Distributed Randomness using Weighted VUFs

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. Shared randomness in blockchain can expand its support for randomized applications and can also help strengthen its security. Many existing blockchains rely on external randomness beacons for shared randomness, but this approach reduces fault tolerance, increases latency, and complicates application development. An alternate approach is to let the blockchain validators generate fresh shared randomness themselves once for every block. We refer to such a design as the *on-chain* randomness.
In this paper, we design an efficient on-chain randomness protocol for Byzantine fault-tolerance based Proof-of-Stake blockchains with weighted validators. A key component of our protocol is a weighted verifiable unpredictable function (VUF). The notable feature of our weighted VUF is that the computation and communication costs of parties are independent of their weight. This is crucial for scalability of on-chain randomness where we repeatedly evaluate the weighted VUF in quick succession. We also design a new scalable publicly verifiable secret sharing (PVSS) scheme with aggregatable transcript and use it to design a distributed key generation (DKG) protocol for our VUF. We implemented our schemes on top of Aptos, a proof-of-stake blockchain deployed in production, conducted an end-to-end evaluation with 112 validators and a total weight of up to 4053. In this setup, our on-chain randomness protocol adds only 133 milliseconds of latency compared to a protocol without randomness. We also demonstrate the performance improvements of our design through rigorous comparison with baseline methods.

1 Introduction

A major limitation of existing blockchains is that they primarily support deterministic computation [48,61]. This is fundamental as the security of blockchains crucially relies on the replicability of the blockchain state. However, many natural applications are inherently randomized. A 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 verifiable random values allows leader-based blockchain protocols to select leaders in a fair, randomized fashion. For asynchronous blockchain protocols, randomization is essential to achieve liveness due to FLP impossibility [35]. Furthermore, integrating randomness into transaction ordering reduces opportunities for maximal extractable value (MEV) attacks by minimizing the advantages of targeted transaction ordering within a block [25].

To support these applications, blockchains rely on shared unpredictable randomness, typically from an external randomness beacon [31,2]. Intuitively, the randomness beacon outputs unpredictable random values at periodic intervals on which the blockchain validators/miners agree. Whenever an application calls for randomness, it uses the agreed-upon output from the randomness beacon.

Limitations of using external randomness beacons. Reliance on external randomness beacons introduces several security and efficiency issues. 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 the corruption of 9 of its 18 members [31]. 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 issue with external randomness beacons is their latency. For example, Drand emits a new random value every 30 seconds, whereas blockchains might produce blocks more frequently (multiple blocks every

second in [3,11]). Waiting for an external beacon causes a delay for protocols and transactions that need fresh randomness.

Lastly, using an external beacon complicates blockchain and application development. Since the external beacon produces randomness asynchronously to the blocks, it forces each randomness call to navigate a two-transaction process for security – first committing to a future, undisclosed beacon output and then consuming the committed beacon output. 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 is to let the blockchain validators generate the shared randomness, once for each block, immediately after the block has been ordered. The shared randomness must be unaffected by which subset of validators computed it, must remain hidden until the block is ordered, and must also be verifiable, i.e., everyone can confirm its validity, thereby preventing corrupt validators from altering the output value. We refer to shared randomness with all these properties as the *on-chain* randomness.

On-chain randomness immediately addresses the above-mentioned issues of external randomness beacons: Since the blockchain validators themselves generate the shared randomness, the fault-tolerant assumption and the availability guarantees of the blockchain remain intact. Moreover, a new shared randomness for every block immediately addresses the latency and synchronization issues. Also, since the random output 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 retrieving the randomness seed embedded in the block and, if necessary, generating additional randomness using a pseudorandom function.

Our contributions. We design an on-chain randomness protocol for Byzantine fault tolerance (BFT)-based Proof-of-Stake (PoS) blockchains [17,41,51,3]. 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 (weight), which is also the threat model we consider for designing on-chain randomness.

A known approach for on-chain randomness in the *unweighted* setting is based on a threshold verifiable unpredictable function (VUF), which are essentially unique threshold signatures [47,30]. In this approach, the keys for computing the VUF are secret-shared among the validators. For each block, validators evaluate the VUF on a unique input (say block height) and use the hash (modeled as a random oracle) of the VUF output as the randomness for the block. The threshold property ensures that the VUF output is available only when more than a threshold fraction of the validators contribute to it.

A natural approach for on-chain randomness with weighted validators is to use a *weighted* VUF, where only a set of validators with a combined minimum weight can compute the VUF output. Unfortunately, no such efficient weighted VUF constructions are known. A naïve approach is to let each validator with weight w emulate w virtual validators and use an unweighted threshold VUF with a threshold of one-third of the total weight. This approach is also known as *virtualization*. However, in this approach, each validator incurs a cost proportional to its weight, and the weighted VUF evaluation cost is proportional to the total weight. These costs can be prohibitive, as we seek to frequently evaluate the weighted VUF once for every block.

The main contribution of this paper is a new weighted VUF protocol where the per validator computation and communication cost is constant and independent of its weight. Hence, the total communication cost is linear in the number of validators that contribute to the VUF output. This results in significant performance improvement compared to virtualization (see §6). For example, our new weighted VUF reduces the communication costs by a factor 7× and 34× for a total weight of 821 and 4053, respectively.

We also present a distributed key generation (DKG) protocol to set up the keys for our VUF scheme in a decentralized manner. Unlike the VUF, the performance of our DKG depends on the total weight of the parties. Since the total weight can be large (a few thousand), existing DKG schemes that only scale to generate a few hundred shares are unsuitable for our case [40,29]. Thus, we design a new DKG protocol with various optimizations to meet our scalability requirements. A key component of the new DKG protocol is a new publicly verifiable secret sharing (PVSS) scheme. The sharing phase of the new PVSS scheme is non-

interactive, its transcripts are aggregatable, and the scheme overall is more efficient than prior PVSS schemes with similar properties [18]. We believe, our new PVSS scheme and our new DKG will be of independent interest.

We implement our weighted VUF-based on-chain randomness, along with our DKG protocol, atop the open-source codebase of Aptos [3], a PoS blockchain deployed in production. Our geo-distributed evaluation with 112 validators with a total weight of up to 4053, corroborates our efficiency and scalability. For example, with these system parameters, generating on-chain randomness for a block only adds 133 milliseconds of latency. We also present a detailed performance comparison with the folklore virtualization-based approach.

2 Preliminaries

We use κ to denote the security parameter. Throughout the paper, we will use "$\leftarrow$" for probabilistic assignment and ":=" for deterministic assignment. For any set S, we use $s \leftarrow\$ S$ to indicate that s is sampled uniformly randomly from S. We use $|S|$ to denote the size of a set S. 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 $\boldsymbol{X} \cdot \boldsymbol{Y} := [x_1 \cdot y_1, \ldots, x_k \cdot y_k]$.

Let GGen be a group generation algorithm that on input 1^κ outputs the description of a prime order groups $(\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T)$. The description contains the prime order p, generators $g \in \mathbb{G}$, $\hat{g} \in \mathbb{G}$, a description of the group operation, and a bilinear pairing operation: $e : \mathbb{G} \times \hat{\mathbb{G}} \to \mathbb{G}_T$ defined below.

Definition 1 (Bilinear Pairing). *Let $(\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T, p, g, \hat{g}) \leftarrow \mathsf{GGen}(1^\kappa)$. A bilienar-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*

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

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

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

We base the security of our protocols on the bilinear Diffie-Hellman (BDH) assumption in $(\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T)$:

Assumption 1 (BDH)**.** *We say that the bilinear Diffie-Hellman (BDH) assumption holds, if for all PPT adversaries $\mathcal{A}$, the following advantage is negligible:*

$$\Pr\left[\mathcal{A}(g, g^a, g^b, \hat{g}^a, \hat{g}^b, \hat{g}^c) = e(g, \hat{g})^{abc} \;\middle|\; \begin{array}{r}(\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T, p, g, \hat{g}) \leftarrow \mathsf{GGen}(1^\kappa) \\ a, b, c \leftarrow\$ \mathbb{F}\end{array}\right] = \varepsilon_{\mathsf{bdh}}.$$

2.1 System Model and Efficiency Metrics

Threat model. We consider a system of n parties, each representing a blockchain validator, where party i has 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 parties. We consider a *static* adversary $\mathcal{A}$ that can corrupt parties with a 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 DKG or VUF protocols but rather a requirement from the blockchain consensus protocol.) We assume a public key infrastructure (PKI) for the PVSS scheme and work in the *random oracle model*, which is needed due to the use of the Fiat-Shamir [34] heuristic to achieve non-interactivity for the weighted PVSS. For our DKG, we assume a partially synchronous network [32] and assume that parties have access to a total order broadcast channel (see Definition 8). For on-chain randomness, the network can be asynchronous.

Efficiency metrics. It is well known that in elliptic curve groups, bilinear pairings are more expensive than exponentiations, and repeated exponentiations are more expensive than multi-exponentiations (i.e., computing the multiplication of multiple exponentiations) [1]. For example, for the BLS12381 elliptic curve, and running on a 10-core 2021 Apple M1 Max, computing a bilinear pairing is 6.75× and 3.75× more expensive than computing an exponentiation in $\mathbb{G}$ and $\hat{\mathbb{G}}$, respectively. A 256 element *multi-exponentiation* is 24× and 18.8× faster than computing 256 individual exponentiations, in $\mathbb{G}$ and $\hat{\mathbb{G}}$, respectively.

Based on these benchmarks, we aim to reduce the number of pairings that need to be computed. In particular, we aim to replace a computation of n pairings with one pairing and an n-wide multi-exponentiation in $\mathbb{G}$ or $\hat{\mathbb{G}}$. For large n, this can improve run time by nearly two orders of magnitude.

2.2 Weighted VUF Definitions

We start with the security of weighted VUF with an idealized key generation algorithm. The advantage of this approach, rather than immediately presenting the VUF keyed by a DKG, is that it lets us focus on the security of the VUF and avoid any nuances due to DKG. Our security definitions are inspired by the threshold signature definitions in [16].

Definition 2 (Weighted VUF). *A $(n, \boldsymbol{w}, w)$-weighted VUF scheme with n parties with weights $\boldsymbol{w} = [w_1, \ldots, w_n]$, a threshold $w < \sum_{i\in[n]} w_i$, and a finite message space $\mathcal{M}$ is a tuple of polynomial time algorithms* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Eval}, \mathsf{ShareEval}, \mathsf{ShareVerify}, \mathsf{Verify}, \mathsf{Derive})$ *where:*

- $\mathsf{Setup}(1^\kappa) \to \mathsf{pp}$: *the setup algorithm takes as input the security parameter κ and outputs public parameters* pp *for the scheme. We assume that all algorithms implicitly take* pp *as input.*
- $\mathsf{KeyGen}(\mathsf{pp}) \to \mathsf{pk}, \mathsf{sk}, \{\mathsf{vk}_i, \mathsf{sk}_i\}_{i\in[n]}$: *the key generation algorithm outputs a public key* pk*, a secret key* sk*, verification and secret keys $\{\mathsf{vk}_i, \mathsf{sk}_i\}_{i\in[n]}$ for each party. The j-th party receives $\mathsf{pk}, \{\mathsf{vk}_i\}_{i\in[n]}, \mathsf{sk}_j$.*
- $\mathsf{Eval}(\mathsf{sk}, m) \to \rho$: *The evaluation algorithm is a deterministic algorithm that takes as input the secret key* sk*, a message m, and outputs ρ.*
- $\mathsf{ShareEval}(\mathsf{sk}_i, m) \to \sigma_i$: *The share evaluation algorithm takes as input a secret key share* sk_i*, a message m, and outputs a VUF share σ_i.*
- $\mathsf{ShareVerify}(\mathsf{vk}_i, m, \sigma_i) \to 0$ *or* 1 : *The share verification takes as input a verification key* vk_i*, a message m, a VUF share σ_i, and outputs either 1 (accept) or 0 (reject).*
- $\mathsf{Derive}(S, \{\sigma_i, \mathsf{vk}_i\}_{i\in S}, m) \to (\rho, \pi)$ *or* $\perp$: *The derivation algorithm takes as input a set $S \subseteq [n]$ of parties with $\sum_{i\in S} w_i > w$, their VUF shares and verification keys $\{\sigma_i, \mathsf{vk}_i\}_{i\in S}$, a message m, and outputs the VUF output ρ and a proof π or $\perp$.*
- $\mathsf{Verify}(\mathsf{pk}, m, \rho, \pi) \to 0$ *or* 1 : *The verification algorithm takes as input a public key* pk*, a message m, a VUF output ρ along with proof π, and outputs either 1 (accept) or 0 (reject).*

The weighted VUF must ensure *correctness*, *uniqueness*, and *unpredictability* properties we define next. Intuitively, the correctness property ensures that (i) honestly generated VUF shares are accepted at other honest parties, and (ii) given valid VUF shares from parties with a combined weight higher than w, the Derive algorithm successfully outputs the VUF output. These properties guarantee that honest parties can compute the VUF output successfully.

Definition 3 (Correctness). *For all security parameters $\kappa \in \mathbb{N}$, all n, all weight distribution $\boldsymbol{w}$, all allowable thresholds w, a $(n, \boldsymbol{w}, w)$-weighted VUF scheme is correct if for all subsets $S \subseteq [n]$ with $\sum_{i\in S} w_i > w$, all messages m, and for $\mathsf{pk}, \mathsf{sk}, \{\mathsf{vk}_i, \mathsf{sk}_i\}_{i\in[n]} \leftarrow \mathsf{KeyGen}(\mathsf{pp})$, the following holds:*

- $\Pr[\mathsf{ShareVerify}(\mathsf{vk}_i, m, \mathsf{ShareEval}(\mathsf{sk}_i, m)) = 1] = 1$
- $\Pr[\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Derive}(S, \{\sigma_i, \mathsf{vk}_i\}_{i\in S}, m)) = 1$
  $: \sigma_i := \mathsf{ShareEval}(\mathsf{sk}_i, m)\ \forall i \in S] = 1$

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

Game $\mathsf{UP\text{-}CMA}^{\mathcal{A}}(1^\kappa, n, \boldsymbol{w}, w)$:

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa)$
$\mathcal{C}, \mathcal{S} \leftarrow \mathcal{A}(\mathsf{pp})$ // $\mathcal{C}, \mathcal{S} \subseteq [n]$
if $\sum_{i \in \mathcal{C} \cup \mathcal{S}} w_i \geq w$: **return** 0
$\mathsf{pk}, \mathsf{sk}, \{\mathsf{vk}_i, \mathsf{sk}_i\}_{i \in [n]} \leftarrow \mathsf{KeyGen}(\mathsf{pp})$
Let $\mathcal{H} := [n] \setminus (\mathcal{C} \cup \mathcal{S})$
$\mathsf{inp} := \mathsf{pp}, \mathsf{vk}, \{\mathsf{pk}_i\}_{i \in [n]}, \{\mathsf{sk}_i\}_{i \in \mathcal{C}}$
// $Q[m]$, initially $\{\}$ for all messages.
$(m^*, \rho^*) \leftarrow \mathcal{A}^{\text{SEVAL}}(\mathsf{inp})$
if $Q[m^*] \subseteq \mathcal{S} \wedge \rho^* = \mathsf{Eval}(\mathsf{sk}, m^*)$:
    return 1
return 0

Oracle $\text{SEVAL}(i, m)$:

if $i \in \mathcal{H} \cup \mathcal{S}$:
    $Q[m] := Q[m] \cup \{i\}$
    $\sigma_i \leftarrow \mathsf{ShareEval}(\mathsf{sk}_i, m)$
    return σ_i
return $\perp$

Fig. 1: The unpredictability security game $\mathsf{UP\text{-}CMA}^{\mathcal{A}}$ for the weighted VUF.

We formally define the uniqueness property next. Intuitively, the uniqueness property ensures that for every input m, there exists only one output $\rho = \mathsf{Eval}(\mathsf{sk}, m)$ that verifies as per Verify algorithm.

Definition 4 (Uniqueness). *For all security parameters $\kappa \in \mathbb{N}$, all n, all weight distribution $\boldsymbol{w}$, all allowable thresholds w, a $(n, \boldsymbol{w}, w)$-weighted VUF scheme ensures uniqueness if for all messages m, and for all $\mathsf{pk}, \mathsf{sk}, \{\mathsf{vk}_i, \mathsf{sk}_i\}_{i \in [n]} \leftarrow \mathsf{KeyGen}(\mathsf{pp})$, and for any (ρ, π) the following holds:*

$$\Pr\left[\mathsf{Verify}(\mathsf{pk}, m, \rho, \pi) = 1 \wedge \mathsf{Eval}(\mathsf{sk}, m) \neq \rho\right] = 0$$

Unpredictability means that an adversary $\mathcal{A}$ corrupting parties with combined weight at most w cannot predict the VUF output for any input on which it was not queried earlier. We formalize this with the $\mathsf{UP\text{-}CMA}^{\mathcal{A}}$ game in Figure 1. Our unpredictability definition is analogous to the TS-UF-0 unforgeability definition of non-interactive threshold signatures from [8].

Definition 5 (Unpredictability). *For all security parameter $\kappa \in \mathbb{N}$, n, weight distribution $\boldsymbol{w}$, allowable thresholds w, we say that a $(n, \boldsymbol{w}, w)$-weighted VUF is unpredictable under chosen message attacks if for all probabilistic polynomial time (PPT) adversaries $\mathcal{A}$,* $\Pr[\mathsf{UP\text{-}CMA}^{\mathcal{A}}(1^\kappa, n, \boldsymbol{w}, w) = 1] = \mathsf{negl}(\kappa)$.

When all parties have equal weights, Definition 2 defines threshold VUF. Similarly, the special case of $n = 1$ and $w = 0$ is the single-server VUF.

Weighted VUF with Distributed Key Generation (DKG). In Definition 2, we defined the security of the weighted VUF with an idealized key generation. The security definition with a DKG is almost identical, except parties use an interactive DKG protocol to generate the VUF keys. More precisely, parties run DKG instead of KeyGen in Definition 2, where the DKG is defined as below:

$\mathsf{DKG}(\mathsf{pp}) \rightarrow \mathsf{pk}, \mathsf{sk}, \{\mathsf{vk}_i, \mathsf{sk}_i\}_{i \in [n]}$: The DKG protocol among n parties with weights $\boldsymbol{w}$, and a threshold w outputs a public key pk, secret key sk, and verification and secret keys $\{\mathsf{vk}_i, \mathsf{sk}_i\}_{i \in [n]}$ of each party. At the end of the protocol, each party i receives $(\mathsf{pk}, \{\mathsf{vk}_i\}_{i \in [n]}, \mathsf{sk}_i)$.

The correctness, uniqueness, and unpredictability definitions of the $(n, \boldsymbol{w}, w)$-weighted VUF scheme with a DKG are identical to Definitions 3 to 5, respectively, except in the $\mathsf{UP\text{-}CMA}^{\mathcal{A}}$ game the game interacts with $\mathcal{A}$ on behalf of parties in $\mathcal{H}$ to run the $\mathsf{DKG}(\mathsf{pp})$ protocol to generate keys.

3 A Weighted VUF

We now describe our weighted VUF scheme, where per-party computation and communication costs are independent of their weight, but the public key size and VUF derivation costs depend on the total weight. For better exposition, we present the VUF constructions in three steps. First, we present a single-party VUF, followed by the unweighted threshold VUF and then the weighted VUF.

$\underline{\mathsf{Setup}(1^\kappa)} \to \mathsf{pp}$
$\mathsf{H} : \{0,1\}^* \to \hat{\mathbb{G}}$
return $(h \in \mathbb{G}, \mathsf{H})$

$\underline{\mathsf{KeyGen}(\mathsf{pp})} \to \mathsf{pk}, \mathsf{sk}, \{\mathsf{vk}_1, \mathsf{sk}_1\}$
Sample $a, r \leftarrow\!\$\ \mathbb{F}$;
$\mathsf{sk}_1 := r$; $\mathsf{vk}_1 := (h^r, h^{ar})$
$\mathsf{sk} := h^a$; $\mathsf{pk} := \{\mathsf{vk}_1\}$
return $\mathsf{pk}, \mathsf{sk}, \{\mathsf{vk}_1, \mathsf{sk}_1\}$

$\underline{\mathsf{Eval}(\mathsf{sk} = h^a, m)} \to \rho$:
return $\rho := e(h^a, \mathsf{H}(m))$

$\underline{\mathsf{ShareEval}(\mathsf{sk}_1 = r, m)} \to \sigma$:
return $\mathsf{H}(m)^{1/r}$

$\underline{\mathsf{ShareVerify}(\mathsf{vk}_1 = (h^r, \cdot), m, \sigma)} \to 0/1$:
// checks $\mathsf{H}(m)^{1/r}$ is computed correctly
// i.e, $e(h^r, \mathsf{H}(m)^{1/r}) = e(h, \mathsf{H}(m))$
assert $e(h^r, \sigma) = e(h, \mathsf{H}(m))$

$\underline{\mathsf{Derive}(\{1\}, \{\sigma, \mathsf{vk}_1 = (\cdot, h^{ar})\}, m)} \to (\rho, \pi)/\perp$:
assert $\mathsf{ShareVerify}(\mathsf{vk}_1, m, \sigma)$
return $e(h^{ar}, \sigma), \pi := \sigma$
// $= e(h^{ar}, \mathsf{H}(m)^{1/r}) = e(h^a, \mathsf{H}(m))$

$\underline{\mathsf{Verify}(\mathsf{pk} = \{\mathsf{vk}_1\}, m, \rho, \pi = \sigma)} \to 0/1$:
$(\rho', \pi') := \mathsf{Derive}(\{1\}, \{\sigma, \mathsf{vk}_1\}, m)$
return $\rho = \rho'$

Fig. 2: Single-party VUF, i.e., a $(1, [1], 0)$-weighted VUF.

3.1 Setting the Ground - a Single-Party VUF

As the name suggests, in a single-party VUF, only a single party knows the VUF secret key and is responsible for VUF evaluation. This VUF illustrates the core ideas that are used in our threshold and weighted VUFs. We summarize the single-party VUF in Figure 2 and describe it next.

Setup and key generation. The public parameters consists of a generator $h \in \mathbb{G}$ and a random oracle $\mathsf{H} : \{0,1\}^* \to \hat{\mathbb{G}}$. The VUF secret key $\mathsf{sk} := h^a$ for a uniformly random $a \leftarrow\!\$\ \mathbb{F}$. The secret key the party receives is $\mathsf{sk}_1 := r \leftarrow\!\$\ \mathbb{F}$, and the corresponding verification key is $\mathsf{pk} = \mathsf{vk}_1 := (h^r, h^{ar})$.

VUF evaluation. For message space $\mathcal{M}$, let $\mathsf{H} : \mathcal{M} \to \hat{\mathbb{G}}$ be a hash function modeled as a random oracle. The VUF output is defined as $\rho = e(h^a, \mathsf{H}(m))$. Note that the VUF output cannot verified directly. Thus, the party computes a verifiable value $\sigma := \mathsf{H}(m)^{1/r}$ using sk_1 and outputs σ.

VUF verification and derivation. Any external verifier $\mathcal{V}$ with access to vk_1 can verify σ using the $\mathsf{ShareVerify}$ algorithm. Precisely, $\mathsf{ShareVerify}$ checks whether the exponents in σ and h^r are multiplicative inverses of each other. Next, Derive computes the VUF output ρ as a *deterministic* function of σ; this is also a verification of the VUF output.

Analysis. Note that the single-party VUF is a degenerate case of a $(n, \boldsymbol{w}, w)$-weighted VUF with $n = 1$, $\boldsymbol{w} = [1]$, and $w = 0$, and therefore its security follows the security of our weighted VUF that we prove in §3.4.

3.2 An Unweighted Threshold VUF

Let n be the total number of parties and t be the threshold, i.e., $t + 1$ VUF shares are needed in order to compute the VUF output. We summarize the construction in Figure 3 and describe it next.

Setup and key generation. The public parameters of the threshold VUF are the same as that of the single-party VUF, i.e., a generator $h \in \mathbb{G}$, and a hash function H. The secret key of party i is $\mathsf{sk}_i := r_i \leftarrow\!\$\ \mathbb{F}$. Let $a(x)$ be a random polynomial of degree t. Let $a = a(0)$ and $a_i := a(i)$ for all $i \in [n]$. Then, the public key of party i is $\mathsf{vk}_i := (h^{r_i}, h^{a_i r_i})$. Again, similar to single party case, the VUF output on input m is $e(h^a, \mathsf{H}(m))$.

VUF computation and verification. The per-party VUF evaluation algorithm is identical to that of single-party VUF, except for each party i using its private key $\mathsf{sk}_i = r_i$ to compute $\sigma_i := \mathsf{H}(m)^{1/r_i}$. Similarly, the VUF output σ_i of party i can be verified using the $\mathsf{ShareVerify}$ algorithm that is similar to that of the single-party VUF.

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

$$\rho := \prod_{i \in S} e\left((h^{a_i r_i})^{\ell_i}, \sigma_i\right); \tag{1}$$

where ℓ_i is the i-th Lagrange coefficient for the set S.

$\underline{\mathsf{Setup}(1^\kappa, n, t)} \to \mathsf{pp}$
$\mathsf{H} : \{0,1\}^* \to \hat{\mathbb{G}}$
return $(h \in \mathbb{G}, \mathsf{H}, n, t)$

$\underline{\mathsf{KeyGen}(\mathsf{pp})} \to \mathsf{pk}, \mathsf{sk}, \{\mathsf{sk}_i, \mathsf{vk}_i\}_{i\in[n]}$:
Sample $a \leftarrow\$ \mathbb{F}$;
$(a_i)_{i\in[n]} \leftarrow \mathsf{ShamirShare}(a, t, n)$
$\mathsf{sk}_i := r_i \leftarrow\$ \mathbb{F}$; $\mathsf{vk}_i := (h^{r_i}, h^{r_i \cdot a_i})$
$\mathsf{sk} := h^a$; $\mathsf{pk} := \{\mathsf{vk}_i\}_{i\in[n]}$
return $\mathsf{pk}, \mathsf{sk}, \{\mathsf{sk}_i, \mathsf{vk}_i\}_{i\in[n]}$

$\underline{\mathsf{Eval}(\mathsf{sk} = h^a, m)} \to \rho$:
return $\rho := e(h^a, \mathsf{H}(m))$

$\underline{\mathsf{ShareEval}(\mathsf{sk}_i = r_i, m)} \to \sigma_i$:
// As for a single-party
return $\mathsf{H}(m)^{1/r_i}$

$\underline{\mathsf{ShareVerify}(\mathsf{vk}_i = (h^{r_i}, \cdot), m, \sigma_i)} \to 0/1$:
// As for a single-party
return $e(h^{r_i}, \sigma_i) = e(h, \mathsf{H}(m))$

$\underline{\mathsf{Derive}(S, \{\sigma_i, \mathsf{vk}_i\}_{i\in S}, m)} \to (\rho, \pi)/\perp$:
parse $\mathsf{vk}_i = (\cdot, h^{a_i r_i})$
assert $\mathsf{ShareVerify}(\mathsf{vk}_i, m, \sigma_i) \quad \forall i \in S$
$(\ell_i)_{i\in S} := \mathsf{LagrangeCoefficients}(S)$
$\rho := \prod_{i\in S} e((h^{a_i r_i})^{\ell_i}, \sigma_i)$; $\pi := (S, \{\sigma_i\}_{i\in S})$
return (ρ, π)

$\underline{\mathsf{Verify}(\mathsf{pk} = \{\mathsf{vk}_i\}_{i\in[n]}, m, \rho, \pi)} \to 0/1$:
parse $\pi = (S, \{\sigma_i\}_{i\in S})$
$(\rho', \cdot) := \mathsf{Derive}(S, \{\sigma_i, \mathsf{vk}_i\}_{i\in S}, m)$
return $\rho = \rho'$

Fig. 3: Threshold VUF, i.e., $(n, [1, \ldots, 1], t)$-weighted VUF.

$\underline{\mathsf{Setup}(1^\kappa, n, \boldsymbol{w}, w)} \to \mathsf{pp}$
$\mathsf{H} : \{0,1\}^* \to \hat{\mathbb{G}}$
return $(h \in \mathbb{G}, \mathsf{H}, n, \boldsymbol{w}, w)$

$\underline{\mathsf{KeyGen}(\mathsf{pp})} \to (\mathsf{sk}_i, \mathsf{vk}_i)_{i\in[n]}$:
parse $[w_1, \ldots, w_n] := \boldsymbol{w}$
$W := \sum_{i\in[n]} w_i$;
$s_i := \sum_{j=1}^{i-1} w_i, \forall i \in [n]$
$a \leftarrow\$ \mathbb{F}$;
$(a_j)_{j\in[W]} \leftarrow \mathsf{ShamirShare}(a, w, W)$
$\mathsf{sk}_i := r_i \leftarrow\$ \mathbb{F}$;
$\mathsf{vk}_i := (h^{r_i}, h^{r_i a_{s_i+1}}, \ldots h^{r_i a_{s_i+w_i}})$
$\mathsf{sk} := h^a$; $\mathsf{pk} := \{\mathsf{vk}_i\}_{i\in[n]}$
return $\mathsf{pk}, \mathsf{sk}, \{\mathsf{sk}_i, \mathsf{vk}_i\}_{i\in[n]}$

$\underline{\mathsf{Eval}(\mathsf{sk} = h^a, m)} \to \rho$:
return $\rho := e(h^a, \mathsf{H}(m))$

$\underline{\mathsf{ShareEval}(\mathsf{sk}_i, m)} \to \sigma_i$:
// As in the threshold case
return $\mathsf{H}(m)^{1/r_i}$

$\underline{\mathsf{ShareVerify}(\mathsf{vk}_i = (h^{r_i}, \cdots), m, \sigma_i)} \to 0/1$:
// As in threshold case
assert $e(h^{r_i}, \sigma_i) = e(h, \mathsf{H}(m))$

$\underline{\mathsf{Derive}(w, S, \{\sigma_i, \mathsf{vk}_i\}_{i\in S}, m)} \to (\rho, \pi)/\perp$:
assert $\mathsf{ShareVerify}(\sigma_i, \mathsf{vk}_i, m), \quad \forall i \in S$
$(\ell_{i,j})_{i\in S} := \mathsf{WeightedLagCoeffs}(S, (s_i, w_i)_{i\in S}, w)$
$\rho := \prod_{i\in S} e\left(\prod_{j\in[w_i]} \left(h^{r_i \cdot (a_{s_i+j})}\right)^{\ell_{i,j}}, \sigma_i\right)$;
// $= \prod_{i\in S} e\left(\prod_{j\in[w_i]} \left(h^{r_i \cdot (a_{s_i+j})}\right)^{\ell_{i,j}}, \mathsf{H}(m)^{\frac{1}{r_i}}\right)$
// $= e(h^a, \mathsf{H}(m))$
$\pi := \{\sigma_i\}_{i\in S}$
return $\rho, \pi := S, \{\sigma_i\}_{i\in S}$

$\underline{\mathsf{Verify}(\mathsf{pk} = \{\mathsf{vk}_i\}_{i\in[n]}, m, \rho, \pi)} \to 0/1$:
parse $\pi = (S, \{\sigma_i\}_{i\in S})$
$\rho', \cdot := \mathsf{Derive}(S, \{\sigma_i, \mathsf{vk}_i\}_{i\in S}, m)$
return $\rho = \rho'$

Fig. 4: Weighted VUF

Verification. Unlike [12,43], our final VUF output is not directly verifiable. Instead, it is possible to verify the VUF shares that are used to derive the final VUF output by checking that $e(h^{r_i}, \sigma_i) = e(h, \mathsf{H}_{\mathbb{G}}(m))$ holds for each $i \in S$ (see §8 for a different verification methodology).

Analysis. The unweighted threshold VUF is a special case of $(n, \boldsymbol{w}, w)$-weighted VUF with $\boldsymbol{w} = [1, \ldots, 1]$, and $w = t$, and hence its security follows the from security of our weighted VUF that we prove in in §3.4. In terms of performance, per-party computation is one exponentiation, and per-party verification requires two bilinear pairings. To derive the VUF output, an aggregator performs $O(n \log^2 n)$ scalar operations to compute the Lagrange coefficients [60], and $|S| + 1$ exponentiations and bilinear pairings.

3.3 The Weighted VUF Protocol

Let $\boldsymbol{w} = [w_1, \ldots, w_n]$ be the weights of the parties, $W := \sum_{i\in[n]} w_i$ be the total weight, and w be the threshold. We summarize the scheme in Figure 4.

Differences from the unweighted VUF. The two differences between our weighted VUF and the unweighted VUF are (i) the public keys, and (ii) the VUF derivation function. More precisely, the public key size of a party in our weighted VUF is proportional to its weight, i.e., public key vk_i of party i contains w_i+1 group elements. Similarly, the VUF derivation time is proportional to the total weight (but the number of pairings, which are the major computation overhead, is only linear in the number of parties). The secret key of each party, the VUF share of each party and its verification, and the final VUF output are unchanged.

Setup and key generation. The public parameters are identical to the single-party VUF, i.e., a generator $h \in \mathbb{G}$, and a hash function $\mathsf{H} : \{0,1\}^* \to \hat{\mathbb{G}}$. The secret key of party i is $\mathsf{sk}_i := r_i \leftarrow\$ \mathbb{F}$, and its length is independent of its weight. The length of the public key, however, is proportional to the weight. Let $a(x)$ be a random degree w polynomial where $a_j := a(j)$ and $a := a(0)$. Also, let $s_i := \sum_{j=1}^{i-1} w_i$. The public key vk_i of party i consists of $w_i + 1$ elements, and is $\mathsf{vk}_i := (h^{r_i}, h^{r_i a_{s_i+1}}, \ldots, h^{r_i a_{s_i+w_i}})$. The VUF output on input m is $e(h^a, \mathsf{H}(m))$.

VUF evaluation and verification. The VUF evaluation and share verification are identical to the unweighted case and are independent of the party's weight.

VUF derivation. Our VUF derivation relies on the fact that, for any set $S \subseteq [n]$ of parties with $\sum_{i\in T} w_i > w$ and corresponding $\{a_{s_i+1}, \ldots, a_{s_i+w_i}\}_{i\in S}$, there exist Lagrange coefficients $\{\ell_{i,j}\}_{i\in S; j\in[w_i]}$ such that $a = a(0)$ can be expressed as $a = \sum_{i\in S}\sum_{j\in w_i} \ell_{i,j} \cdot a_{s_i+j}$.

Any aggregator, upon receiving valid VUF evalutions from a set $S \subseteq [n]$ of parties with a combined weight greater than w, computes the VUF output as:

$$\begin{aligned}\rho &= \prod_{i\in S} e(\prod_{j\in[w_i]} (h^{r_i a_{s_i+j}})^{\ell_{i,j}}, \sigma_i) = \prod_{i\in S}\prod_{j\in[w_i]} e((h^{r_i a_{s_i+j}})^{\ell_{i,j}}, \mathsf{H}(m)^{1/r_i}) \\ &= \prod_{i\in S}\prod_{j\in[w_i]} e(h^{\ell_{i,j}\cdot a_{s_i+j}}, \mathsf{H}(m)) = e(h^a, \mathsf{H}(m)) \end{aligned} \tag{2}$$

Protocol intuition. An intuitive way to view our VUF scheme is to view h^a as the global VUF key and $[h^{a_{s_i+1}}, \ldots, h^{a_{s_i+w_i}}]$ are the i-th party's VUF keys. This implies that on a input m, the VUF shares of party i are $[e(h^{a_{s_i+j}}, \mathsf{H}(m))]_{j\in w_i}$. However, unlike the virtualization-based approach, parties in our scheme do not store their VUF keys locally. Instead, a trusted key generation (or a DKG) publishes encryptions of all the VUF keys, and each party i receives and stores the randomness r_i used to encrypt its VUF keys. Later, to evaluate the VUF on any input m, each party i uses its encryption randomness r_i to compute $\mathsf{H}(m)^{1/r_i}$ and sends it to the aggregator. Upon receiving $\mathsf{H}(m)^{1/r_i}$ from party i, the aggregator then uses it to compute the VUF output shares of party i on m, and combines the more than w VUF shares to compute the VUF output $e(h^a, \mathsf{H}(m))$.

Readers familiar with functional encryption [15] primitive can notice that our VUF scheme combines functional encryption and threshold cryptography. More specifically, each party first publishes encryptions of its VUF keys (in our case, the trusted key generation publishes these encryptions). Later, on a input m, each party i publishes the functional decryption key $\mathsf{H}(m)^{1/r_i}$, that the aggregator uses to compute $e(h^{a_{s_i+j}}, \mathsf{H}(m))$ for each $j \in [w_i]$.

3.4 VUF Analysis with Trusted Key Generation

In this section, we prove the security of weighted VUF, assuming a trusted key generation algorithm generates its keys. We will only prove the security of weighted VUF, as the single party and unweighted VUF are special cases of the weighted VUF, and hence their security is implied by its security. In §C, we analyze the security of the weighted VUF when instantiated with our DKG.

The *correctness* property can be easily verified through observation. For uniqueness, for each $i \in S$, $\mathsf{ShareVerify}$ checks that $e(h^{r_i}, \sigma_i)$=$e(h, \mathsf{H}(m))$, which implies that $\sigma_i = \mathsf{H}(m)^{1/r_i}$. Moreover, Derive only aggregates valid VUF shares, thus, from equation (2) we get that $\rho = e(h^a, \mathsf{H}(m))$ is the unique output.

We prove *unpredictability* assuming the hardness of the BDH assumption in pairing group $(\mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T)$ in the random oracle model.

Theorem 1 (Unpredictability). *Assuming the hardness of bilinear Diffie-Hellman (BDH), the weighted VUF protocol of Figure 4 is existentially unpredictable as per Definition 5.*

Proof. We will prove this via a sequence of games. Game $\mathbf{G}_0$ is the $\mathsf{UP\text{-}CMA}^{\mathcal{A}}$ game, and game $\mathbf{G}_3$ is the interaction of $\mathcal{A}$ with $\mathcal{A}_{\mathsf{bdh}}$. Here on, for any game $\mathbf{G}_i$, we will use "$\mathbf{G}_i \Rightarrow 1$" as a shorthand for the event that a PPT adversary $\mathcal{A}$ breaks the unpredictability property in game $\mathbf{G}_i$.

GAME $\mathbf{G}_0$: This game is the unpredictability game $\mathsf{UP\text{-}CMA}^{\mathcal{A}}$ for our weighted VUF scheme, where the game follows the honest protocol. Here, the game provides $\mathcal{A}$ access to the random oracle using standard lazy sampling. Let $h := g^{\alpha_h}$ for some $\alpha_h \in \mathbb{F}^*$. Recall that $h \in \mathbb{G}$ is a random generator.

We also make some purely conceptual changes to the game. Assuming the game outputs 1, let (m^*, ρ^*) denote the VUF output predicted by $\mathcal{A}$. Then, we assume that $\mathcal{A}$ always queries $\mathsf{H}(m^*)$ before outputting the predicted value. This change is without loss of generality and does not change the advantage of $\mathcal{A}$. This is because one could always build a wrapper adversary that internally runs $\mathcal{A}$ but makes a query $\mathsf{H}(m^*)$ before terminating. Clearly, we have

$$\mathsf{Adv}^{\mathsf{UP\text{-}CMA}}_{\mathcal{A},\mathsf{VUF}}(\kappa) = \Pr[\mathbf{G}_0 \Rightarrow 1] = \varepsilon_{\mathsf{vuf}}.$$

GAME $\mathbf{G}_1$: Let q_s be the upper bound on the number of VUF queries on distinct inputs. In this game, we introduce a map $\mathsf{bit\text{-}map}[m] \in \{0,1\}$ that maps messages m to bits. For each query $\mathsf{H}(m)$ with a new message m, we sample $\mathsf{bit\text{-}map}[m]$ from a Bernoulli distribution Ber_η with parameter $\eta = 1/(q_s+1)$. That is, $\mathsf{bit\text{-}map}[m]$ is set to 1 with probability $1/(q_s+1)$ and to 0 otherwise.

The game aborts if $\mathcal{A}$ queries VUF shares for two or more messages for which $\mathsf{bit\text{-}map}$ is set to 1. Stating this differently, the game aborts if there exist two messages (m, m') with $\mathsf{bit\text{-}map}[m] = \mathsf{bit\text{-}map}[m'] = 1$, where $m \neq m'$ and $\mathcal{A}$ has queried for VUF shares on both messages m and m'. The game also aborts if $\mathsf{bit\text{-}map}[m^*] = 0$ for the message m^* or if $\mathcal{A}$ queries for VUF shares on a message m^* from parties not in $\mathcal{S}$. Clearly, if no abort occurs, games $\mathbf{G}_0$ and $\mathbf{G}_1$ are the same. Further, the view of $\mathcal{A}$ is independent of the map b. We obtain:

$$\Pr[\mathbf{G}_1 \Rightarrow 1] = \eta(1-\eta)^{q_s} \cdot \Pr[\mathbf{G}_0 \Rightarrow 1]$$

Now, using an analysis similar to [24,14], we get that

$$\Pr[\mathbf{G}_1 \Rightarrow 1] \geq \frac{1}{4q_s} \cdot \Pr[\mathbf{G}_0 \Rightarrow 1].$$

GAME $\mathbf{G}_2$: In this game, we change how we program the random oracle H. More specifically, we first sample $c \leftarrow\$ \mathbb{F}^*$. Next, for the k-th query to H on a new input m_k, we sample $\beta_k \leftarrow\$ \mathbb{F}$, and then depending upon the value of $\mathsf{bit\text{-}map}[m_k]$, we program the random oracle as follows:

$$\mathsf{bit\text{-}map}[m_k] = 0 \Rightarrow \mathsf{H}(m_k) := \hat{g}^{\beta_k}; \quad \mathsf{bit\text{-}map}[m_k] = 1 \Rightarrow \mathsf{H}(m_k) := \hat{g}^{c \cdot \beta_k} \tag{3}$$

For each k with $\mathsf{bit\text{-}map}[m_k] = 1$, since β_k for each k is uniformly random and independent of $c \neq 0$, then $c \cdot \beta_k$ is also uniformly random. This implies that we program H on each input with a uniformly random value. Therefore, we get $\Pr[\mathbf{G}_1 \Rightarrow 1] = \Pr[\mathbf{G}_2 \Rightarrow 1]$.

GAME $\mathbf{G}_3$: Recall from Figure 1 that $\mathcal{C}, \mathcal{S} \subseteq [n]$ are sets of parties with combined weight $W := \sum_{i \in \mathcal{C} \cup \mathcal{S}} w_i \leq w$, and $\mathcal{H} := [n] \setminus (\mathcal{C} \cup \mathcal{S})$. In this game, we change how we sample the secret keys and compute the VUF shares of parties in $\mathcal{H}$, as follows:

1. For each party $i \in \mathcal{C} \cup \mathcal{S}$ with weight w_i, we sample $a_{i,j} \leftarrow\$ \mathbb{F}$ for each $j \in [w_i]$. Let $a(\cdot)$ be the degree w polynomial such that $a(0) = a$ and $a(s_i + j) = a_{i,j}$, where $s_i := \sum_{k \in [i-1]} w_k$. When $W = w$, the polynomial $a(x)$ is uniquely defined. Otherwise, we randomly choose $w - W$ additional values $a(j)$ for values $j > W$ to define the polynomial.
2. For each $i \in \mathcal{C} \cup \mathcal{S}$, we sample $r_i \leftarrow\$ \mathbb{F}$, use $\mathsf{sk}_i := r_i$ and compute the public key vk_i as:

$$\mathsf{vk}_i := \{h^{r_i}, h^{r_i \cdot a_{i,1}}, \ldots, h^{r_i \cdot a_{i,w_i}}\} \tag{4}$$

3. For each party $i \in \mathcal{H}$, we sample $u_i \leftarrow\$ \mathbb{F}$. It holds that $u_i = b \cdot r_i$ for some unknown $r_i \in \mathbb{F}$. Next, we compute $g^{a_{i,j}}$ for each $j \in [w_i]$ using interpolation in the exponent, and compute vk_i as:

$$\mathsf{vk}_i := \{g^{u_i}, (g^{a_{i,1}})^{u_i}, \cdots, (g^{a_{i,w_i}})^{u_i}\} = \{h^{r_i}, h^{r_i a_{i,1}}, \cdots, h^{r_i \cdot a_{i,w_i}}\} \tag{5}$$

here in equation (5) we use the fact that $u_i = b \cdot r_i$ and $h = g^b$.

4. On the k-th partial VUF query (i, m_k) for party $i \in \mathcal{S} \cup \mathcal{H}$:
   (a) If $i \in \mathcal{S}$, follow the honest protocol.
   (b) For $i \in \mathcal{H}$, if $\mathsf{bit\text{-}map}[m_k] = 0$, output $\sigma_i := (\hat{g}^b)^{\beta_k/u_i} = \mathsf{H}(m_k)^{b/u_i} = \mathsf{H}(m_k)^{1/r_i}$; Otherwise, abort.

Clearly, the distribution of the secret keys $\{\mathsf{sk}_i\}_{i \in \mathcal{C} \cup \mathcal{S}}$ and the polynomial $a(\cdot)$ remains unchanged in this game. Next, since u_i for $i \in \mathcal{H}$ are uniformly random and independent of $b \neq 0$, $r_i := u_i/b$ is uniformly random. This implies that the public keys $\{\mathsf{vk}_i\}_{i \in [n]}$ are identically distributed as in game $\mathbf{G}_2$. Finally, for every message m_k and $i \in \mathcal{H}$ we output the unique correct VUF share σ_i satisfying the validity check $e(h^{r_i}, \sigma_i) = e(h^{r_i}, \hat{g}^{b \cdot \beta_k / r_i}) = e(h, \hat{g}^{b \cdot x_k}) = e(h, \mathsf{H}(m))$. Combining all the above, we get that $\Pr[\mathbf{G}_2 \Rightarrow 1] = \Pr[\mathbf{G}_3 \Rightarrow 1]$.

Combining all our claims, we get that:

$$\Pr[\mathbf{G}_3 \Rightarrow 1] \geq \frac{1}{4q_s} \cdot \Pr[\mathbf{G}_0 \Rightarrow 1] \implies \Pr[\mathbf{G}_3 \Rightarrow 1] \geq \frac{\varepsilon_{\mathsf{vuf}}}{4 \cdot q_s} \tag{6}$$

We next argue that whenever $\mathcal{A}$ wins in $\mathbf{G}_3$, we can use $\mathcal{A}$ to build an adversary $\mathcal{A}_{\mathsf{bdh}}$ to break the BDH assumption. We formally describe $\mathcal{A}_{\mathsf{bdh}}$'s interaction with $\mathcal{A}$ in Figure 12, and describe the critical points it next.

1. $\mathcal{A}_{\mathsf{bdh}}$ on input a BDH tuple $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c)$ uses $h := g^b$ and implicitly uses $\mathsf{sk} := h^a$ as the VUF secret key. Note that in game $\mathbf{G}_3$, the game can interact with $\mathcal{A}$ without knowing sk.
2. $\mathcal{A}_{\mathsf{bdh}}$ uses $\hat{g}^c$ to program the random oracle as in equation (3)

Let (m^*, ρ^*) be the output of $\mathcal{A}$ during its interaction with $\mathcal{A}_{\mathsf{bdh}}$. Then, by definition, we have that $\rho^* = \mathsf{Eval}(\mathsf{sk}, m^*) = e(h^a, \mathsf{H}(m^*)) = e(h^a, \hat{g}^{\beta_k \cdot c})$. Now, since $h^a = g^{ab}$, we get that $\rho^* = e(g, \hat{g})^{abc \cdot \beta_k}$. $\mathcal{A}_{\mathsf{bdh}}$ then outputs $(\rho^*)^{1/\beta_k}$ as the BDH solution. Therefore, we get that, whenever $\mathcal{A}$ wins game $\mathbf{G}_3$, $\mathcal{A}_{\mathsf{bdh}}$ can break BDH, hence we have:

$$\varepsilon_{\mathsf{bdh}} \geq \Pr[\mathbf{G}_3 \Rightarrow 1] \implies \varepsilon_{\mathsf{vuf}} \leq 4 \cdot q_s \cdot \varepsilon_{\mathsf{bdh}} \qquad \square$$

Performance analysis. Other than a public key of size $W + n$, the weighted VUF has several advantages over a naïve approach of running the threshold VUF with W virtual parties: (i) The per-party secret key, VUF evaluation cost, and VUF share size are constant, independent of the weight of the party. (ii) The per-party VUF share verification cost is also constant. (iii) The aggregator computes only $O(|S|)$ pairings to verify all shares and derive the final output.

4 Aggregatable PVSS for Group Elements

In §3, we described our weighted VUF, assuming that the keys are generated by a trusted key generation functionality. However, since our goal is to achieve on-chain randomness in a decentralized setting, relying on a centralized trusted party is undesirable. Instead, we aim to use a distributed key generation protocol (DKG). Note that, in our weighted VUF, the public key size is proportional to the total weight. Specifically, the public keys consist of W evaluations (in the exponent and scaled with secret keys) of a w-degree polynomial $a(\cdot)$, where W is the total weight and w is the threshold. Moreover, it is crucial that $h^{a(0)}$ remains hidden from $\mathcal{A}$. Therefore, we want our DKG to generate a polynomial of degree w and output W evaluations in the exponent. Given that both W and w can be very large (up to several thousand in our evaluation), our DKG needs to be scalable. Unfortunately, prior DKG schemes scale only to a few hundred shares [40,29].

A key component of our scalable DKG protocol is an efficient non-interactive aggregatable PVSS scheme. However, existing aggregatable PVSS schemes, such as SCRAPE [18], have prohibitively high verification

costs. Specifically, in SCRAPE, a verifier must perform W bilinear pairings to verify a single transcript, which is impractical for our application of weighted DKG. To address this, we designed a new PVSS scheme that reduces verification to multi-exponentiations of width W and only a constant number of pairings. As discussed in §2, (multi)-pairings are 7-21× more expensive than multi-exponentiations. Our new weighted PVSS, while still virtualization-based, is significantly more efficient than previous schemes.

4.1 PVSS Definitions

A (publicly verifiable) secret sharing (PVSS) scheme lets a secret holder $\mathcal{D}$ (also commonly known as the dealer) share a secret s among a set of n parties with a weighted access structure of threshold w. Intuitively, public verifiability ensures that any external verifier $\mathcal{V}$ can check that $\mathcal{D}$ has correctly shared a secret among n parties. This must be done without learning any information about the shares or the secret. In this paper, we focus on non-interactive weighted PVSS.

Definition 6 (Non-interactive PVSS). *A non-interactive PVSS is a tuple of PPT algorithms* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Share}, \mathsf{Verify}, \mathsf{DecShare}, \mathsf{Recon})$ *as defined below.*

- $\mathsf{PVSS.Setup}(1^\kappa, n, t) \to \mathsf{pp}$ *takes as input the security parameter and outputs the public parameters* pp *of the PVSS scheme. We assume that all algorithms below implicitly take* pp *as input.*
- $\mathsf{PVSS.KeyGen}(i) \to (\mathsf{dk}_i, \mathsf{ek}_i)$*. The key generation protocol takes as input a party index* i*, and outputs decryption key* dk_i *and the corresponding encryption key* ek_i*. Party* i *keeps* dk_i *private and publishes the encryption key* ek_i*. Let* $\mathbf{ek} := [\mathsf{ek}_i]_{i\in[n]}$*.*
- $\mathsf{PVSS.Share}(\mathbf{ek}, s) \to \mathsf{trx}$*. The dealer* $\mathcal{D}$ *uses it to non-interactively generate shares* $s_1, \ldots, s_n$ *for the secret* s*. It encrypts the* i*-th share* s_i *with the encryption key* ek_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.* $\mathcal{D}$ *also computes a commitment* com *to the secret shares, and outputs* $\mathsf{trx} := (\mathsf{com}, \{c_i, \pi_i\}_{i\in[n]})$ *as the PVSS transcript.*
- $\mathsf{PVSS.Verify}(\mathbf{ek}, \mathsf{trx}) \to 0/1$ *: The transcript verification algorithm takes as input a transcript* $\mathsf{trx} = (\mathsf{com}, \{c_i, \pi_i\}_{i\in[n]})$*, a vector of encryption keys* $\mathbf{ek}$*, and outputs* 1 *(accept) iff* com *is a valid commitment to shares of some secret, and* c_i *values are encryptions of these shares. Otherwise, output* 0 *(reject).*
- $\mathsf{PVSS.DecShare}(\mathsf{trx}, i, \mathsf{dk}_i) \to s_i$ *: The share decryption algorithm takes as input a PVSS transcript* trx*, an index* i *and the* i*-th decryption key* dk_i*, outputs the decryption of the* i*-th ciphertext* c_i *in* trx*.*
- $\mathsf{PVSS.Recon}(\mathbf{ek}, \mathsf{trx}, S, \{\mathsf{dk}_i\}_{i\in S}) \to s$*. In this step, each party* $i \in S$ *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* $(\mathbf{ek}, \mathsf{trx})$ *can validate* $\tilde{\pi}_i$*. Lastly, valid decrypted shares from parties with combined weight greater than* w *can be combined to recover the original secret* s *shared by* $\mathcal{D}$*.*

We also require our PVSS scheme to be aggregatable, i.e., to provide a public function $\mathsf{Aggregate}$ *that anyone can use to aggregate two PVSS transcripts into a single transcript.*

- $\mathsf{PVSS.Aggregate}(\mathsf{trx}_1, \mathsf{trx}_2) \to \mathsf{trx}$ *takes as input two PVSS transcripts* trx_1 *and* trx_2 *for secrets* h^{a_1} *and* h^{a_2}*, respectively, and outputs a PVSS transcript* trx *for the aggregated secret* $h^{a_1} \cdot h^{a_2}$*.*

Required properties. A PVSS scheme must ensure *correctness, verifiability*, and *secrecy*. Informally, *correctness* means that if $\mathcal{D}$ 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 parties to use the wrong shares are detected (except with negligible probability). *Secrecy* means that the view of any subset of parties with combined weight less than w can be simulated by a simulator that only sees a commitment to the shared secret. We also require the PVSS scheme to be *aggregatable*, meaning that we want the size of the aggregated transcript to be of the same order as that of a single transcript. We formalize these properties in Appendix B and prove they are sufficient for the security of our weighted VUF.

Secret reconstruction. While we define and describe the secret reconstruction algorithm of the PVSS, in our application of DKG, the shared secret is never reconstructed.

$\underline{\mathsf{PVSS.Setup}(1^\kappa, n, t)} \to \mathsf{pp}$
Let $g, h \in \mathbb{G}$ and $\hat{g} \in \hat{\mathbb{G}}$ be three independent generators
return $\mathsf{pp} := (g, h, \hat{g}, n, t)$.

$\underline{\mathsf{PVSS.KeyGen}(i)} \to (\mathsf{dk}_i, (\mathsf{ek}_i, \mathsf{pok}_i))$
$\mathsf{dk}_i \leftarrow\$ \mathbb{F}; \quad \mathsf{ek}_i := g^{\mathsf{dk}_i}$
return $(\mathsf{dk}_i, (\mathsf{ek}_i, \mathsf{PoK.Dlog}(g, \mathsf{ek}_i, \mathsf{dk}_i)))$ // PoK from Fig. 6

$\underline{\mathsf{PVSS.Share}(\mathbf{ek}, a_0)} \to \mathsf{trx}$
$p(X) := \sum_{i=0}^{t} a_i X^i$, where $(a_1, \ldots, a_t) \leftarrow\$ \mathbb{F}^t$
$V_0, V_1, \ldots, V_n := g^{p(0)}, g^{p(1)}, \ldots, g^{p(n)}$
$\hat{V}_0, \hat{V}_1, \ldots, \hat{V}_n := \hat{g}^{p(0)}, \hat{g}^{p(1)}, \ldots, \hat{g}^{p(n)}$
$\mathsf{pok} \leftarrow \mathsf{PoK.Dlog}(\hat{V}_0, \hat{g}, p(0))$ // PoK from Fig. 6
$\mathbf{pok} := [\mathsf{pok}]$ // Store as a vector of length 1
$r \leftarrow\$ \mathbb{F} \qquad \hat{R} := \hat{g}^r$
$C_0, C_1, \ldots, C_n := g^r, h^{p(1)}\mathsf{ek}_1^r, \ldots, h^{p(n)}\mathsf{ek}_n^r$
return $(\hat{R}, \mathbf{pok}, \boldsymbol{V}, \hat{\boldsymbol{V}}, \boldsymbol{C})$.

$\underline{\mathsf{PVSS.Verify}(\mathbf{ek}, \mathsf{trx})} \to \{0, 1\}$
$(\hat{R}, \mathbf{pok}, \boldsymbol{V}, \hat{\boldsymbol{V}}, \boldsymbol{C}) := \mathsf{trx}$
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** $\mathsf{PoK.DlogVer}(\mathsf{pok}_i)$
assert $V_0 = \prod_{j \in \mathbf{pok}} V_0^{(j)}$ // $V_0^{(j)}$ refers to V_0 of $\mathsf{pok}_j \in \mathbf{pok}$
$\forall i \in [0, n]$, **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

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

$\underline{\mathsf{PVSS.DecShare}(\mathsf{trx}, i, \mathsf{dk}_i)} \to s_i$
$(\cdot, \boldsymbol{C} = [C_0, C_1, \ldots, C_n]) := \mathsf{trx}$
return $s_i := C_i / (C_0)^{\mathsf{dk}_i}$

$\underline{\mathsf{PVSS.Recon}(\mathbf{ek}, \mathsf{trx}, S, \{\mathsf{dk}_i\}_{i \in S})} \to s$
Let $\tilde{s}_i := \mathsf{PVSS.DecShare}(\mathsf{trx}, i, \mathsf{dk}_i)$ for all $i \in S$
assert $\exists Q \subseteq S, |Q| \geq t + 1$ such that: // find subset Q of $\geq t + 1$ valid shares
$\quad e(\tilde{s}_i, \hat{g}) = e(h, \hat{V}_i), \quad \forall i \in Q$
// Compute Lagrange coeff. $\ell_{Q,i}(0) = \prod_{j \in Q \ j \neq i} j/(j - i)$
return $s \leftarrow \prod_{i \in Q} \tilde{s}_i^{\ell_{Q,i}(0)}$ // $= h^{p(0)}$

Fig. 5: Threshold PVSS and transcript aggregation.

4.2 Unweighted PVSS Design

Since our new non-interactive weighted PVSS is virtualization-based, without loss of generality, we will describe our scheme with n parties, assuming parties are unweighted. Our scheme lets a dealer $\mathcal{D}$ share a random secret $s \in \hat{\mathbb{G}}$. To verify the PVSS transcript, a verifier $\mathcal{V}$ needs to perform only four pairings and four 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, which is the multiplication of the secrets shared by all dealers. The aggregated transcript is also publicly verifiable.

$\underline{\mathsf{PoK.Dlog}(g, y, \alpha) \to \mathsf{pok}}$
$r \leftarrow\$ \mathbb{F}; \quad u := g^r$
// H modeled as a random oracle
$c := \mathsf{H}(g, y, u) \in \mathbb{F}; \quad z := r + c \cdot \alpha$
return (g, y, u, z)

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

Fig. 6: Schnorr protocol for Proof-of-knowledge (PoK) of $\alpha = \log_g y$

The only added verification cost is verifying one additional proof-of-knowledge per PVSS transcript that was aggregated.

Our PVSS protocol combines ideas from the SCRAPE PVSS protocol [18] for group element secrets and Groth's PVSS for field element secrets. We summarize the PVSS scheme in Figure 5, and describe its details next.

Setup The public parameters of our scheme are $(n, t, g, h, \hat{g})$. Here, g, h are two uniformly random and independent generators of $\mathbb{G}$, and $\hat{g}$ is generator of $\hat{\mathbb{G}}$.

Key Generation. The decryption key of party i is $\mathsf{dk}_i \leftarrow\$ \mathbb{F}$ and $\mathsf{ek}_i := g^{\mathsf{dk}_i}$ is the encryption key. We also require each party i to also publish, along with its encryption key ek_i, a proof-of-knowledge (PoK) pok_i of its decryption key dk_i. Concretely, we use the non-interactive variant of the Schnorr identification scheme as our PoK [53] (see Figure 6). Each party must provide this proof once, when it joins the system (and also whenever it changes its public key).

Intuitively, the attached PoK prevents the adversary from launching rogue-key attacks [50,13]. If PoK's are not present, parties could set their public keys as a function of the public keys of other parties. As a result, even a single corrupt party could set its key to learn the shared secret.

Transcript generation ($\mathsf{PVSS.Share}$). Let $p(x)$ be a random degree t polynomial. The share of party i is $h^{p(i)}$. $\mathcal{D}$ first computes two commitment vectors:

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

$\mathcal{D}$ also 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 parties from sharing secrets that depend on the secrets shared by other parties (e.g., canceling previous secrets).

$\mathcal{D}$ 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.

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

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. A verifier $\mathcal{V}$ first validates the PoKs included in **pok**. Then, $\mathcal{V}$ checks that $\boldsymbol{V}$ commits to evaluations of a polynomial of degree $\leq t$ using the low-degree test from [18] (see Appendix A).

Next, $\mathcal{V}$ checks that the encryptions are valid, i.e., that C_i encrypts the share of party 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 three 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 equal. Namely $e(C_0, \hat{g}) = e(g, \hat{R})$.
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 $i \in [0, n]$, $e(g, \hat{V}_i) = e(V_i, \hat{g})$.

Optimizing transcript verification. The PVSS transcript verification procedure we described so far requires $5n + 2$ pairings. This cost can be reduced to four pairings and four n-wide multi-exponentiations by verifying a random linear combination of all n ciphertexts and commitments. 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, $\mathcal{V}$ here checks that a random linear combination of the ciphertexts is valid with respect to the same random linear combination of the commitments and the encryption keys and that $e(C_0, \hat{g}) = e(g, \hat{R})$. 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 always fails, except with a negligible probability.

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

On using the commitments. The commitments $V_0, \ldots, V_n$ are not directly used in the PVSS protocol. They are included since we use them in the proof of security of the VUF protocol that uses the PVSS for generating its keys.

4.3 Weighted Aggregatable PVSS using Virtualization

Our weighted PVSS scheme is virtualization-based, and treats party i with weight w_i as w_i virtual parties. $\mathcal{D}$ shares its secret using the unweighted PVSS scheme with (W, w)-threshold secret sharing. Each party receives a number of shares that is equal to its weight. More precisely, party i with weight w_i has w_i independent encryption keys, i.e.,

$$\mathsf{dk}_i := \{\mathsf{dk}_{i,j}\}_{j \in [w_i]};\ \mathsf{ek}_i := \{\mathsf{ek}_{i,j}\}_{j \in [w_i]}$$

where for each $j \in [w_i]$, $\mathsf{dk}_{i,j} \leftarrow\$ \mathbb{F}$ and $\mathsf{ek}_{i,j} := g^{\mathsf{dk}_{i,j}}$. Party i then receives one share per encryption key, which is encrypted using that encryption key.

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

Regarding performance, the PVSS sharing and verification costs are proportional to the total weight W. Specifically, to verify a transcript, $\mathcal{V}$ performs four W-wide exponentiations and four bilinear pairings, which are orders of magnitude more efficient than performing W pairings. Additionally, each party i needs to decrypt w_i ciphertexts.

5 Distributed Key Generation

In this section, we describe a distributed key generation (DKG) protocol to generate secret keys of our weighted VUF in a decentralized manner. Recall from §3 that the secret keys in our VUF are independent random values, whereas the public keys are W evaluations (in the exponent and scaled with secret keys) of a degree w polynomials. More precisely, the public keys are $[\mathsf{pk}_i]_{i \in [n]}$, for

$$\mathsf{pk}_i = [h^{r_i}, h^{a_{s_i+1} \cdot r_i}, \ldots, h^{a_{s_i+w_i} \cdot r_i}]$$

where $s_i = \sum_{j \in [i-1]} w_j$ and r_i is the secret key of party i.

Note that the structure of our VUF keys is different from the typical key structure in threshold cryptosystems, where the secret keys are threshold shares (typically using the Shamir-secret sharing scheme) of a random secret. Thus, the DKG protocols used in those schemes are not applicable to us, and hence, we present a new DKG protocol. We will describe our DKG in two steps. First, we present a new scalable

general-purpose DKG for group element secrets, as existing DKG schemes do not scale to our requirements. Second, we describe how to augment our standard DKG (or any standard DKG) using only one round of communication to generate keys for our VUF. We adopt this two-step approach because of its modularity and the applicability of our new scalable standard DKG in other applications [43].

We organize the rest of the section as follows. We begin by defining the standard DKG and then present our new DKG construction that meets this definition. We then describe the augmentation step needed to generate keys for our weighted VUF.

5.1 DKG Definition

Definition 7 (Weighted DKG). *A weighted distributed key generation (DKG) protocol amounts to secret sharing a random secret* $\mathsf{dsk} := h^a \leftarrow\$ \mathbb{G}$ *and making public the public key* $\mathsf{dpk} := \hat{g}^a$ *and threshold public keys* $[\mathsf{dpk}_i]_{i\in[n]}$*. Let* n *be the number of parties with weights* $[w_1, \ldots, w_n]$*, and let* $W := \sum_{i\in[n]} w_i$ *be the total weight. Let* $p(\cdot) \in \mathbb{F}[x]$ *be a polynomial of degree* w *such that* $z := h^{p(0)}$ *is the shared secret. At the end of the protocol party* i *outputs* $\mathsf{dsk}_i := h^{p(s_i+1)}, \ldots, h^{p(s_i+w_i)}$*, its* w_i *shares of the secret key of the secret* dsk*, where* $s_i := \sum_{j\in[i-1]} w_j$*. We require the DKG protocol to satisfy the following* correctness *properties in the presence of an adversary* $\mathcal{A}$ *that corrupts parties with a combined weight of up to* w*.*

(C1) All subsets of $w+1$ *shares provided by honest parties define the same unique secret key* $\mathsf{dsk} = h^a$*.*

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

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

Note that we do not require the 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 [37,40,56,29]. Instead, we directly prove that our weighted VUF scheme is $\mathsf{UP\text{-}CMA}^{\mathcal{A}}$ secure when the game runs our DKG protocol with $\mathcal{A}$ to generate the secret keys. A similar approach was used in [40,23,27] to prove the combination of Pedersen's DKG scheme [49,43] with many existing threshold cryptosystems. We present the combined proof in Appendix C.

5.2 Weighted DKG Design

The weighted DKG uses the non-interactive PVSS scheme described in in §4. The natural approach for constructing a DKG given a non-interactive PVSS scheme is as follows: First, each party, as a dealer, publishes a PVSS transcript for a random secret using a total order broadcast channel (see Definition 8). Then, each party locally validates all 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 shares from valid PVSS transcripts and aggregating them locally.

This approach has appeared in many prior works [37,52,42,44,20]. Apart from its simplicity, it has additional advantages: It uses the total order broadcast channel in a black-box manner and can use more efficient non-aggregatable PVSS schemes, such as [18,21]. However, this approach has some disadvantages: *first*, it always requires all PVSS transcripts to be sent over the broadcast channel, which can be prohibitively expensive; *second*, it has higher latency, as it might not be possible to broadcast all PVSS transcripts simultaneously. For example, if a blockchain is used as the broadcast channel, it might not be possible to fit all the PVSS transcripts in a single block.

We adopt a different approach that leverages the aggregation property of our PVSS scheme. Our DKG protocol addresses the above-mentioned concerns in the common-case operation (i.e., with no or a few active corruptions) and is thus more appropriate for our use-case of on-chain randomness for PoS blockchains. The main advantage of this DKG protocol 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.

PUBLIC PARAMETERS & INPUTS:

Weights $\boldsymbol{w} = [w_1, \ldots, w_n]$. Let $W := \sum_{i \in [n]} w_i$ and w be the threshold.
PVSS public parameters pp // $= \mathsf{PVSS.Setup}(1^\kappa, W, w)$
$\mathsf{dk}_i, \mathbf{ek} := [\mathsf{ek}_j]_{j \in [W]}$ // where for all $j \in [W]$, let $(\mathsf{dk}_i, \mathsf{ek}_j) := \mathsf{PVSS.KeyGen}(j)$
Signature verification keys of all parties and signing key of party i

---

SHARING PHASE:

Let $s_i \leftarrow\$ \mathbb{F}$
Let $\mathsf{trx}_i := (\mathsf{pok}_i, \cdot) \leftarrow \mathsf{PVSS.Share}(\mathbf{ek}, s_i)$
Let $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{pok}_i)$
send $(\sigma_i, \mathsf{trx}_i)$ to all

AGREEMENT PHASE:

Let $w_+ := w_i$, $\mathsf{trx} := \mathsf{trx}_i$, and $\boldsymbol{\sigma} = \{\sigma_i\}$
upon receiving $(\sigma_j, \mathsf{trx}_j)$ from party j : // only once
    if $\mathsf{PVSS.Verify}(\mathbf{ek}, \mathsf{trx}_j) = 1$ and σ_j is valid :
        $\mathsf{trx} := \mathsf{PVSS.Aggregate}(\mathsf{trx}, \mathsf{trx}_j)$
        $\boldsymbol{\sigma} := \boldsymbol{\sigma} \cup \{\sigma_j\}$ and $w_+ := w_+ + w_j$
        if $w_+ \geq w$: **break**
// Repeat until the first honest party outputs
if chosen as a broadcaster :
    broadcast $(\boldsymbol{\sigma}, \mathsf{trx})$ using a total order broadcast
upon receiving $(\boldsymbol{\sigma}, \mathsf{trx})$ from the broadcast channel :
    Let T be the indices of parties with signatures in $\boldsymbol{\sigma}$
    assert $\sigma_k \in \boldsymbol{\sigma}$ for each $k \in T$ are valid
    assert trx is valid for the set T
    assert $\sum_{i \in T} w_i \geq w$
    if all checks are successful :
        output trx and go to the key-derivation phase

KEY DERIVATION PHASE:

Let $\mathsf{trx} = (\cdot, \hat{\boldsymbol{V}}, \cdot)$ be the output of the agreement phase.
Let $\mathsf{dsk}_i := \mathsf{PVSS.DecryptShare}(\mathsf{trx}, i, \mathsf{dk}_i)$
Let $\mathsf{dpk} := \hat{\boldsymbol{V}}[0]$ and $\mathsf{dpk}_j := \hat{\boldsymbol{V}}[s_{j-1} : s_j]$, $\forall j \in [n]$
return $\mathsf{dsk}_i, \mathsf{dpk}, \{\mathsf{dpk}_j\}_{j \in [n]}$

Fig. 7: Weighted DKG protocol for party i

The public parameters for the DKG scheme consist of the public parameters of the PVSS scheme, a 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. The DKG scheme works in three phases: *Sharing, Agreement*, and *Key derivation*. We summarize our protocol in Figure 7, and describe it next.

Sharing phase. During the sharing phase, each party i computes the PVSS transcript $\mathsf{trx}_i := (\cdot, [\mathsf{pok}_i], \cdot) \leftarrow \mathsf{PVSS.Share}(\mathbf{ek}, s_i)$ for a uniformly random secret $s_i \leftarrow\$ \mathbb{F}$. 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 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.

$\underline{\mathsf{AugmentKeyGen}(\mathsf{dsk}_i = [\mathsf{dsk}_{i,j}]_{j\in[w_i]})} \to (\mathsf{sk}_i, \mathsf{pk}_i)$
$\mathsf{sk}_i := r_i \leftarrow\$ \mathbb{F}$
$\mathsf{pk}_i := (h^{r_i}, [\mathsf{dsk}_{i,j}^{r_i}]_{j\in[w_i]})$ // $= (h^{r_i}, [h^{r_i \cdot a_{s_i+j}}]_{j\in[w_i]})$
return $(\mathsf{sk}_i, \mathsf{pk}_i)$
$\underline{\mathsf{KeyVerify}(\mathsf{pk}_i, \mathsf{dpk}_i)} \to \{0, 1\}$
Parse $(h^{r_i}, [\mathsf{pk}_{i,j}]_{j\in[w_i]}) = \mathsf{pk}_i$ and $[\mathsf{dpk}_{i,j}]_{j\in[w_i]} = \mathsf{dpk}_i$
Sample $\gamma_1, \ldots, \gamma_{w_i} \leftarrow\$ \mathbb{F}$
assert $e(h^{r_i}, \prod_{j\in[w_i]} \mathsf{dpk}_{i,j}^{\gamma_j}) = e(\prod_{j\in[w_i]} \mathsf{pk}_{i,j}^{\gamma_j}, \hat{g})$
// checking whether $e(h^{r_i}, \hat{g}^{a_{s_i+j}}) = e(h^{r_i \cdot a_{s_i+j}}, \hat{g}), \forall j \in [w_i]$
// by checking a random linear combination of the terms

Fig. 8: DKG augmentation step

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 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 **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 a valid aggregated transcript is output by the total order broadcast.

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 because 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 agreed-upon aggregated transcript from the agreement phase. During the key-derivation phase, each party i locally derives its DKG secret key dsk_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 DKG public key dpk and the threshold public keys $\{\mathsf{dpk}_j\}_{i\in[n]}$ from the $\hat{\boldsymbol{V}}$ vector included in the aggregated PVSS transcript.

5.3 DKG Augmentation for Weighted VUF

Next, we describe how we augment our DKG protocol to generate keys for our weighted VUF. We summarize these steps in Figure 8 and describe them next.

Protocol. Each party i uses $\mathsf{AugmentKeyGen}$ to generate its VUF keys, where the secret key sk_i is a random field element r_i, i.e., $\mathsf{sk}_i := r_i \leftarrow\$ \mathbb{F}$. Party i computes its VUF public key pk_i using its DKG secret key dsk_i as in Figure 8, and sends pk_i to all, using a reliable broadcast (see Definition 9). Anyone, upon receiving pk_j from party j, uses the $\mathsf{KeyVerify}$ algorithm and the DKG public key dpk_i to check its validity. Intuitively, the $\mathsf{KeyVerify}$ algorithm ensures that each party i uses a single random value r_i as the exponent for all of its VUF public keys.

Analysis. The correctness of the DKG protocol we describe in §5.2 follows from the correctness of the PVSS scheme and the security guarantees of the total order broadcast protocol (see Definition 8). It is easy to see that a correct DKG execution and a successful $\mathsf{KeyVerify}$ check ensure that the public keys pk_i have the

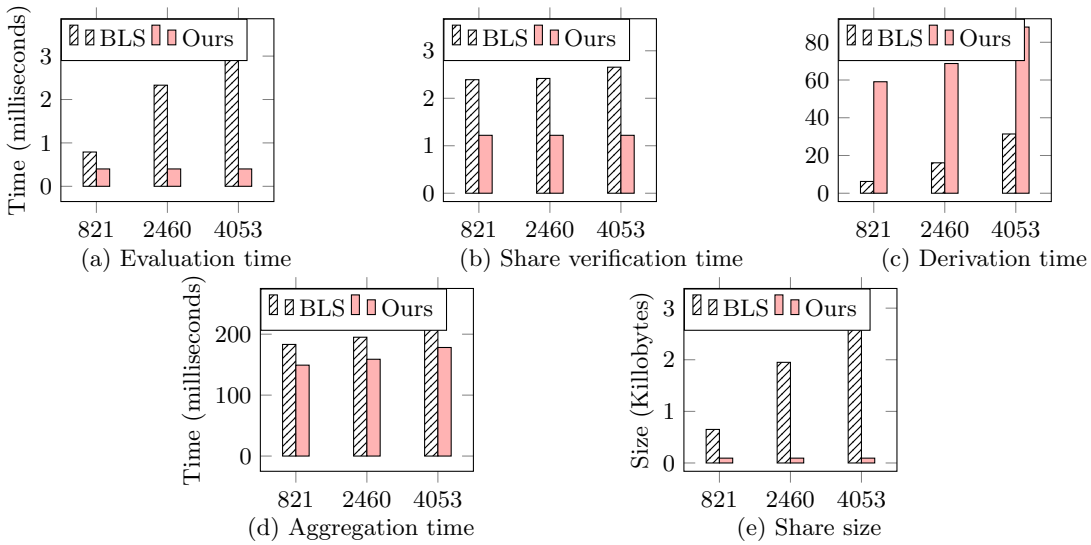

Fig. 10: Micro-benchmarks of BLS virtualization and our scheme. Here, x-axis denotes the total weight of the system.

correct structure needed for weighted VUF. In Appendix C, we prove the security of our weighted VUF scheme when our DKG protocol (with augmentation) is used to generate the VUF keys.

6 Implementation and Evaluation

We implement our on-chain randomness protocol in Rust atop the open source implementation of Aptos blockchain [3], a proof-of-stake blockchain (`https://github.com/aptos-labs/aptos-core`). Our implementation includes all parts of our system, i.e., a weighted DKG, a weighted VUF, and the steps to derive shared randomness using a weighted VUF. We will make our implementation publicly available. For cryptography, we use the `blstrs` library [58], which implements efficient finite field and elliptic curve arithmetic. Throughout our implementation, we use (for both our implementation and the baselines) Pippenger's method [10] for multi-exponentiation of group elements.

6.1 Micro-benchmarks

Metrics. For our microbenchmark, we measure the *evaluation time*, *share verification time*, *derivation time*, *aggregation time*, and *share size*. The evaluation time refers to the time a party takes to compute its VUF share on a message. The share verification time measures the time the aggregator takes to verify a VUF share. The derivation time measures the time an aggregator takes to compute the VUF output given a set of valid VUF shares. The aggregation time measures the time an aggregator takes to verify a subset of VUF shares necessary to compute the VUF output and the time to derive the VUF output. More precisely, if VUF shares from t parties are needed to derive the VUF output, then the aggregation time is the sum of t share verification times and the VUF derivation time. The share size is the size of the VUF share of a party.

For these metrics, we compare our VUF scheme with the threshold VUF based on Boldyreva's BLS threshold signatures [12] with virtualization to support weights. From here on, we refer to this baseline as the BLS virtualization. While measuring the share verification time of the BLS virtualization, we implement the optimization where the verifier verifies all w_i shares of party i in a batch using only two pairings and two w_i-wide multi-exponentiations.

Results. We microbenchmark the computation costs using a t2d-standard-32 Google Cloud virtual machine with 32 vCPUs and 128 GB of memory. We evaluate both the baseline and our scheme using the BLS12-381 elliptic curve. Also, throughout our implementation, we use 256-bit scalars for the random linear combination-based checks. We report our results in Table 10. We seek to illustrate that our scheme improves over the BLS virtualization.

Table 1: End-to-end latency with 112 validators

Total weights	DKG latency (sec)		On-chain randomness latency (ms)
	Sharing	Agreement	
821	1.4	18.6	110
2460	2.3	40.7	121
4053	3.0	61.0	133

Evaluation time. As expected, the average per-party evaluation 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.

Share verification time. As expected, the share verification times of our scheme are constant (1.22ms), independent of the total weight, and are about 2× faster than those of BLS virtualization.

Derivation time. The derivation time of both BLS virtualization and our scheme is proportional to the total weight. In our scheme, the aggregator needs to compute $O(t)$ additional pairings compared to the BLS virtualization, where t 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 virtualization increases. This also explains the slower growth of our VUF's aggregation time. 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. For all three weight distributions we consider, we need to combine VUF shares from 74 parties on average. Thus, for both BLS virtualization and our approach, the aggregation time is the sum total of 74 share verification time and the derivation time. As expected, although the 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. Moreover, as we expect the gap between the derivation time of our approach and the BLS virtualization to reduce with higher total weight, the aggregation will further improve compared to BLS virtualization.

Share size. In the BLS virtualization, the average share size grows linearly with the increasing total weights (from 668 to 3297 bytes). On the other hand, in our scheme, the share size is constant (96 bytes). Even for the smallest total weight that we consider, 821, our scheme reduces the share size by a factor of 7×. The reduction is 34× for a total weight of 4053.

To summarize, even though our derivation time is slower than that of BLS virtualization, the overall aggregation time of our VUF is better. Furthermore, our communication costs are significantly better than that of BLS virtualization. This is particularly important since computation can be easily parallelized, whereas communication cannot.

6.2 End-to-End Evaluation

We now discuss the end-to-end evaluation of our on-chain randomness atop the Aptos blockchain. The Aptos blockchain proceeds in incremental epochs, where each epoch lasts about two hours, and the validators and their stake distribution can change only during epoch changes.

Implementation. The blockchain consensus can be modeled as a total order broadcast (Definition 8), which outputs the same sequence of committed blocks at all validators. Before every epoch change, validators of the current epoch run the weighted DKG (Figure 7) to generate VUF keys for the next epoch. At the beginning of the new epoch, validators of the new epoch decrypt their keys from the DKG transcript and run the DKG augmentation step (Figure 8) to generate the VUF keys for the new epoch.

For any block B, we use the epoch number and the block height as the VUF input. To generate the VUF output for block B, validators wait until block B is committed by the total order broadcast and then

exchange their VUF shares for block B. Each validator, upon receiving enough valid shares, locally generates the VUF output and hashes it to derive the shared randomness for the block B.

For efficiency, we run the share verification step of different parties in parallel and parallelize the VUF derivation using multi-threading.

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. The average simulated inter-region and intra-region round-trip time is 168 and 100 ms, respectively. For the weight distribution, we used the stake distribution of the 112 validators of the Aptos blockchain from Oct 18, 2023. We used three different total weights (after appropriate rounding): 821, 2460, and 4053. We used 67% of the total weight as the reconstruction threshold for randomness beacon, to illustrate the performance of our protocol even under high reconstruction threshold.

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

Results. We summarize the evaluation results in Table 1. Note that the DKG latency depends almost linearly on the total weight. This is expected, 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 increasing total weight. This is due to the communication costs being unaffected by the total weight. The computation increases with the total weight but is easily parallelizable and not a bottleneck.

7 Related Work

Weighted VUF. To the best of our knowledge, our work is the first weighted VUF construction with concrete efficiency. As discussed in §1, prior to our work, the only known approach to concretely efficient weighted VUF was through virtualization, where a party with weight w emulates w virtual parties. As we illustrate in §6, our weighted VUF is more efficient than the virtualization-based approach. We want to note that the verifiable random function (VRF) used in the proof-of-stake blockchain Algorand is not a threshold VRF, nor is it weighted. Instead, Algorand employs a single-party VRF, which is sufficient for their application of sampling a committee for consensus.

Weighted threshold cryptography. Beimel [5,7] presented the first characterization of a weighted secret sharing (WSS) scheme where the share size is sublinear in the weight of the party. Following works on WSS has explored other approaches such as Chinese remainder theorem [62,38], allowed only restricted classes of hierarchical weights [59,33], or wiretap channels [9]. All these works are theoretical and have very high concrete costs. We emphasize that using a weighted secret sharing for threshold cryptography typically requires a linear secret sharing, and this property is not guaranteed by all weighted constructions.

Very recently, [26,39] designed concretely efficient threshold signature schemes with weighted parties. However, their signatures are not unique and depend on the subset of signers used to aggregate the threshold signature. Thus, these schemes cannot be used as a weighted VUF.

Non-interactive PVSS schemes. Starting with the work of Stadler [57], numerous works have studied non-interactive PVSS schemes [54,37,18,19,42,21,44,20]. Many of these schemes focus on secret sharing a group element secret for better efficiency [18,19,43,21], with the most efficient being [21]. However, PVSS transcripts of [21] are not aggregatable, and hence is not suitable for our DKG. The most efficient aggregatable PVSS is SCRAPE [18], but its transcript verification is costly, requiring W pairings. In contrast, our scheme requires only four pairings and four W-wide multi-exponentiations, making it significantly faster. In terms of techniques, our PVSS scheme combines techniques from SCRAPE and the PVSS scheme due to Groth [42], where the latter focuses on sharing field element secrets and requires expensive range proofs. We adapt Groth's scheme to share group element secrets, thereby eliminate the need for these range proofs.

Distributed key generation. Numerous works have studied interactive DKG protocols under various network assumption, and there are relatively fewer non-interactive DKG constructions, such as [37,42,44,20,46,45].

As we discuss in §5, all these 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 parties.

We adopt a different approach where parties first locally aggregate a subset of PVSS transcripts and then publish the aggregated transcripts in a round-robin manner until the first valid transcript appears on the broadcast channel. We adopt this approach to improve the efficiency of our DKG in the common-case operation, i.e., with no or a few active corruptions. For example, in our approach, 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.

8 Discussion and Conclusion

We presented an efficient on-chain randomness protocol for BFT-based Proof-of-Stake blockchains with weighted validators. A key component of our protocol is a weighted verifiable unpredictable function (VUF) with constant computation and communication costs for each party. We also designed a scalable publicly verifiable secret sharing scheme with an aggregatable transcript, which we used to develop a distributed key generation protocol for our weighted VUF. We implemented our schemes on the Aptos blockchain and evaluated them with 112 geo-distributed validators. Our evaluation shows that our on-chain randomness protocol added only 133 milliseconds of latency and demonstrated performance improvements over baseline methods.

VUF output verification using multi-signatures. One limitation of our weighted VUF is the inefficiency of output verification, which requires re-deriving the output and can be costly. However, this is not a major issue for our on-chain randomness application. We can reduce verification costs for blockchain clients by having validators sign a weighted multi-signature on the final output. Blockchains like Aptos and Sui already use multi-signatures to sign the blockchain state after each block, so the VUF output can be included in the blockchain state with minimal overhead. Clients already check the multisignature, which will also verify the VUF output. For example, verifying a BLS multi-signature with 128 signers takes about 972 microseconds, compared to 900 microseconds for a single BLS threshold signature on an Apple M1 machine with 10-core CPU and 32GB of memory.

On lower-bounds. It is important to note that while there are lower bounds and impossibility results for weighted secret sharing (see, e.g., [6]), they do not apply to computing a VUF. In fact, the setup cost of the VUF for each validator depends on its weight. However, since this setup is only run once per epoch, this is not a major bottleneck.

9 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

1. zkalc is a cryptographic calculator! (2023), `https://github.com/mmaker/zkalc`
2. Chainlink VRF documentation (2024), `https://chain.link/vrf`
3. Aptos: The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure (2022), accessed: 2023-02-19
4. Bagherzandi, A., Cheon, J.H., Jarecki, S.: Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In: Proceedings of the 15th ACM conference on Computer and communications security. pp. 449–458 (2008)
5. Beimel, A., Tassa, T., Weinreb, E.: Characterizing ideal weighted threshold secret sharing. In: Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings 2. pp. 600–619. Springer (2005)

6. Beimel, A., Tassa, T., Weinreb, E.: Characterizing ideal weighted threshold secret sharing. In: Theory of Cryptography, (2005)
7. Beimel, A., Weinreb, E.: Monotone circuits for monotone weighted threshold functions. Information Processing Letters **97**(1), 12–18 (2006)
8. Bellare, M., Crites, E., Komlo, C., Maller, M., Tessaro, S., Zhu, C.: Better than advertised security for non-interactive threshold signatures. In: Annual International Cryptology Conference. pp. 517–550. Springer (2022)
9. Benhamouda, F., Halevi, S., Stambler, L.: Weighted secret sharing from wiretap channels. In: 4th Conference on Information-Theoretic Cryptography (ITC 2023). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2023)
10. Bernstein, D.J.: Pippenger's exponentiation algorithm (2002), `https://cr.yp.to/papers/pippenger.pdf`
11. Blackshear, S., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Li, X., Logan, M., Menon, A., Nowacki, T., Sonnino, A., et al.: Sui lutris: A blockchain combining broadcast and consensus. arXiv preprint arXiv:2310.18042 (2023)
12. Boldyreva, A.: Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In: International Workshop on Public Key Cryptography (2003)
13. Boneh, D., Drijvers, M., Neven, G.: Compact multi-signatures for smaller blockchains. In: International Conference on the Theory and Application of Cryptology and Information Security (2018)
14. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: Advances in Cryptology—ASIACRYPT 2001: 7th International Conference on the Theory and Application of Cryptology and Information Security Gold Coast, Australia, December 9–13, 2001 Proceedings 7. pp. 514–532. Springer (2001)
15. Boneh, D., Sahai, A., Waters, B.: Functional encryption: Definitions and challenges. In: Theory of Cryptography: 8th Theory of Cryptography Conference, TCC (2011)
16. Boneh, D., Shoup, V.: A Graduate Course in Applied Cryptography, v. 06 (January 2023)
17. Buterin, V., Griffith, V.: Casper the friendly finality gadget. arXiv preprint arXiv:1710.09437 (2017)
18. Cascudo, I., David, B.: Scrape: Scalable randomness attested by public entities. In: International Conference on Applied Cryptography and Network Security. pp. 537–556. Springer (2017)
19. Cascudo, I., David, B.: Albatross: publicly attestable batched randomness based on secret sharing. In: International Conference on the Theory and Application of Cryptology and Information Security (2020)
20. Cascudo, I., David, B.: Publicly verifiable secret sharing over class groups and applications to dkg and yoso. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques (2024)
21. Cascudo, I., David, B., Garms, L., Konring, A.: Yolo yoso: fast and simple encryption and secret sharing in the yoso model. In: International Conference on the Theory and Application of Cryptology and Information Security (2022)
22. Chen, Y.H., Lindell, Y.: Optimizing and implementing fischlin's transform for uc-secure zero-knowledge. Cryptology ePrint Archive (2024)
23. Chu, H., Gerhart, P., Ruffing, T., Schröder, D.: Practical Schnorr threshold signatures without the algebraic group model. In: Annual International Cryptology Conference. pp. 743–773. Springer (2023)
24. Coron, J.S.: On the exact security of full domain hash. In: Annual International Cryptology Conference. pp. 229–235. Springer (2000)
25. Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., Juels, A.: Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In: 2020 IEEE Symposium on Security and Privacy (SP) (2020)
26. Das, S., Camacho, P., Xiang, Z., Nieto, J., Bunz, B., Ren, L.: Threshold signatures from inner product argument: Succinct, weighted, and multi-threshold. In: ACM CCS (2023)
27. Das, S., Ren, L.: Adaptively secure bls threshold signatures from ddh and co-cdh. In: Annual International Cryptology Conference. Springer (2024)
28. Das, S., Xiang, Z., Tomescu, A., Spiegelman, A., Pinkas, B., Ren, L.: Verifiable secret sharing simplified. In: (To appear) IEEE Security and Privacy (SP) (2025)
29. Das, S., Yurek, T., Xiang, Z., Miller, A., Kokoris-Kogias, L., Ren, L.: Practical asynchronous distributed key generation. In: IEEE Security and Privacy (SP) (2022)
30. Dodis, Y.: Efficient construction of (distributed) verifiable random functions. In: Public Key Cryptography—PKC 2003: International Workshop on Practice and Theory in Public Key Cryptography (2002)
31. Drand: Drand-a distributed randomness beacon daemon. `https://drand.love/` (2023), accessed: 2023-02-19
32. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM (JACM) (1988)
33. Farras, O., Padró, C.: Ideal hierarchical secret sharing schemes. IEEE transactions on information theory (2012)
34. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques (1986)

35. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2), 374–382 (1985)
36. Fischlin, M.: Communication-efficient non-interactive proofs of knowledge with online extractors. In: Annual International Cryptology Conference. pp. 152–168. Springer (2005)
37. Fouque, P.A., Stern, J.: One round threshold discrete-log key generation without private channels. In: International Workshop on Public Key Cryptography (2001)
38. Garg, S., Jain, A., Mukherjee, P., Sinha, R., Wang, M., Zhang, Y.: Cryptography with weights: Mpc, encryption and signatures. In: Annual International Cryptology Conference (2023)
39. Garg, S., Jain, A., Mukherjee, P., Sinha, R., Wang, M., Zhang, Y.: hints: Threshold signatures with silent setup. In: 2024 IEEE Symposium on Security and Privacy (SP) (2024)
40. Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Secure distributed key generation for discrete-log based cryptosystems. Journal of Cryptology (2007)
41. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: 26th symposium on operating systems principles (2017)
42. Groth, J.: Non-interactive distributed key generation and key resharing. IACR Cryptol. ePrint Arch. **2021**, 339 (2021)
43. Gurkan, K., Jovanovic, P., Maller, M., Meiklejohn, S., Stern, G., Tomescu, A.: Aggregatable distributed key generation. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques (2021)
44. Kate, A., Mangipudi, E.V., Mukherjee, P., Saleem, H., Thyagarajan, S.A.K.: Non-interactive vss using class groups and application to dkg. In: Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (2024)
45. Katz, J.: Round-optimal, fully secure distributed key generation. In: Annual International Cryptology Conference. pp. 285–316. Springer (2024)
46. Komlo, C., Goldberg, I., Stebila, D.: A formal treatment of distributed key generation, and new constructions. Cryptology ePrint Archive (2023)
47. Micali, S., Rabin, M., Vadhan, S.: Verifiable random functions. In: 40th annual symposium on foundations of computer science (1999)
48. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Decentralized business review (2008)
49. Pedersen, T.P.: A threshold cryptosystem without a trusted party. In: Workshop on the Theory and Application of of Cryptographic Techniques (1991)
50. Ristenpart, T., Yilek, S.: The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In: EUROCRYPT: Annual International Conference on the Theory and Applications of Cryptographic Techniques (2007)
51. Rocket, T.: Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. Available [online].[Accessed: 4-12-2018] (2018)
52. Schindler, P., Judmayer, A., Stifter, N., Weippl, E.: Ethdkg: Distributed key generation with ethereum smart contracts. Cryptology ePrint Archive (2019)
53. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Advances in Cryptology—CRYPTO'89. Springer (1990)
54. Schoenmakers, B.: A simple publicly verifiable secret sharing scheme and its application to electronic voting. In: Annual International Cryptology Conference. pp. 148–164. Springer (1999)
55. Shamir, A.: How to share a secret. Communications of the ACM (1979)
56. Shrestha, N., Bhat, A., Kate, A., Nayak, K.: Synchronous distributed key generation without broadcasts. IACR Communications in Cryptology (2024)
57. Stadler, M.: Publicly verifiable secret sharing. In: EUROCRYPT '96. Springer (1996)
58. Supranational: BLST: Bls signatures. `https://github.com/supranational/blst` (2024)
59. Tassa, T.: Hierarchical threshold secret sharing. Journal of cryptology **20**, 237–264 (2007)
60. Tomescu, A., Chen, R., Zheng, Y., Abraham, I., Pinkas, B., Gueta, G.G., Devadas, S.: Towards scalable threshold cryptosystems. In: IEEE Symposium on Security and Privacy (SP) (2020)
61. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper (2014)
62. Zou, X., Maino, F., Bertino, E., Sui, Y., Wang, K., Li, F.: A new approach to weighted multi-secret sharing. In: 2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN). pp. 1–6. IEEE (2011)

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

Fig. 11: The low-degree test algorithm from SCRAPE [18].

A Additional Preliminaries

Shamir secret sharing. The Shamir secret sharing [55] embeds the secret s in the constant term of a polynomial $p(x) = s + a_1 x + a_2 x^2 + \cdots + a_d x^d$, where other coefficients $a_1, \cdots, a_d$ are chosen uniformly randomly from a field $\mathbb{F}$. The i-th share of the secret is $p(i)$, i.e., the polynomial evaluated at i. Given $d+1$ distinct shares, one can efficiently reconstruct the polynomial and the secret s using Lagrange interpolation. Also, s is information-theoretically hidden from an adversary that knows d or fewer shares. Throughout this paper, we use the notation $\mathsf{ShamirShare}(s, d, n)$ to denote an algorithm that outputs n Shamir shares of the secret s using a degree d polynomial.

Broadcast channels. Recall from §5, our DKG protocol (Figure 7) relies on a total order broadcast, and the DKG augmentation step (Figure 8) relies on a reliable broadcast. For completeness, we define them next.

Definition 8 (Total Order Broadcast). *In a distributed system with n parties $\{1, 2, \ldots, n\}$, 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.*

Definition 9 (Reliable Broadcast). *A* reliable broadcast *is a protocol that allows a designated party D, referred to as the sender, to broadcast a message to a set of n parties $\{1, 2, \ldots, n\}$. We use the convention that $D \in [n]$. A reliable broadcast protocol must satisfy the following properties.*

- Agreement. *If two honest parties i and j output m_i and m_j, respectively, then $m_i = m_j$.*
- Totality. *If an honest party outputs a message, then every honest party i eventually outputs a message.*
- Validity. *If the sender is honest, then every honest party i eventually outputs $m_i = m$.*

Low-degree test. Our PVSS scheme uses the low-degree test from [18] to check the degree of the committed polynomial. We describe this check in Figure 11. Here, to check that the committed polynomial is of degree at most t, a verifier multiplies the commitments in the exponent with a random word from the dual code and checks that the result is $1_{\mathbb{G}}$, i.e., the identity element of $\mathbb{G}$. This verification process is information-theoretically sound with an error probability of $1/|\mathbb{F}|$, where $\mathbb{F}$ is the scalar field of $\mathbb{G}$.

B Publicly Verifiable Secret Sharing (PVSS)

B.1 PVSS Definitions

Definition 10 (PVSS Correctness). *A PVSS scheme is correct if for all security parameters $\kappa \in \mathbb{N}$, n, t with $t < n$, subsets $S \subseteq [n]$ with $|S| \leq t$, secrets $s \in \mathbb{F}$, $\mathsf{pp} \leftarrow \mathsf{PVSS.Setup}(1^\kappa, n, t)$, and for $\{\mathsf{dk}_i, \mathsf{ek}_i \leftarrow \mathsf{PVSS.KeyGen}(i)\}_{i\in[n]}$, the following holds:*

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 $h := g^b$. Use lazy programming for $\mathsf{H} : \{0,1\}^* \to \hat{\mathbb{G}}$. Send $\mathsf{pp} := (h, \mathsf{H})$ to $\mathcal{A}$.
2. Let $\mathcal{C}, \mathcal{S} \subseteq [n]$ be the set of corrupt parties and with combined weight $W := \sum_{i \in \mathcal{C} \cup \mathcal{S}} w_i \leq w$. Let $\mathcal{H} := [n] \setminus (\mathcal{C} \cup \mathcal{S})$.
3. For each party $i \in \mathcal{C} \cup \mathcal{S}$ with weight w_i, sample $a_{i,j} \leftarrow\$ \mathbb{F}$ for each $j \in [w_i]$. Let $a(\cdot)$ be the degree w polynomial such that $a(0) = a$ and $a(s_i + j) = a_{i,j}$. (If $W = w$, then this polynomial is uniquely defined. Otherwise, choose at random $w - W$ additional values $a(j)$ for values $j > W$ in order to define the polynomial.)
4. For each $i \in \mathcal{H}$, sample $r_i \leftarrow\$ \mathbb{F}$, use $\mathsf{sk}_i := r_i$ and compute the public key vk_i as:

$$\mathsf{vk}_i := \{h^{r_i}, h^{r_i \cdot a_{i,1}}, \ldots, h^{r_i \cdot a_{i,w_i}}\}. \tag{7}$$

5. For each party $i \in [n] \setminus \mathcal{H}$, sample $u_i \leftarrow\$ \mathbb{F}$. It holds that $u_i = b \cdot r_i$ for some unknown $r_i \in \mathbb{F}$. Then, first compute $g^{a_{i,j}}$ for each $j \in [w_i]$ using interpolation in the exponent, and then compute vk_i as:

$$\mathsf{vk}_i := \{g^{u_i}, (g^{a_{i,1}})^{u_i}, \cdots, (g^{a_{i,w_i}})^{u_i}\} = \{h^{r_i}, h^{r_i a_{i,1}}, \cdots, h^{r_i \cdot a_{i,w_i}}\} \tag{8}$$

   here in equation (8) we use that $u_i = b \cdot r_i$ and $h = g^b$.
6. Send $\{\mathsf{sk}_i\}_{i \in \mathcal{C}}, \{\mathsf{vk}_i\}_{i \in [n]}$ to $\mathcal{A}$.

Simulating random oracle queries.

1. Let q_s be the upper bound on the number of VUF queries. Let $\eta := 1/(q_s + 1)$.
2. On the k-th random oracle query on a message m_k: if $\mathsf{H}(m_k) \neq \perp$, return $\mathsf{H}(m_k)$. Otherwise, sample $\mathsf{bit\text{-}map}[m_k] \leftarrow \mathsf{Ber}(\eta)$, where $\mathsf{Ber}(\eta)$ samples a bit with Bernoulli distribution with parameter η. Also, sample $\beta_k \leftarrow\$ \mathbb{F}$. Next,
   (a) If $\mathsf{bit\text{-}map}[m_k] = 0$, program $\mathsf{H}(m_k) := g^{\beta_k}$.
   (b) Alternatively, if $\mathsf{bit\text{-}map}[m_k] = 1$, program $\mathsf{H}(m_k) := g^{\beta_k \cdot c}$.

Simulating VUF queries.

1. On k-th partial signature query (i, m_k) for party $i \in \mathcal{S} \cup \mathcal{H}$, if $i \in \mathcal{S}$, follow the honest protocol. Otherwise,
   (a) If $\mathsf{bit\text{-}map}[m_k] = 0$, output $\sigma_i := (\hat{g}^b)^{\beta_k / u_i} = \mathsf{H}(m_k)^{b/u_i} = \mathsf{H}(m_k)^{1/r_i}$;
   (b) If $\mathsf{bit\text{-}map}[m_k] = 1$, abort.

Computing BDH solution.

Let (m^*, ρ^*) be $\mathcal{A}$'s output, then return $(\rho^*)^{1/\beta_k}$ as the BDH solution. Here β_k is such that $\mathsf{H}(m^*) = \hat{g}^{c \cdot \beta_k}$.

Fig. 12: Interaction of $\mathcal{A}_{\mathsf{bdh}}$ with $\mathcal{A}$ during weighted VUF security reduction

– $\Pr[\mathsf{PVSS.Verify}(\mathbf{ek}, \mathsf{PVSS.Share}(\mathbf{ek}, s)) = 1] = 1$
– $\Pr[\mathsf{PVSS.Recon}(\mathbf{ek}, \mathsf{trx}, S, \{\mathsf{dk}_i\}_{i \in S}) = h^s : \mathsf{trx} \leftarrow \mathsf{PVSS.Share}(\mathbf{ek}, s)] = 1$

Here, $\mathbf{ek} = [\mathsf{ek}_i]_{i \in [n]}$*, and* $h \in \mathbb{G}$ *is a generator and is part of* pp.

Definition 11 (PVSS Verifiability). *If* $\mathsf{PVSS.Verify}(\mathbf{ek}, \mathsf{trx})$ *accepts a transcript* $\mathsf{trx} = (\mathsf{com}, \{c_i, \pi_i\}_{i \in [n]})$*, then, with overwhelming probability, the* c_i*'s are encryptions of valid shares of some secret with the encryptions keys in* $\mathbf{ek}$*. 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 which requires that for every PPT adversary $\mathcal{A}$ that corrupts parties with combined weight upto w, there exists a PPT simulator $\mathsf{Sim}_{\mathsf{pvss}}$, that on input of a commitment com to a uniformly random secret $s \in \mathbb{F}$, produces a view $\mathcal{A}$ that is indistinguishable from $\mathcal{A}$'s view of a honestly generated PVSS transcript with s as the secret. We formalize this with the $\mathsf{PVSS\text{-}Sec}^{\mathcal{A}}$ game in Figure 13.

We note that there are alternative ways to define the secrecy property for a PVSS scheme (see, for example, [18,28]). We define secrecy based on how the secrecy property affects the overall security of the weighted VUF scheme.

Definition 12 (PVSS Secrecy). *Consider the game in Figure 13. For all security parameters* $\kappa \in \mathbb{N}$*,* n, t *with* $t < n$*, we say that a PVSS scheme ensures secrecy if for all PPT adversaries* $\mathcal{A}$*, the following holds:*

$$|\Pr[\mathsf{PVSS\text{-}Sec}^{\mathcal{A}}(1^\kappa, n, t, 0) = 1] - \Pr[\mathsf{PVSS\text{-}Sec}^{\mathcal{A}}(1^\kappa, n, t, 1) = 1]| = \mathsf{negl}(\kappa).$$

Game $\mathsf{PVSS\text{-}Sec}^{\mathcal{A}}(1^\kappa, n, t, b)$:

$\mathsf{pp} \leftarrow \mathsf{PVSS.Setup}(1^\kappa, n, t)$
// We assume that $\mathcal{A}$ is stateful and stores previous inputs
$\mathcal{C} \leftarrow \mathcal{A}(\mathsf{pp})$
if $|\mathcal{C}| \geq t$: **return** $\perp$
Let $\mathcal{H} := [n] \setminus \mathcal{C}$
Let $a \leftarrow\$ \mathbb{F}$
if $b = 0$:
    $((\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{PVSS.KeyGen}(i))_{i \in \mathcal{H}}$
    $(\mathsf{ek}_i, \mathsf{dk}_i)_{i \in \mathcal{C}} \leftarrow \mathcal{A}((\mathsf{ek}_i)_{i \in \mathcal{H}})$
    $\mathbf{ek} := [\mathsf{ek}_i]_{i \in [n]}$
    $\mathsf{trx} \leftarrow \mathsf{PVSS.Share}(\mathbf{ek}, a)$
    $\mathsf{com} := \mathsf{Commit}(a)$
else if $b = 1$:
    $(\mathsf{ek}_i)_{i \in \mathcal{H}} \leftarrow \mathsf{Sim}_{\mathsf{pvss}}(\mathsf{com}, \mathcal{C})$
    $(\mathsf{ek}_i, \mathsf{dk}_i)_{i \in \mathcal{C}} \leftarrow \mathcal{A}((\mathsf{ek}_i)_{i \in \mathcal{H}}))$
    $\mathbf{ek} := [\mathsf{ek}_i]_{i \in [n]}$
    $\mathsf{trx} \leftarrow \mathsf{Sim}_{\mathsf{pvss}}(\mathsf{com}, \mathbf{ek})$
$b' \leftarrow \mathcal{A}(\mathsf{trx})$
return b'

Fig. 13: PVSS Secrecy game.

Inputs: $\mathsf{pp} := (g, g^b, \hat{g}, n, t)$, $\mathcal{C}, \mathcal{H} := [n] \setminus \mathcal{C}$, and $\mathsf{com} := (V_0, \hat{V}_0) = (g^a, \hat{g}^a)$ for some $a \in \mathbb{F}$
DKG inputs. $\hat{g}, \hat{g}^b \in \hat{\mathbb{G}}$ // Needed for DKG simulation

Simulating key generation.

1. Let $h = g^b$ for some unknown $b \in \mathbb{F}$. This implies that, $h^a = g^{ab}$ is the shared secret. (Note that, $\mathsf{Sim}_{\mathsf{pvss}}$ cannot directly compute g^{ab}).
2. Let $a(\cdot) \in \mathbb{F}[x]$ be a polynomial of degree t such that $a(0) = a$. Compute $a(\cdot)$ as follows. For each malicious party $i \in \mathcal{C}$, sample $a(i) \leftarrow\$ \mathbb{F}$, and compute $(V_i, \hat{V}_i) := (g^{a(i)}, \hat{g}^{a(i)})$. If $|\mathcal{C}| < t$ then choose at random $t - |\mathcal{C}|$ additional values $a(j)$ for values $j > |\mathcal{C}|$ in order to define the polynomial.
3. For each honest party $i \in \mathcal{H}$, compute $(V_i, \hat{V}_i) := (g^{a(i)}, \hat{g}^{a(i)})$ using interpolation in the exponent.
4. For each honest party $i \in \mathcal{H}$, 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.

Transcript generation:

5. Let $(\mathsf{dk}_i, \mathsf{ek}_i) \in \mathbb{F} \times \mathbb{G}$ for each $i \in \mathcal{C}$, be the encryption and decryption of the corrupt parties. Extract, the decryption keys dk_i for each malicious party $i \in \mathcal{C}$ using the proof-of-knowledge extractor.

// Computing ciphertexts

6. Sample $r' \leftarrow\$ \mathbb{F}$ and set $C_0 := g^b g^{r'}$ and $\hat{R} := \hat{g}^b \hat{g}^{r'}$. This implicitly sets $r = \log_g R = b + r'$.
7. For each corrupt party $i \in \mathcal{C}$, 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 $i \in \mathcal{C}$, to compute these ciphertexts as $C_i := h^{a(i)} R^{\mathsf{dk}_i}$.
8. For each honest party $i \in \mathcal{H}$, compute its ciphertext C_i as:

$$C_i := R^{\theta_i} g^{-a(i)r'} = g^{\theta_i r - a(i)r'} = g^{(\theta_i - a(i))r + b \cdot a(i)}$$

   using $g^{a(i)}, r'$ and θ_i. Since $\mathsf{ek}_i = g^{\theta_i - a(i)}$, C_i is equal to $\mathsf{ek}_i^r \cdot g^{b \cdot a(i)} = \mathsf{ek}_i^r \cdot h^{a(i)}$.
9. Output: $\mathsf{trx} := (\hat{R}, \mathsf{pok}, [V_i]_{i \in [0,n]}, [\hat{V}_i]_{i \in [0,n]}, [C_i]_{i \in [n]})$ as the PVSS transcript.

Fig. 14: Secrecy simulator $\mathsf{Sim}_{\mathsf{pvss}}$ for our PVSS scheme in Figure 5.

B.2 Security of PVSS

Theorem 2. *Protocol 5 satisfies the secrecy property as per Definition 12.*

Proof. We describe the simulator $\mathsf{Sim}_{\mathsf{pvss}}$ that interacts with adversary $\mathcal{A}$ in Figure 14. The tricky parts of Figure 14 are: (i) $\mathsf{Sim}_{\mathsf{pvss}}$ extracting the decryption keys of malicious parties, and (ii) programming the random oracle at selected inputs to simulate the proof-of-knowledge on behalf of honest parties. Apart from these two steps, it is easy to see that the rest of the interaction is identical to the real protocol execution. We will now analyze these two non-trivial steps.

For (ii), $\mathsf{Sim}_{\mathsf{pvss}}$ needs to program the random oracle H at random inputs of its choice to successfully simulate the proofs-of-knowledge of honest parties. This does not introduce any additional error, even if $\mathcal{A}$ queries the random oracle at these inputs. This is because Sim's inputs are independent of $\mathcal{A}$'s actions. Hence,

$\mathsf{Sim}_{\mathsf{pvss}}$ can program the random oracle on its chosen inputs first and only then allow $\mathcal{A}$ to send queries to the random oracle.

For (i), $\mathsf{Sim}_{\mathsf{pvss}}$ needs to extract $O(n)$ decryption keys simultaneously. Note that the proof-of-knowledge protocol we use Σ-protocol (Figure 6). Thus, due to the multi-forking Lemma for Σ-protocols [4], $\mathsf{Sim}_{\mathsf{pvss}}$ can successfully extract all decryption keys, except with a negligible probability. We note that we can also use the Fischlin transformation [36,22] in an online manner.

Let $\mathsf{ExtFail}$ be the event that $\mathsf{Sim}_{\mathsf{pvss}}$ fails to extract all the decryption keys from parties in $\mathcal{C}$. Also, let $\varepsilon_{\mathsf{ext\text{-}fail}} := \Pr[\mathsf{ExtFail}]$ be the probability of the even $\mathsf{ExtFail}$. Then, we have that:

$$\Big|\Pr\Big[\mathsf{PVSS\text{-}Sec}^{\mathcal{A}}(1^\kappa, n, t, 0) = 1 | \neg\mathsf{ExtFail}\Big] - \Pr\Big[\mathsf{PVSS\text{-}Sec}^{\mathcal{A}}(1^\kappa, n, t, 1) = 1 | \neg\mathsf{ExtFail}\Big]\Big| = 0. \tag{9}$$

Let $\varepsilon_{\mathsf{ext\text{-}fail}}$ be the upper-bound on the probability that $\mathsf{Sim}_{\mathsf{pvss}}$ fails to extract all the decryption keys of parties in $\mathcal{C}$. Then, from equation (12), we get:

$$\Big|\Pr[\mathsf{PVSS\text{-}Sec}^{\mathcal{A}}(1^\kappa, n, t, 0) = 1] - \Pr[\mathsf{PVSS\text{-}Sec}^{\mathcal{A}}(1^\kappa, n, t, 1) = 1\Big| \leq \varepsilon_{\mathsf{ext\text{-}fail}} \qquad \square$$

C Weighted VUF Security with DKG

In this section, we prove that our weighted VUF is $\mathsf{UP\text{-}CMA}^{\mathcal{A}}$ secure as per Definition 5, when the game runs our DKG protocol in §5 with $\mathcal{A}$ to generate the secret keys. As in Section 3.4, the proof is based on assuming hardness of BDH in the random oracle model.

Theorem 3. *The weighted VUF protocol of Figure 4 is unpredictable as per Definition 5 when its keys are generated using the DKG protocol in §5.*

Proof. As in §3.4, we will prove this via a sequence of games. Game $\mathbf{G}_0$ is the $\mathsf{UP\text{-}CMA}^{\mathcal{A}}$ game, and game $\mathbf{G}_5$ is the interaction of $\mathcal{A}$ with $\mathcal{A}_{\mathsf{bdh}}$. Here on, for any game $\mathbf{G}_i$, we will use "$\mathbf{G}_i \Rightarrow 1$" as a shorthand for the event that a PPT adversary $\mathcal{A}$ predicts the VUF output in game $\mathbf{G}_i$.

GAME $\mathbf{G}_0$ **to** GAME $\mathbf{G}_2$: Similar to game $\mathbf{G}_0$ to $\mathbf{G}_2$ in §3.4, except the game runs the DKG protocol in Figure 7 with $\mathcal{A}$ instead of the KeyGen functionality, to generate the signing keys. Hence, by a similar argument as in §3.4, we get that:

$$\Pr[\mathbf{G}_2 \Rightarrow 1] \geq \frac{\varepsilon_{\mathsf{vuf}}}{4 \cdot q_s}. \tag{10}$$

Here, $\varepsilon_{\mathsf{vuf}}$ is the winning probability of $\mathcal{A}$ in the $\mathsf{UP\text{-}CMA}^{\mathcal{A}}$ game.

GAME $\mathbf{G}_3$: This game is identical to game $\mathbf{G}_2$, except for the following changes. Right after $\mathcal{A}$ specifies $\mathcal{C}$, the game randomly samples an honest party i', i.e., $i' \leftarrow\$ \mathcal{H} := [n]$. The game then interacts the with $\mathcal{A}$ as in game $\mathbf{G}_2$.

Let $\mathcal{Q} \subseteq [n]$ be the subset of parties whose PVSS transcripts are included in the aggregated transcript output by the DKG protocol. Then, if $i' \notin \mathcal{Q}$ the game aborts. Note that $\mathcal{Q}$ consists of parties with a combined weight greater than w, i.e., $\sum_{i\in\mathcal{Q}} w_i > \sum_{i\in\mathcal{C}} w_i$. Hence, $\mathcal{Q}$ consists of at least one honest party, i.e., $\mathcal{Q} \cap \mathcal{H} \neq \varnothing$. Since, i' is chosen uniformly at random, and $\mathcal{A}$'s view is independent of i', we have

$$\Pr[\mathbf{G}_3 \Rightarrow 1] \geq \frac{1}{|\mathcal{H}|} \cdot \Pr[\mathbf{G}_2 \Rightarrow 1] \tag{11}$$

GAME $\mathbf{G}_4$: This game is identical to game $\mathbf{G}_3$, except that the game aborts if it fails to extract (i) the decryption keys of the malicious parties, and (ii) the PVSS secrets shared by the malicious parties during the DKG protocol.

Note that the game needs to extract up to w decryption keys and up to t PVSS secrets simultaneously. The proof-of-knowledge protocol we use is a Σ-protocol (Figure 6), in which the knowledge-extractor relies

on rewinding. Therefore, to simultaneously extract both the decryption keys and the PVSS secrets, we need to rely on the multi-forking Lemma [4] in an manner similar to [23]. We also note that we can also use the Fischlin transformation [36,22] in an online manner.

Let $\mathsf{ExtFail}$ be the event that the game fails to extract all the decryption keys and PVSS secrets from parties in $\mathcal{C}$. Clearly, if the game does not abort, then the view of $\mathcal{A}$ in games $\mathbf{G}_3$ and $\mathbf{G}_4$ are identically distributed. Therefore, we have

$$\Pr\left[\mathbf{G}_3 \Rightarrow 1 | \neg \mathsf{ExtFail}\right] = \Pr\left[\mathbf{G}_4 \Rightarrow 1 | \neg \mathsf{ExtFail}\right] \tag{12}$$

Let $\varepsilon_{\mathsf{ext\text{-}fail}}$ be an upper-bound on the probability of the event $\mathsf{ExtFail}$, then from equation (12), we get:

$$|\Pr[\mathbf{G}_3 \Rightarrow 1] - \Pr[\mathbf{G}_4 \Rightarrow 1]| \leq \varepsilon_{\mathsf{ext\text{-}fail}}. \tag{13}$$

GAME $\mathbf{G}_5$: This game is identical to game $\mathbf{G}_4$, except that the game uses $\mathsf{Sim}_{\mathsf{pvss}}$ to simulate the DKG protocol with $\mathcal{A}$. We decribe these changes in four parts: key registration, DKG simulation, DKG augmentation simulation, and VUF evaluation simulation. Let $\boldsymbol{w} = [w_1, \ldots, w_n]$ be the weight distribution

Key registration: The game samples $a \leftarrow\!\$\ \mathbb{F}$. It then generates the encryption keys of all honest parties by running the key-generation part of $\mathsf{Sim}_{\mathsf{pvss}}$ (steps 1-4 in Figure 14) on behalf of party i' with $\mathsf{com} := (g^a, \hat{g}^a)$ as the commitment. Recall that we choose party i' in game $\mathbf{G}_3$.

Let $\mathbf{ek}_{\mathcal{H}} = [\mathsf{ek}'_i]_{i \in \mathcal{H}}$ be the PVSS encryption keys of the honest parties, where $\mathsf{ek}'_i := [\mathsf{ek}_{s_i+1}, \ldots, \mathsf{ek}_{s_i+w_i}]$ and $s_i := \sum_{j \in [i-1]} w_j$. Send, $\mathbf{ek}'$ to $\mathcal{A}$. Let $\mathbf{ek}_{\mathcal{C}} := [\mathsf{ek}_i]_{i \in \mathcal{C}}$ be the encryption keys output by $\mathcal{A}$. Let $\mathbf{ek} := \mathbf{ek}_{\mathcal{H}} \| \mathbf{ek}_{\mathcal{C}}$ be the encryption keys of all parties. Here we use $\|$ to denote the concatenation operation.

DKG simulation: First, for party i', we generate its PVSS transcript $\mathsf{trx}_{i'}$ by running the transcript-generation part of $\mathsf{Sim}_{\mathsf{pvss}}$ (steps 5-9 in Figure 14). Note that by running $\mathsf{Sim}_{\mathsf{pvss}}$ on behalf of party i', we generate the simulated transcript for the secret h^a. Next, for all parties $i \in \mathcal{H} \setminus \{i'\}$, we honestly generate the PVSS transcript. Next, we honestly participate in the rest of the DKG protocol.

Let $\mathcal{Q} \subseteq [n]$ be the subset of parties whose PVSS transcripts are included in the aggregated transcript output by the DKG protocol. Let $a(\cdot)$ be the aggregated polynomial, i.e., $a(x) = \sum_{i \in \mathcal{Q}} a_i(x)$. Here $a_i(x)$ is the polynomial shared by party i during the DKG. The shared secret is $h^{a(0)}$.

Recall from game $\mathbf{G}_3$ that we have that $i' \in \mathcal{Q}$. Therefore, we can write $a(0) = a + a_{\mathcal{H}} + a_{\mathcal{C}}$ for some $(a_{\mathcal{H}}, a_{\mathcal{C}}) \in \mathbb{F}^2$. Here $a_{\mathcal{H}}$ is the sum of the secrets shared by the dealers in $Q \cap \mathcal{H} \setminus \{i'\}$. Similarly, $a_{\mathcal{C}}$ is the sum of the secrets shared by the dealers in $Q \cap \mathcal{C}$. Clearly, the game knows $a_{\mathcal{H}}$ since it sampled these locally, and the game can compute $a_{\mathcal{C}}$ using the PVSS secrets it extracts in game $\mathbf{G}_4$.

DKG augmentation simulation: We generate the augmented keys on behalf of each honest party exactly as in §3.4. More specifically, for each party $i \in \mathcal{H}$, we sample $u_i \leftarrow\!\$\ \mathbb{F}$. It holds that $u_i = b \cdot r_i$ for some unknown $r_i \in \mathbb{F}$. Next, we compute $g^{a_{i,j}}$ for each $j \in [w_i]$ using interpolation in the exponent, and compute vk_i as:

$$\mathsf{vk}_i := \{g^{u_i}, (g^{a_{i,1}})^{u_i}, \cdots, (g^{a_{i,w_i}})^{u_i}\} = \{h^{r_i}, h^{r_i a_{i,1}}, \cdots, h^{r_i \cdot a_{i,w_i}}\} \tag{14}$$

here in equation (14) we use the fact that $u_i = b \cdot r_i$ and $h = g^b$.

VUF evaluation simulation: We simulate the VUF evaluation queries exactly as in Figure 12. More precisely, on the k-th partial VUF query (i, m_k) for party $i \in \mathcal{S} \cup \mathcal{H}$, the game simulates the query as follows.

1. If $i \in \mathcal{S}$, follow the honest protocol.
2. For $i \in \mathcal{H}$, if $\mathsf{bit\text{-}map}[m_k] = 0$, output $\sigma_i := (\hat{g}^b)^{\beta_k / u_i} = \mathsf{H}(m_k)^{b/u_i} = \mathsf{H}(m_k)^{1/r_i}$; Otherwise, abort.

From the proof of Theorem 2 (see equation (12)), we get that conditioned on the game having already extracted the decryption keys of parties in $\mathcal{C}$, the simulated PVSS transcript of party i' is identically distributed as the real PVSS transcript. Since the game follows the honest DKG protocol for all honest parties except i', the view of $\mathcal{A}$ during the DKG protocol during game $\mathbf{G}_5$ is identically distributed as in game $\mathbf{G}_4$. Moreover, using an argument similar the proof of Theorem 1, we have that the distribution of the secret keys $\{\mathsf{sk}_i\}_{i \in \mathcal{C} \cup \mathcal{S}}$, the public keys $\{\mathsf{vk}_i\}_{i \in [n]}$, and the VUF shares is identical to the distribution in game $\mathbf{G}_4$. Therefore, combining all the above, we get that $\Pr[\mathbf{G}_4 \Rightarrow 1] = \Pr[\mathbf{G}_5 \Rightarrow 1]$.

Combining all the above, we get that:

$$\Pr[\mathbf{G}_5 \Rightarrow 1] \geq \frac{1}{4 \cdot |\mathcal{H}| \cdot q_s} \cdot \Pr[\mathbf{G}_0 \Rightarrow 1] - \varepsilon_{\mathsf{ext\text{-}fail}}$$
$$\implies \Pr[\mathbf{G}_5 \Rightarrow 1] \geq \frac{\varepsilon_{\mathsf{vuf}}}{4 \cdot |\mathcal{H}| \cdot q_s} - \varepsilon_{\mathsf{ext\text{-}fail}} \tag{15}$$

We next argue that whenever $\mathcal{A}$ wins in $\mathbf{G}_5$, we can use $\mathcal{A}$ to build an adversary $\mathcal{A}_{\mathsf{bdh}}$ to break the BDH assumption. We summarize $\mathcal{A}_{\mathsf{bdh}}$'s interaction with $\mathcal{A}$ in Figure 12, and describe the critical points next.

1. $\mathcal{A}_{\mathsf{bdh}}$ on input a BDH tuple $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c)$ uses $h := g^b$ and implicitly uses h^a as the PVSS secret of party i'.
2. $\mathcal{A}_{\mathsf{bdh}}$ uses $\hat{g}^c$ to program the random oracle as in equation (3)

Let (m^*, ρ^*) be the output of $\mathcal{A}$ during its interaction with $\mathcal{A}_{\mathsf{bdh}}$. Then, $\mathcal{A}_{\mathsf{bdh}}$ outputs $(\rho^*)^{1/\beta_k} \cdot e((g^b)^{a_{\mathcal{H}}+a_{\mathcal{C}}}, \hat{g}^{-c})$ as the BDH solution.

Next, we argue that $(\rho^*)^{1/\beta_k} \cdot e((g^b)^{a_{\mathcal{H}}+a_{\mathcal{C}}}, \hat{g}^{-c})$ is the correct BDH solution. Note that since ρ^* is a valid VUF output and $h = g^b$, we have that:

$$\rho^* = e(h^{a(0)}, \mathsf{H}(m^*)) = e(h^{(a+a_{\mathcal{H}}+a_{\mathcal{A}})}, \hat{g}^{\beta_k \cdot c}) = e(g^{b(a+a_{\mathcal{H}}+a_{\mathcal{A}})}, \hat{g}^{\beta_k \cdot c})$$
$$\implies (\rho^*)^{1/\beta_k} \cdot e((g^b)^{(a_{\mathcal{H}}+a_{\mathcal{A}})}, \hat{g}^{-c}) = e(g, \hat{g})^{abc} \tag{16}$$

From equation (16), we get that, whenever $\mathcal{A}$ wins game $\mathbf{G}_5$, $\mathcal{A}_{\mathsf{bdh}}$ can break BDH, hence we have:

$$\varepsilon_{\mathsf{bdh}} \geq \Pr[\mathbf{G}_5 \Rightarrow 1] \implies \varepsilon_{\mathsf{vuf}} \leq 4 \cdot |\mathcal{H}| \cdot q_s \cdot (\varepsilon_{\mathsf{bdh}} + \varepsilon_{\mathsf{ext\text{-}fail}}) \qquad \square$$

Inputs: $(g, g^a, g^b, \hat{g}, \hat{g}^a, \hat{g}^b, \hat{g}^c) \in \mathbb{G}^3 \times \hat{\mathbb{G}}^4$ for some $a, b, c \leftarrow\$ \mathbb{F}$, and $(n, \boldsymbol{w}, w)$.

Setup and DKG Simulation:

1. Set $h := g^b$. Use lazy programming for $\mathsf{H} : \{0,1\}^* \to \hat{\mathbb{G}}$. Send $\mathsf{pp} := (g, h, \hat{g}, \mathsf{H})$ to $\mathcal{A}$.
2. Receive $\mathcal{C}, \mathcal{S} \subseteq [n]$ from $\mathcal{A}$ where $W := \sum_{i \in \mathcal{C} \cup \mathcal{S}} w_i \leq w$. Let $\mathcal{H} := [n] \setminus (\mathcal{C} \cup \mathcal{S})$.
3. Sample one honest party $i' \in \mathcal{H}$ uniformly at random.
4. Run $\mathsf{Sim}_{\mathsf{pvss}}$ on input $(g^a, \hat{g}^a)$ to generate the encryption keys of honest signers. Send the encryption keys of honest parties $\mathbf{ek}_{\mathcal{H}} := \{\mathsf{ek}_i\}_{i \in \mathcal{H} \cup \mathcal{S}}$ to $\mathcal{A}$.
5. Wait to receive $\mathbf{ek}_{\mathcal{A}} := \{\mathsf{ek}_i\}_{i \in \mathcal{C}}$ from $\mathcal{A}$. Extract $\mathsf{dk}_i \in \mathbb{F}$ for each $i \in \mathcal{C}$ using the proof-of-knowledge (PoK) extractor. Abort, if the PoK extractor fails.
6. Send $\{\mathsf{sk}_i\}_{i \in \mathcal{C}}, \{\mathsf{vk}_i\}_{i \in [n]}$ to $\mathcal{A}$.

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 by running $\mathsf{Sim}_{\mathsf{pvss}}$.
9. Honestly participate in the rest of the DKG protocol. Let $\mathcal{Q}$ be the set of parties whose transcripts are aggregated by the DKG protocol. Abort, if $i' \notin \mathcal{Q}$.
10. While simulating the DKG protocol, extract the PVSS secrets shares by the $\mathcal{A}$, using the PoK extractor. Abort, if the PoK extractor fails.
11. Let $a(\cdot)$ be the aggregated polynomial, i.e., $a(x) = \sum_{i \in \mathcal{Q}} a_i(x)$. Here $a_i(x)$ is the polynomial shared by party i during the DKG. The shared secret is $h^{a(0)}$.
12. Let $a_{\mathcal{H}} := \sum_{i \in \mathcal{Q} \cap \mathcal{H} \setminus \{i'\}} a_i(0)$. Similarly, let $a_{\mathcal{C}} := \sum_{i \in \mathcal{Q} \cap \mathcal{C}} a_i(0)$. Note that by definition we have $a = a_{i'}(0)$. Next, since, $i' \in \mathcal{Q}$, we have $a(0) = a + a_{\mathcal{H}} + a_{\mathcal{C}}$.

DKG augmentation simulation:

13. $\mathcal{A}_{\mathsf{bdh}}$ generates the augmented public keys of honest parties as in the VUF simulation in Figure 12. Note that $\mathcal{A}_{\mathsf{bdh}}$ learns $\boldsymbol{V} = [g^{a(0)}, g^{a(1)}, \ldots, g^{a(W)}]$ and $\hat{\boldsymbol{V}} = [\hat{g}^{a(0)}, \hat{g}^{a(1)}, \ldots, \hat{g}^{a(W)}]$ that are needed during step (3) in Figure 12, as part of the DKG output.

VUF evaluation simulation:

// Identical to Figure 12

Computing the BDH solution.

Let (m^*, ρ^*) be $\mathcal{A}$'s VUF output for the message $m_{k'}$, then output $(\rho^*)^{1/\beta_k} \cdot e((g^b)^{a_{\mathcal{H}} + a_{\mathcal{C}}}, \hat{g}^{-c})$ as the BDH solution.

Fig. 15: The interaction of $\mathcal{A}_{\mathsf{bdh}}$ with $\mathcal{A}$ to break the BDH assumption.