in Figure 18. It is obvious that $\lfloor w/B \rfloor \leq \lfloor w/B + c \rfloor \leq \lceil w/B \rceil$. The case $c = 0$ is the same as rounding down, and the case $c = 1 - \varepsilon$, when ε is vanishing, is the same as rounding up. The choice of c enables us to explore the range between these options. The case $c = 1/2$ corresponds to rounding to the closest integer.

B.2 Rounding to the Closest Integer

The following short analysis shows that the best outcome is when the number of shares is rounded to the closest integer, namely when using $c = 0.5$ and providing $\lfloor w/B + 0.5 \rfloor$ shares to a validator with weight w. In order to arrive at this conclusion, we take into consideration the following points:

This rounding scheme has some immediate properties:

- Validator i receives $t_i = \lfloor w_i/B + c \rfloor$ shares.
- The total number of shares is $T = \sum_{i=1,\ldots,n} t_i$. A set S of validators receives $T(S) = \sum_{i \in S} t_i$ shares.
- The *total rounded weight* is $W_R = \sum_{i=1,\ldots,n} (t_i \cdot B) = T \cdot B$. This is the total weight for which shares were actually assigned, and can be smaller or greater than the actual total weight.
- Let $R(S)$ be the *relative* original weight of a set of validators, i.e. $R(S) = \frac{W(S)}{W}$. Let $R_T(S)$ be the relative weight of the *shares* assigned to a set of validators, i.e. $R_T(S) = \frac{T(S)}{T}$. Rounding is likely to make $R_T(S)$ different than $R(S)$. The goal is to make $R_T(S)$ as close as possible to $R(S)$.
- Define $\delta_i = t_i - w_i/B$. This value corresponds to how many shares validator i gained due to rounding. If we rounded w_i/B down then δ_i is *negative* and is in the range $(-(1-c), 0]$. If we rounded up then $\delta_i \in [0, c)$ and is positive. In the case of rounding down, δ_i refers to the red areas in Fig. 18. For rounding up, δ_i refers to the green areas in this figure.
- Following is some useful notation. All the values defined here can be easily computed given the stake distribution $(w_1, \ldots, w_n)$, the weight per share B, and the rounding threshold c.
  - Δ_D is the total number of shares that were lost due to rounding down. Namely $\Delta_D = -\sum_{\delta_i < 0} \delta_i$. This is the sum of all red areas in Fig. 18. When rounding-up ($c = 1$), we have $\Delta_D = 0$.
  - Δ_U is the total number of shares that were gained due to rounding up. Namely $\Delta_U = \sum_{\delta_i > 0} \delta_i$. This is the sum of all green areas in Fig. 18. In the case of simple rounding-down ($c = 0$), we have $\Delta_U = 0$.
  - We also define $\Delta = \sum_{i=1}^{n} |\delta_i| = \Delta_D + \Delta_U$.

The following inequality corresponds to the most extreme case of increasing the weight of a committee due to rounding, which happens when all the weight that was added to validators was added to members of the committee.[6]

[6] A sharper bound can be achieved by considering only the green areas of subsets whose some is smaller than $R(S)$, and not the sum of all green areas. But this requires solving a knapsack problem.

The new relative weight corresponds to the original weight plus the the added weight (the green areas), divided by the total weight after rounding:

$$R_T(S) = \frac{T(S)}{T} \leq \frac{R(S) \cdot W + \Delta_U \cdot B}{W_R} \tag{6}$$

The following inequality corresponds to the most extreme case of a committee losing relative weight due to rounding, which happens when only members of this committee lose weight due to rounding:

$$R_T(S) \geq \frac{R(S) \cdot W - \Delta_D \cdot B}{W_R} \tag{7}$$

Limiting the uncertainty caused by rounding Suppose that we have a target threshold $0 < p < 1$ of the weights. We fix B and c, and can then calculate the total new weight W_R and the red and green areas Δ_D, Δ_U. Then

- A set with original relative weight $R(S) < (p \cdot W_R - \Delta_U \cdot B)/W$ will always have less than p relative weight after rounding. (Based on Equation 6.)
- A set with original relative weight $R(S) > (p \cdot W_R + \Delta_D \cdot B)/W$ will always have more than p relative weight after rounding. (Based on Equation 7.)
- We define the **uncertainty range** as the relative size of the sub-range of weights for which we are unsure whether the a validator set with that weight will achieve the threshold. We would like to minimize the size of the uncertainty range. The size of the uncertainty range is $\frac{(\Delta_D + \Delta_U) \cdot B}{W} = \Delta \cdot \frac{B}{W}$. (Note that this value does not depend on the threshold p.)
- It is easiest to analyze the effect of the rounding parameter c by observing Figure 18. Each validator is represented in that figure by a rectangle. Each rectangle has a horizontal line, corresponding to the fractional part of the weight of the validator, $f_i = w_i/B - \lfloor w_i/B \rceil$, where $f_i \in [0, 1)$. The contribution of the validator to the size of the uncertainty range is either f_i if $f_i \leq 1 - c$ (this corresponds to rounding down, i.e. a red area), or $1 - f_i$ if $f_i > 1 - c$ (this corresponds to rounding up, which is represented by a green area). We can observe that $\min(f_i, 1 - f_i) \leq 0.5$, and that setting $c = 0.5$ ensures that the contribution of this validator to the uncertainty range is exactly $\min(f_i, 1 - f_i)$, which is the minimal possible contribution.
- The **worst case** happens when all fractional parts (blue lines) have a value very close to 0.5, say 0.5001. In that case, regardless of whether we round up or down, the total weight that validators gain or lose is about 0.5 times the number of validators. In other words, the worst case is a loss of $n/2$ shares.
  This worst case occurs for a specific and very bad distribution of the stake to the validators. For each specific stake distribution and a value of B we can compute the corresponding worst case bound on the size of the uncertainty range.

Example: For example, for the specific distribution of the Aptos stake used in [dST23], the usage of $B = 500,000$ tokens per share and $c = 0.5$ ensures an uncertainty range of size 1.1%, and also minimizes $|\Delta_U - \Delta_D|$. For a threshold of $1/3$ and an uncertainty range around it, the uncertainty range is about $[0.3245, 0.3355]$, which is 1.1% of the total weight.

B.3 Algorithms

Based on the previous analysis, we describe several simple algorithms for round stakes into weights for secret sharing:

- Algorithm 1 implements a straightforward approach for calculating the maximum size of the uncertainty range.
- Algorithm 2 takes as input the distribution of stakes and a target threshold for the stake. It then calculates the allocation of shares (weight) to each validator and determines a new threshold for the shares. This threshold ensures that no committee with a stake below the target threshold of the stake will have a share count exceeding the new threshold of shares. This is done while minimizing the uncertainty range.
- Algorithm 3 checks a range of values for the parameter B (the amount of stake per share). For each value it runs Algorithm 2. It then returns the value of B which minimizes the size of the uncertainty range, as well as the corresponding assignment of shares.
- Algorithm 4 uses an upper bound for uncertainty range size as input, and searches for the smallest number of shares that maintains the uncertainty range below this bound.

Algorithm 1 – bounding the uncertainty range Algorithm 1 computes an upper bound on the size of the uncertainty range. This bound is independent of the stake distribution among the validators. The algorithm receives the following inputs:

– The number of validators n.
– The stake for each validator $(w_1, \dots, w_n)$.
– The amount of stake B for which a share is given (e.g., $B = 500,000$).

The algorithm outputs an upper bound on the size of the uncertainty range, which corresponds to the worst stake distribution (which is likely different than any given stake distribution). The output is given as the relative size of the uncertainty range compared to the total number of shares.

Algorithm 1 Algorithm for computing an upper bound on the uncertainty range.

```
function $\mathrm{UR}_{\mathrm{Bound}}(n, (w_1, \dots, w_n), B)$        ▷ $B$ is the weight per share
    $W \leftarrow w_1 + \cdots + w_n$        ▷ sum of all weights
    $WorstCaseUR \leftarrow n \cdot (B/2)/W$        ▷ worst case size of uncertainty range
    return $WorstCaseUR$
```

Algorithm 2 – allocating shares Algorithm 2 answers the following question: Given the stake of each validator, a target threshold $TargetThreshold$ of the stake, and the amount of stake per share B, compute the number of shares given to each validator and a new threshold $ThresholdShares$ of the shares, such that no committee of validators that has $TargetThreshold$ fraction of the stake, can have more than $ThresholdShares$ shares. This is done while minimizing the uncertainty range.

Algorithm 2 receives the following inputs:

– The number of validators n.
– The stake for each validator $(w_1, \dots, w_n)$.
– The amount of stake B for which a share is given (e.g., $B = 500,000$).
– The target threshold, *TargetThreshold*, which is a fraction of the stake in the range $(0, 1)$.

The algorithm rounds the number of shares per validator to the closest integer and outputs the following values:

– The number of shares per validator $(t_1, \dots, t_n)$.
– The total number of shares $T = t_1 + \cdots + t_n$. (This number T will be close to $(\sum_{i-1}^{n} w_i)/B$, but due to rounding it is likely to be different from it.)
– The new threshold expressed as a number of shares ($ThresholdShares$).
It is guaranteed that any subset of validators that has this many shares has total weight that is larger than the target threshold of the stake. In other words, if the original stake of a subset is less than the *TargetThreshold* input parameter (say, less than $1/3$ of the stake) then this subset of validators will have less than $ThresholdShares$ shares.
– The size of the uncertainty range, $UncertaintyRangeSize \in [0, 1]$.

Algorithm 3 – Finding how much stake to assign per share Algorithm 2 receives as input the value B, which is the amount of stake per share. It seems that a more natural input is the desired number of shares, as this number directly affects the overhead. Naively, the number of shares is just the total stake divided by B. A better approach, however, is to start with the desired number of shares and search for a good choice for B, for multiple reasons: (1) Different values of B might affect the accuracy of rounding. (It is definitely possible to show artificial examples demonstrating an extreme behavior of the accuracy of rounding depending on B.) (2) It is easier for the user of this algorithm to define the goal as the desired number of shares, than as a the amount of stake per share, B.

Algorithm 2 Algorithm for assigning shares to validators and computing a bound on the rounding error.

```
function ROUNDING($n, (w_1, \dots, w_n), B, TargetThreshold$)        ▷ $B$ is the weight per share
    $W \leftarrow w_1 + \cdots + w_n$                                  ▷ sum of all weights
    $DeltaD \leftarrow 0;\ \ DeltaU \leftarrow 0$
    for all $i \in \{1, \dots, n\}$ do
        $s_i \leftarrow w_i/B$                                         ▷ ideal number of shares
        $t_i \leftarrow \lfloor w_i/B + 0.5 \rfloor$                   ▷ rounded number of shares
        if $s_i - t_i > 0$ then                                        ▷ rounding down
            $DeltaD \leftarrow DeltaD + (s_i - t_i)$
        else                                                           ▷ rounding up
            $DeltaU \leftarrow DeltaU + (t_i - s_i)$
    $Delta \leftarrow DeltaD + DeltaU$
    $T \leftarrow t_1 + \cdots + t_n$                                  ▷ total number of shares
    $UncertaintyRangeSize \leftarrow Delta \cdot B/W$
    $ThresholdShares \leftarrow \lceil TargetThreshold \cdot W/B + DeltaU \rceil$   ▷ new threshold of shares
    return $(t_1, \dots, t_n), T, ThresholdShares, UncertaintyRangeSize$
```

Algorithm 3 checks multiple values of B in some small range around the value corresponding to the target number of shares, and returns the value which minimizes the size of the uncertainty range, as well as returning the corresponding assignment of shares. (The goal is not to minimize the number of shares, since we are searching for B in a small range and the number of shares will be about the same for all values of B that we are checking.)

Intuitively it seems that for random/natural assignments of stake, all values of B in a small range will behave similarly. It is better though that given the stake distribution we examine multiple options for B, in order not to stumble upon choices of stake distribution that, either adversely or through luck, result in a relatively large uncertainty range for a specific choice of B.

The input to Algorithm 3 contains $n, (w_1, \dots, w_n)$ and $TargetThreshold$ as in Algorithm 2. But instead of B, the input contains a range $[TS1, TS2]$ for the desired number of shares, and a number of attempts to try (this number could be large). The algorithm works by setting $TargetB$ to multiple values in a corresponding range of stake-per-share, and choosing the one which minimizes the size of the uncertainty range. The algorithm verifies that the number of shares is in the range $[TS1, TS2]$.

Algorithm 3 Find a weight per share which minimizes the uncertainty range under the constraint that the number of shares is in [TS1,TS2].

```
function FINDINRANGE($n, (w_1, \dots, w_n), TargetThreshold, TS1, TS2, attempts$)
    $B1 \leftarrow \lfloor 0.95 \cdot (w_1 + \cdots + w_n)/TS2 \rfloor$     ▷ an arbitrary lower bound for $B$
    $B2 \leftarrow \lfloor (1.05 \cdot (w_1 + \cdots + w_n)/TS1 \rfloor$    ▷ an arbitrary upper bound for $B$
    $Bstep = \ (B2 - B1)/attempts$                                      ▷ Need not be an integer
    $BestUCRS \leftarrow 1$                                             ▷ initial size of uncertainty range
    for $i \in [0, attempts]$ do                                        ▷ actually try $attempts + 1$ values
        $\lfloor B \leftarrow B1 + i \cdot Bstep \rfloor$               ▷ naive weight per share
        $((t_1, \dots, t_n), T, TS, UCRS) \leftarrow$ ROUNDING($n, (w_1, \dots, w_n), B, TargetThreshold$)  ▷ $TS$ is $ThresholdShares$,
$UCRS$ is $UncertaintyRangeSize$
        if $UCRS < BestUCRS$ and $TS \in [TS1, TS2]$ then               ▷ Reduced the uncertainty range
                ▷ We also check that the number of shares is within the range, since rounding might move it slightly
            $(bt_1, \dots, bt_n) \leftarrow (t_1, \dots, t_n)$
            $BestT \leftarrow T$
            $BestTS \leftarrow TS$
            $BestUCRS \leftarrow UCRS$
    return $(bt_1, \dots, bt_n), BestT, BestTS, BestUCRS$
```

Algorithm 4 – finding a minimal number of shares for a given size of the uncertainty range Algorithm 4 receives as input an upper bound for the size of the uncertainty range $LimitUCRS$. It searches for the smallest number of shares that maintains the size of the uncertainty range below this bound. We describe here a very simple algorithm that simply starts with $B = 500000$ and increases this parameter by steps of 100000 until the uncertainty range reaches the bound. The algorithm could be easily generalized.

The output of the algorithm is as in the previous algorithms, and in addition contains the value of the stake-per-share parameter B that is used.

Algorithm 4 Find an B which minimizes the number of shares, under the constraint that the uncertainty range is smaller than the given bound.

```
function FINDFORUCRS($n, (w_1, \dots, w_n), TargetThreshold, LimitUCRS$)
    $B \leftarrow 0$
    $NewB \leftarrow 500000$                                ▷ An arbitrary value that currently looks good
    $Step \leftarrow 100000$                                ▷ Step with which $B$ is increased
    $((Newt_1, \dots, Newt_n), NewT, NewTS, NewUCRS) \leftarrow$ ROUNDING($n, (w_1, \dots, w_n), NewB, TargetThreshold$)
    while $NewUCRS < LimitUCRS$ do                          ▷ uncertainty range still smaller than the bound
        $B \leftarrow NewB$
        $(t_1, \dots, t_n) \leftarrow (Newt_1, \dots, Newt_n)$
        $T \leftarrow NetT$
        $TS \leftarrow NetTS$
        $UCRS \leftarrow NetUCRS$                           ▷ $NewTS$ is $ThresholdShares$, $NewUCRS$ is $UncertaintyRangeSize$
        $NewB \leftarrow NewB + Step$
        $((Newt_1, \dots, Newt_n), NewT, NewTS, NewUCRS) \leftarrow$ ROUNDING($n, (w_1, \dots, w_n), NewB, TargetThreshold$)
    if $B == 0$ then                                        ▷ If algorithm failed in the first attempt
        print("Failed since the bound on the uncertainty range is too small.")
        break
    return $B, (t_1, \dots, t_n), T, TS, UCRS$
```

C On-chain Randomness

This section describes the details of our on-chain randomness implementation.

C.1 Preliminaries

Communication. All nodes are pairwise connected by a reliable and authenticated channel. The network is assumed to be asynchronous, although the protocol uses a BFT *SMR* (state machine replication, Definition 14) that may rely on stronger network assumptions such as partial synchrony. All nodes also have access to a Byzantine *reliable broadcast* channel defined as follows:

Definition 13 (Reliable Broadcast [Bra87]). *A protocol for a set of nodes $\{1, ..., n\}$, where a distinguished node called the broadcaster holds an initial input M, is a reliable broadcast protocol, if the following properties hold*

- Agreement: *If an honest node outputs a message M' and another honest node outputs M'', then $M' = M''$.*
- Validity: *If the broadcaster is honest, all honest nodes eventually output the message M.*
- Totality: *If an honest node outputs a message, then every honest node eventually outputs a message.*

State machine replication. Let *SMR* be a Byzantine fault tolerant state machine replication protocol defined as the following.

Definition 14 (State Machine Replication [CL02]). *A state machine replication protocol (SMR) is a protocol for a set of nodes $\{1, ..., n\}$, where all nodes have access to the following interfaces:*

- *SMR.input*(txn)*: Any node can input a transaction* txn *to SMR.*
- $\mathsf{b} \leftarrow$ *SMR.output*$()$*: SMR will output a block, consisting of multiple transactions. We use* $\mathsf{b.rank} = (\mathsf{b.e}, \mathsf{b.r})$ *to denote the rank of* b*, which is a tuple of the block's epoch and round numbers.*

The protocol must satisfy the following properties:

- Safety: *If an honest node has* $\mathsf{b} \leftarrow$ *SMR.output*$()$ *and another honest node has* $\mathsf{b}' \leftarrow$ *SMR.output*$()$*, and* $\mathsf{b.rank} = \mathsf{b}'.\mathsf{rank}$*, then* $\mathsf{b} = \mathsf{b}'$.
- Liveness: *If an honest node invokes SMR.input*(txn)*, every honest node eventually has* $\mathsf{b} \leftarrow$ *SMR.output*$()$ *where* $\mathsf{txn} \in \mathsf{b}$.
- Verifiable: *If an honest node has* $\mathsf{b} \leftarrow$ *SMR.output*$()$*, then* b *can be verified.*

C.2 Protocol: On-chain Randomness

We first describe a simple setup phase for randomness generation in Figure 19, where each node invokes a *reliable broadcast* defined in Definition 13. Recall that our weighted VUF (Figure 3) requires the nodes to locally generate an augmented key pair after the DKG, and use other nodes' augmented public key pair for share verification. As a result, at the beginning the an new epoch, all the nodes will generate augmented key pairs locally and send their augmented public keys via *reliable broadcast*. When a *reliable broadcast* outputs a valid augmented public key, a node records the augmented public key in $apks$. The Agreement property of *reliable broadcast* guarantees that all honest nodes always agree on any node's augmented public key, thus no malicious node can equivocate its augmented public key. Since the latency of *reliable broadcast* is smaller than *SMR*, before *SMR* outputs the first block in the new epoch, *reliable broadcast* already finishes. In other words, the setup phase will not add extra delay to *SMR*.

The generation of randomness for every block is described in Figure 20. When *SMR* outputs a new block, the node generates and multicasts the randomness share by signing VUF on the rank number using its augmented secret key. The node also adds the block to the end of the $block\_queue$, which is a queue structure that holds the blocks output by *SMR* awaiting randomness, and the prefix of blocks that have randomness can proceed to execution.

When receiving a randomness share for rank r, a node does nothing if it already has the randomness for rank r. Otherwise, the node verifies the randomness share against the sender's augmented public key. If the apk is not yet present, the node will asynchronously wait for the apk in a non-blocking manner. If the share is valid, the node stores the share in $shares$ of rank r. Once $shares$ of rank r contain enough randomness shares generated by nodes of weights more than W, the node aggregates the shares to produce the proof and randomness, and add them to map. The node will try to pop and execute the prefix of blocks that have randomness in the block queue, whenever a new block comes or a new randomness is generated.

Inputs:

1. The parameters n, t of the VUF, and a BDH input $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c) \in \mathbb{G}^3 \times \hat{\mathbb{G}}^4$ where $a, b, c \leftarrow\$ \mathbb{F}$.
2. Let $\mathcal{M} \subset [n]$ with $|\mathcal{M}| \leq t$ be the set of corrupt parties. Let $\mathcal{H} = [n] \setminus \mathcal{M}$ be the set of honest parties.

Encryption Key Generation:

3. Use $(g, \hat{g}, h = g^b)$ as the CRS, for $c \leftarrow\$ \mathbb{F}$.
4. Sample one honest party $i' \in \mathcal{H}$ uniformly at random.
5. Compute the encryption keys of honest parties assuming that honest party i' will share h^a as its PVSS secret. This means that these keys as generated as is defined in the PVSS simulation in Figure 10, and that the simulator does not know the corresponding private keys.
   Send the encryption keys of honest parties $\{\mathsf{ek}_i\}_{i \in \mathcal{H}}$ to $\mathcal{A}$.
6. Wait to receive $\{\mathsf{ek}_i\}_{i \in \mathcal{M}}$ from $\mathcal{A}$. Extract $\mathsf{dk}_i \in \mathbb{F}$ for each $i \in \mathcal{M}$ using the PoK extractor.

DKG Simulation:

7. For each honest party $i \in \mathcal{H} \setminus \{i'\}$, generate the PVSS transcript honestly.
8. For honest party i', compute the simulated transcript with h^a as its secret as is described in the PVSS simulator of Figure 10.
9. Send the PVSS transcripts on behalf of every honest party to $\mathcal{A}$. Wait to receive PVSS transcripts from $\mathcal{A}$, if any.
10. Follow the rest of the DKG protocol as per the specification in Figure 12. Let Q be the set of dealers whose transcripts are aggregated by the DKG protocol to generate the VUF key. Since $|\mathcal{M}| \leq t$ and $|Q| > t$ then $Q \cap \mathcal{H} \neq \phi$.
11. Let $a(\cdot)$ be the aggregated polynomial, i.e., $a(x) = \sum_{i \in Q} a_i(x)$. Here $a_i(x)$ is the polynomial shared by party i during the DKG. The shared secret is $h^{a(0)}$.
12. Let us assume that $i' \in Q$, which happens with probability $\geq 1/|\mathcal{H}| > 1/n$. Then the VUF secret key is $a(0) = a + a_H + a_M$ for some $(a_H, a_M) \in \mathbb{F}^2$. Here a_H is the sum of the secrets shared by the dealers in $Q \cap \mathcal{H} \setminus \{i'\}$. Similarly, a_M is the sum of the secrets shared by the dealers in $Q \cap \mathcal{M}$. Clearly, $\mathcal{A}_{\mathrm{BDH}}$ knows a_H since it sampled the secrets of honest parties, and it can also compute a_M since it can *extract* the secrets of all participants in $\mathcal{M}$ using the PoK extractor applied to the secret they share in the PVSS where they are the dealer.
    (This means that $\mathcal{A}_{\mathrm{BDH}}$ does not need to decrypt the shares it receives, which is crucial since it does not know the private keys of $\mathcal{H}$.)
13. As part of the aggregated transcript, $\mathcal{A}_{\mathrm{BDH}}$ learns $\boldsymbol{V} = [g^{a(0)}, g^{a(1)}, \ldots, g^{a(n)}]$ and $\hat{\boldsymbol{V}} = [\hat{g}^{a(0)}, \hat{g}^{a(1)}, \ldots, \hat{g}^{a(n)}]$.
14. (Generating augmented public keys.) In the VUF protocol, each participant needs to broadcast its augmented public key apk_i. $\mathcal{A}_{\mathrm{BDH}}$ generates the augmented public keys of honest parties as in the VUF simulation in Figure 5: For each honest participant $i \in \mathcal{H}$ it samples $u_i \leftarrow\$ \mathbb{F}$. It holds that $u_i = b \cdot r_i$ for some $r_i \in \mathbb{F}$ unknown to $\mathcal{A}_{\mathrm{BDH}}$. It computes $\pi_i = g^{u_i} = h^{r_i}$. Then, since it knows $g^{a(i)}$ it can compute $\mathsf{rk}_i = g^{a(i) u_i} = g^{a(i) \cdot b \cdot r_i} = h^{a(i) r_i}$. The augmented public key is then $\mathsf{apk}_i = (\pi_i, \mathsf{rk}_i)$. $\mathcal{A}_{\mathrm{BDH}}$ broadcasts the augmented public keys of the honest parties, $\{\mathsf{apk}_i\}_{i \in \mathcal{H}}$.

VUF signature simulation:

15. $\mathcal{A}_{\mathrm{BDH}}$ responds to VUF signing queries just as in the VUF simulation of Figure 5:
    (a) Let q_{H} be an upper bound on the number of random oracle queries that $\mathcal{A}$ can make.
    (b) $\mathcal{A}_{\mathrm{BDH}}$ samples a random index $k' \leftarrow\$ [q_{\mathsf{H}}]$.
    (c) On the k-th random oracle query on a message m_k, if $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) \neq \perp$, then $\mathcal{A}_{\mathrm{BDH}}$ returns $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. Otherwise, if $k \neq k'$, then $\mathcal{A}_{\mathrm{BDH}}$ samples $x_k \leftarrow\$ \mathbb{F}$, sets $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^{x_k}$, stores (m_k, x_k), and returns $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. When $k = k'$, $\mathcal{A}_{\mathrm{BDH}}$ returns $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^c$.
    (d) On k-th partial signature query (i, m_k) for each signer $i \in \mathcal{H}$, if $m_k \neq m_{k'}$, $\mathcal{A}_{\mathrm{BDH}}$ outputs $\sigma = (\hat{g}^b)^{x_k / u_i}$. Otherwise, it aborts.

Finally, let ρ be the value $\mathcal{A}$ outputs as the VUF output for $m_{k'}$. $\mathcal{A}_{\mathrm{BDH}}$ then outputs $\rho \cdot e((g^b)^{a_H + a_M}, \hat{g}^{-c})$ as the BDH output.

Fig. 13: Simulation of the unweighted threshold VUF when it is instantiated with a DKG.

Inputs:

1. The parameters w, W of the VUF, and a BDH input $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c) \in \mathbb{G}^3 \times \hat{\mathbb{G}}^4$ where $a, b, c \leftarrow\$ \mathbb{F}$.
2. Let $\mathcal{M} \subset [n]$ with $\sum_{i \in \mathcal{M}} w_i \le w$ be the set of corrupt parties. Let $\mathcal{H} = [n] \setminus \mathcal{M}$ be the set of honest parties.

Encryption Key Generation: (as in the unweighted case)

3. Use $(g, \hat{g}, h = g^b)$ as the CRS, for $c \leftarrow\$ \mathbb{F}$.
4. Sample one honest party $i' \in \mathcal{H}$ uniformly at random.
5. Compute the encryption keys of honest parties assuming that honest party i' will share h^a as its PVSS secret. This means that these keys as generated as is defined in the PVSS simulation in Figure 10, and that the simulator does not know the corresponding private keys.
   Send the encryption keys of honest parties $\{\mathsf{ek}_i\}_{i \in \mathcal{H}}$ to $\mathcal{A}$.
6. Wait to receive $\{\mathsf{ek}_i\}_{i \in \mathcal{M}}$ from $\mathcal{A}$. Extract $\mathsf{dk}_i \in \mathbb{F}$ for each $i \in \mathcal{M}$ using the PoK extractor.

DKG Simulation:

7. For each honest party $i \in \mathcal{H} \setminus \{i'\}$, generate the PVSS transcript honestly.
8. For honest party i', compute the simulated transcript with h^a as its secret as is described in the PVSS simulator of Figure 10.
9. Send the PVSS transcripts on behalf of every honest party to $\mathcal{A}$. Wait to receive PVSS transcripts from $\mathcal{A}$, if any.
10. Follow the rest of the DKG protocol as per the specification in Figure 12. Let Q be the set of dealers whose transcripts are aggregated by the DKG protocol to generate the VUF key. Then $Q \cap \mathcal{H} \neq \phi$.
11. Let $a(\cdot)$ be the aggregated polynomial, i.e., $a(x) = \sum_{i \in Q} a_i(x)$. Here $a_i(x)$ is the polynomial shared by party i during the DKG. The shared secret is $h^{a(0)}$.
12. Let us assume that $i' \in Q$, which happens with probability $\ge 1/|\mathcal{H}| > 1/n$. Then the VUF secret key is $a(0) = a + a_H + a_M$ for some $(a_H, a_M) \in \mathbb{F}^2$. Here a_H is the sum of the secrets shared by the dealers in $Q \cap \mathcal{H} \setminus \{i'\}$. Similarly, a_M is the sum of the secrets shared by the dealers in $Q \cap \mathcal{M}$. Clearly, $\mathcal{A}_{\mathrm{BDH}}$ knows a_H since it sampled the secrets of honest parties, and it can also compute a_M since it can *extract* the secrets of all participants in $\mathcal{M}$ using the PoK extractor applied to the secret they share in the PVSS where they are the dealer.
    (This means that $\mathcal{A}_{\mathrm{BDH}}$ does not need to decrypt the shares it receives, which is crucial since it does not know the private keys of $\mathcal{H}$.)
13. As part of the aggregated transcript, $\mathcal{A}_{\mathrm{BDH}}$ learns $\boldsymbol{V} = [g^{a(0)}, g^{a(1)}, \dots, g^{a(W)}]$ and $\hat{\boldsymbol{V}} = [\hat{g}^{a(0)}, \hat{g}^{a(1)}, \dots, \hat{g}^{a(W)}]$.
14. (Generating augmented public keys.) In the VUF protocol, each participant needs to broadcast an augmented public key apk_i for each share i assigned to it. $\mathcal{A}_{\mathrm{BDH}}$ generates the augmented public keys of honest parties as in the VUF simulation in Figure 5: Denote by $a_{i,1}, \dots, a_{i,w_i}$ the shares that are assigned to participant i. For each honest participant i the simulator $\mathcal{A}_{\mathrm{BDH}}$ samples $u_i \leftarrow\$ \mathbb{F}$. It holds that $u_i = b \cdot r_i$ for some $r_i \in \mathbb{F}$ unknown to $\mathcal{A}_{\mathrm{BDH}}$. The simulator computes $\pi_i = g^{u_i} = h^{r_i}$. Then, for all shares $a_{i,j}$ assigned to i, the simulator interpolates $g^{a(i,j)}$ in the exponent and computes $\mathsf{rk}_{i,j} = g^{a(i,j)u_i} = g^{a(i,j) \cdot b \cdot r_i} = h^{a(i,j)r_i}$. The augmented public key is then $\mathsf{apk}_i = (\pi_i, \mathsf{rk}_1, \dots, \mathsf{rk}_{w_i})$. The simulator broadcasts the augmented public keys $\{\mathsf{apk}_i\}$ of all honest parties.

VUF signature simulation:

15. $\mathcal{A}_{\mathrm{BDH}}$ responds to VUF signing queries just as in the VUF simulation of Figure 6:
    (a) Let q_{H} be an upper bound on the number of random oracle queries that $\mathcal{A}$ can make.
    (b) $\mathcal{A}_{\mathrm{BDH}}$ samples a random index $k' \leftarrow\$ [q_{\mathsf{H}}]$.
    (c) On the k-th random oracle query on a message m_k, if $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) \neq \perp$, then $\mathcal{A}_{\mathrm{BDH}}$ returns $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. Otherwise, if $k \neq k'$, then $\mathcal{A}_{\mathrm{BDH}}$ samples $x_k \leftarrow\$ \mathbb{F}$, sets $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^{x_k}$, stores (m_k, x_k), and returns $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. When $k = k'$, $\mathcal{A}_{\mathrm{BDH}}$ returns $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^c$.
    (d) On k-th partial signature query (i, m_k) for each signer $i \in \mathcal{H}$, if $m_k \neq m_{k'}$, $\mathcal{A}_{\mathrm{BDH}}$ outputs $\sigma = (\hat{g}^b)^{x_k/u_i}$. Otherwise, it aborts.

Finally, let ρ be the value $\mathcal{A}$ outputs as the VUF output for $m_{k'}$. $\mathcal{A}_{\mathrm{BDH}}$ then outputs $\rho \cdot e((g^b)^{a_H + a_M}, \hat{g}^{-c})$ as the BDH output.

Fig. 14: Simulation of the weighted VUF when it is instantiated with a DKG, where the PVSS is based on virtualization. The differences from the simulation of the unweighted setting in Figure 13 are highlighted in gray.

Inputs:

1. The parameters w, W of the VUF, and a BDH input $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c) \in \mathbb{G}^3 \times \hat{\mathbb{G}}^4$ where $a, b, c \leftarrow\$ \mathbb{F}$.
2. Let $\mathcal{M} \subset [n]$ with $\sum_{i \in \mathcal{M}} w_i \le w$ be the set of corrupt parties. Let $\mathcal{H} = [n] \setminus \mathcal{M}$ be the set of honest parties.

Encryption Key Generation: (as in the unweighted case)

3. Use $(g, \hat{g}, h = g^b)$ as the CRS, for $c \leftarrow\$ \mathbb{F}$.
4. Sample one honest party $i' \in \mathcal{H}$ uniformly at random.
5. Compute the encryption keys of honest parties assuming that honest party i' will share h^a as its PVSS secret. This means that these keys as generated as is defined in the PVSS simulation in Figure 10, and that the simulator does not know the corresponding private keys.
   Send the encryption keys of honest parties $\{\mathsf{ek}_i\}_{i \in \mathcal{H}}$ to $\mathcal{A}$.
6. Wait to receive $\{\mathsf{ek}_i\}_{i \in \mathcal{M}}$ from $\mathcal{A}$. Extract $\mathsf{dk}_i \in \mathbb{F}$ for each $i \in \mathcal{M}$ using the PoK extractor.

DKG Simulation:

7. For each honest party $i \in \mathcal{H} \setminus \{i'\}$, generate the PVSS transcript honestly.
8. For honest party i', compute the simulated transcript with h^a as its secret as is described in the PVSS simulator of Figure 11.
9. Send the PVSS transcripts on behalf of every honest party to $\mathcal{A}$. Wait to receive PVSS transcripts from $\mathcal{A}$, if any.
10. Follow the rest of the DKG protocol as per the specification in Figure 12. Let Q be the set of dealers whose transcripts are aggregated by the DKG protocol to generate the VUF key. Then $Q \cap \mathcal{H} \ne \phi$.
11. Let $a(\cdot)$ be the aggregated polynomial, i.e., $a(x) = \sum_{i \in Q} a_i(x)$. Here $a_i(x)$ is the polynomial shared by party i during the DKG. The shared secret is $h^{a(0)}$.
12. Let us assume that $i' \in Q$, which happens with probability $\ge 1/|\mathcal{H}| > 1/n$. Then the VUF secret key is $a(0) = a + a_H + a_M$ for some $(a_H, a_M) \in \mathbb{F}^2$. Here a_H is the sum of the secrets shared by the dealers in $Q \cap \mathcal{H} \setminus \{i'\}$. Similarly, a_M is the sum of the secrets shared by the dealers in $Q \cap \mathcal{M}$. Clearly, $\mathcal{A}_{\mathrm{BDH}}$ knows a_H since it sampled the secrets of honest parties, and it can also compute a_M since it can *extract* the secrets of all participants in $\mathcal{M}$ using the PoK extractor applied to the secret they share in the PVSS where they are the dealer.
    (This means that $\mathcal{A}_{\mathrm{BDH}}$ does not need to decrypt the shares it receives, which is crucial since it does not know the private keys of $\mathcal{H}$.)
13. As part of the aggregated transcript, $\mathcal{A}_{\mathrm{BDH}}$ learns $\boldsymbol{V} = [g^{a(0)}, g^{a(1)}, \ldots, g^{a(W)}]$ and $\hat{\boldsymbol{V}} = [\hat{g}^{a(0)}, \hat{g}^{a(1)}, \ldots, \hat{g}^{a(W)}]$.
14. (Generating augmented public keys.) In the VUF protocol, each participant needs to broadcast an augmented public key apk_i for each share i assigned to it. $\mathcal{A}_{\mathrm{BDH}}$ generates the augmented public keys of honest parties as in the VUF simulation in Figure 5: Denote by $a_{i,1}, \ldots, a_{i,w_i}$ the shares that are assigned to participant i. For each honest participant i the simulator $\mathcal{A}_{\mathrm{BDH}}$ samples $u_i \leftarrow\$ \mathbb{F}$. It holds that $u_i = b \cdot r_i$ for some $r_i \in \mathbb{F}$ unknown to $\mathcal{A}_{\mathrm{BDH}}$. The simulator computes $\pi_i = g^{u_i} = h^{r_i}$. Then, for all shares $a_{i,j}$ assigned to i, the simulator interpolates $g^{a(i,j)}$ in the exponent and computes $\mathsf{rk}_{i,j} = g^{a(i,j)u_i} = g^{a(i,j) \cdot b \cdot r_i} = h^{a(i,j)r_i}$. The augmented public key is then $\mathsf{apk}_i = (\pi_i, \mathsf{rk}_1, \ldots, \mathsf{rk}_{w_i})$. The simulator broadcasts the augmented public keys $\{\mathsf{apk}_i\}$ of all honest parties.

VUF signature simulation:

15. $\mathcal{A}_{\mathrm{BDH}}$ responds to VUF signing queries just as in the VUF simulation of Figure 6:
    (a) Let q_{H} be an upper bound on the number of random oracle queries that $\mathcal{A}$ can make.
    (b) $\mathcal{A}_{\mathrm{BDH}}$ samples a random index $k' \leftarrow\$ [q_{\mathsf{H}}]$.
    (c) On the k-th random oracle query on a message m_k, if $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) \ne \perp$, then $\mathcal{A}_{\mathrm{BDH}}$ returns $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. Otherwise, if $k \ne k'$, then $\mathcal{A}_{\mathrm{BDH}}$ samples $x_k \leftarrow\$ \mathbb{F}$, sets $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^{x_k}$, stores (m_k, x_k), and returns $\mathsf{H}_{\hat{\mathbb{G}}}(m_k)$. When $k = k'$, $\mathcal{A}_{\mathrm{BDH}}$ returns $\mathsf{H}_{\hat{\mathbb{G}}}(m_k) = \hat{g}^c$.
    (d) On k-th partial signature query (i, m_k) for each signer $i \in \mathcal{H}$, if $m_k \ne m_{k'}$, $\mathcal{A}_{\mathrm{BDH}}$ outputs $\sigma = (\hat{g}^b)^{x_k/u_i}$. Otherwise, it aborts.

Finally, let ρ be the value $\mathcal{A}$ outputs as the VUF output for $m_{k'}$. $\mathcal{A}_{\mathrm{BDH}}$ then outputs $\rho \cdot e((g^b)^{a_H + a_M}, \hat{g}^{-c})$ as the BDH output.

Fig. 15: Simulation of the weighted VUF when it is instantiated with a DKG, where the PVSS uses one public key per participant. The differences from the simulations in Figures 13 and 14 are highlighted in gray.

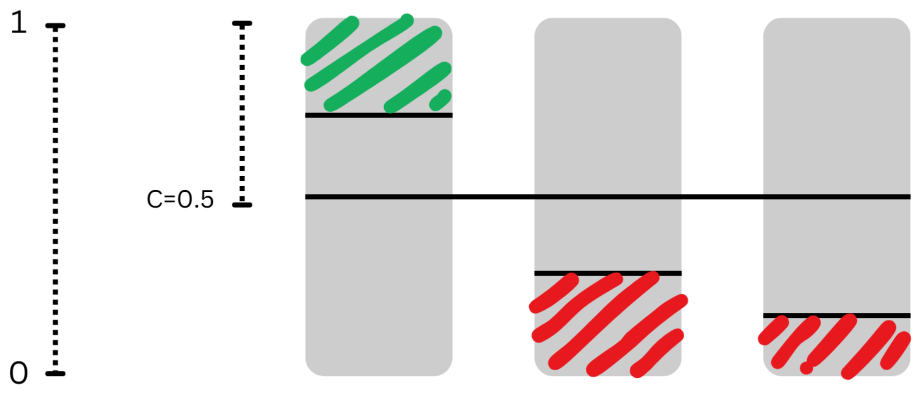


Fig. 18: Illustration of rounding. Each rectangle corresponds to the area between 0 and 1. The blue line shows the fractional part of w_i/B. When rounding up the resulting weight of the validator gains the area above this line (colored in green). When rounding down the weight of the validator loses the area below this line (colored red). The black line that goes all across the drawing corresponds to the value of c, and is the threshold deciding whether to round down or up.

public parameters: VUF public parameters crs
public input: decryption key dk_i of node i
local variables: a vector $apks$ that stores the augmented public keys received from all nodes,

```
1: upon epoch e starts due to SMR outputs DKG trx do
2:     $(\mathsf{sk}_i, (\mathsf{pk}_j)_{j\in[n]}, \mathsf{pk}) \leftarrow \mathsf{PVSS.DecryptShare}_{\mathsf{pp}}(\mathsf{trx}, i, \mathsf{dk}_i)$
3:     $(\mathsf{ask}_i, \mathsf{apk}_i) \leftarrow \mathsf{VUF.AugmentKeyPair}(\mathsf{sk}_i, \mathsf{pk}_i)$
4:     reliable broadcast $\langle \mathtt{AugPK}, \mathsf{apk}_i \rangle$
5: upon reliable broadcast invoked by node $j$ outputs $\langle \mathtt{AugPK}, \mathsf{apk}_j \rangle$ do
6:     if $apks[j] = \perp$ and $\mathsf{VUF.PubKeyVerify}(\mathsf{pk}_j, \mathsf{apk}_j) = 1$ then
7:         $apks[j] \leftarrow \mathsf{apk}_j$
```

Fig. 19: The Setup Phase for On-chain Randomness.

– public parameters: VUF reconstruction threshold W
– local variables:
  - a vector $apks$ that stores the augmented public keys received from all nodes
  - a hashMap $shares$ that maps the rank number to the set of received randomness shares
  - a hashMap map that maps the rank number to the generated randomness
  - a queue $block\_queue$ that stores ordered blocks

```
1:  upon b ← SMR.output() do
2:      $\sigma_i \leftarrow \mathsf{VUF.ShareSign}(\mathsf{ask_i}, \mathsf{b.rank})$
3:      send $\langle \texttt{RandShare}, \mathsf{b.rank}, \sigma_i \rangle$ to all nodes
4:      $block\_queue.push\_back(\mathsf{b})$
5:      try_dequeue()
6:  upon receiving $\langle \texttt{RandShare}, r, \sigma_j \rangle$ from node $j$ do
7:      if $map[r] \neq \perp$ then
8:          return
9:      wait until $apks[j] \neq \perp$
10:     if $\mathsf{VUF.ShareVerify}(apks[j], r, \sigma_j) = 1$ then
11:         $shares[r] \leftarrow shares[r] \cup \{(j, \sigma_j)\}$
12:     if $\sum_{(j,\cdot) \in shares[r]} w_j > \mathsf{W}$ then
13:         $(T, \sigma) \leftarrow \mathsf{VUF.ShareAggregate}(r, \{j | (j, \cdot) \in shares[r]\}, \{\sigma \mid (\cdot, \sigma) \in shares[r]\})$
14:         $\mathsf{vuf} \leftarrow \mathsf{VUF.Derive}(\mathsf{T}, \sigma)$
15:         $map[r] \leftarrow hash(\mathsf{vuf})$
16:         try_dequeue()
17: upon try_dequeue() do
18:     while $b \leftarrow block\_queue.pop\_front()$ do
19:         if $map[b.\mathsf{rank}] \neq \perp$ then
20:             $b.\mathsf{randomness} \leftarrow map[b.\mathsf{rank}]$
21:             execute $b$ and persist the execution result
22:         else
23:             $block\_queue.push\_front(\mathsf{b})$
24:             break
```

Fig. 20: On-chain Randomness using Weighted VUF.