Quadratic Asynchronous DKG from Plain Setup

Ittai Abraham[1], Renas Bacho[2], and Gilad Stern*[3,4]

[1]a16z
[2]Ruhr University Bochum
[3]Tel Aviv University
[4]Hebrew University

Abstract

Asynchronous distributed key generation (ADKG) is a fundamental primitive for building threshold cryptosystems and fault-tolerant distributed protocols in adversarial network settings. A central objective in this line of work is to achieve ADKG with $O(n^2)$ communication and constant round complexity under minimal setup assumptions. Recent progress has led to subcubic-communication ADKG protocols under different trade-offs. Feng and Tang (CRYPTO 2025) presented an ADKG protocol with $O(n^2)$ communication and $O(1)$ rounds, but at the cost of a cubic-communication setup phase in which each party posts a linear-sized public key on a public bulletin board. In contrast, Abraham et al. (PODC 2026) achieved an ADKG protocol with a standard setup phase, where each party posts only a constant-sized public key, but with $O(n^{2+1/k})$ communication and $O(k)$ rounds for a tunable parameter $k \leq \log n$. These results leave open whether one can simultaneously obtain constant-round complexity and quadratic communication under a standard setup phase.

In this work, we resolve this open problem by presenting the first ADKG protocol that achieves $O(n^2)$ communication and $O(1)$ rounds while requiring only a standard setup phase in which each party posts a constant-sized public key on a public bulletin board. Our protocol is resilient to a strongly adaptive adversary corrupting up to $f < n/3$ parties and assumes only random oracles and secure erasures, both of which are also assumed by prior subcubic-communication ADKG constructions.

1 Introduction

Distributed key generation (DKG) [Ped91] allows a set of n parties to jointly generate a public-secret key pair $(\mathsf{pk}, \mathsf{sk})$ such that the secret key is never reconstructed in full but instead shared among the parties using a threshold secret sharing scheme. Cryptographic operations requiring the secret key can then be performed by a qualified threshold of parties, but not by fewer. DKG is a fundamental primitive in threshold cryptography, with key applications including threshold encryption and threshold signature schemes, and it is widely used in Byzantine fault-tolerant (BFT) distributed protocols to reduce both communication and round complexity [YMR+19, GKKS+22]. In many real-world systems, communication delays are unpredictable, and message delivery cannot be assumed to be synchronous or to occur within known time bounds. This has motivated a substantial line of work on DKG in the asynchronous network model [KKMS20, AJM+21], referred to as *asynchronous DKG (ADKG)*, where messages may be arbitrarily delayed, but must eventually be delivered in finite time. Over the last several years, ADKG has been studied extensively, leading to protocols that tolerate an optimal number of Byzantine faults, namely up to $f < n/3$ corrupted parties.

*Gilad Stern is partially supported by the Israel Science Foundation (Grant No. 1336/22) and by the European Union (ERC, FTRC, 101043243). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

An important part of this line of work is improving the efficiency of such protocols, allowing cheap set-up costs for future protocols. Recent progress has brought the communication cost of ADKG below the long-standing cubic barrier in the number of parties n, albeit with different trade-offs. Denoting by λ the security parameter, Feng and Tang [FT24, FGL+25] presented an ADKG protocol with $O(\lambda^2 n^2)$ communication and $O(1)$ rounds, but at the cost of a cubic-communication setup phase in which each party posts a linear-sized public key on a public bulletin board.[1] In contrast, Abraham et al. [ABLS25] achieved an ADKG protocol with a standard setup phase, where each party posts only a constant-sized public key, but with $O(\lambda^2 n^{2+1/k})$ communication and $O(k)$ rounds for a tunable parameter $1 \leq k \leq \log n$. These results leave open whether one can simultaneously obtain quadratic communication and constant-round complexity under a standard setup phase.

1.1 Our Contribution

In this work, we resolve this open problem by presenting the first asynchronous DKG protocol that simultaneously achieves:

- **Security:** Our protocol is resilient to a strongly adaptive adversary that can corrupt up to $f < n/3$ parties. This is optimal for asynchronous DKG.
- **Efficiency:** Our protocol has an expected communication cost of $O(\lambda^2 n^2)$ bits and terminates in expected $O(1)$ rounds, where λ denotes the security parameter.
- **Setup:** Our protocol assumes a plain public-key infrastructure, in which each party posts a constant-sized public key on a public bulletin board. Moreover, it assumes random oracles and secure erasures, both of which are also assumed in prior subcubic-communication ADKG constructions.

At a high level, our construction combines ideas from recent subcubic ADKG protocols with new techniques that eliminate the need for a cubic-communication setup phase. We believe that the insights underlying our approach may be of independent interest. We provide a comparison with existing asynchronous DKG protocols in Table 1.

Furthermore, we discuss more related work on DKG in Section 2.8, and refer to [BK25] for a systematization of the extensive DKG literature.

2 Technical Overview

We briefly review the standard cubic-communication approaches and the two recent subcubic ADKG protocols [FT24, ABLS25]. Then, we outline our novel ideas that allow us to achieve the desired properties without a silent setup.

2.1 Cubic-Communication Approach

We briefly recall the high-level structure that underlies essentially all existing cubic-communication asynchronous DKG protocols. A DKG protocol among a set of n parties, of which up to f may be malicious, aims to generate a secret key sk that is shared among the parties via a (t, n)-threshold secret sharing scheme, so that each party i obtains a share sk_i.[2]

Generic structure. From a structural point of view, nearly all known ADKG protocols decompose into two conceptual components: a *sharing phase*, in which each party verifiably shares a secret, and a *consensus phase*, which determines which of these sharings contribute to the final key. Concretely, each party $i \in [n]$ samples a random value s_i and acts as a dealer in an asynchronous verifiable secret sharing (AVSS) instance, ensuring that every party j eventually obtains a share $s_{i,j}$ whenever dealer i's AVSS instance completes.

[1] Feng and Tang [FT24] design an erasure-free common coin protocol, while Feng et al. [FGL+25] present a generic erasure-based, coin-aided ADKG framework. Feng and Tang instantiate the framework of [FGL+25] with their coin protocol to obtain their ADKG protocol.

[2] In asynchrony, protocols may use a dual threshold $t \in [f, n-f)$, whereas in synchrony one typically sets $t = f < n/2$.

Table 1: Comparison of existing asynchronous DKG protocols.

Protocol	Resilience	Communication	Rounds	Setup Cost	Setup	Additional Notes
Kokoris et al. [KKMS20]	$f < n/3$	$O(\lambda n^4)$	$O(n)$	$O(\lambda n^2)$	RO, PKI	without pairings
Abraham et al. [AJM+21]	$f < n/3$	$O(\lambda n^3 \log n)$	$O(1)$	$O(\lambda n^2)$	RO, PKI	$O(1)$-round MVBA
Gao et al. [GLL+21]	$f < n/3$	$O(\lambda n^3)$	$O(1)$	$O(\lambda n^2)$	RO, PKI	variant of [AJM+21]
Das et al. [DYX+22]	$f < n/3$	$O(\lambda n^3)$	$O(\log n)$	$O(\lambda n^2)$	RO, PKI	verifiable encryption
Das et al. [DXKKR23]	$f < n/3$	$O(\lambda n^3)$	$O(\log n)$	$O(\lambda n^2)$	RO, PKI	without pairings
Zhang et al. [ZDL+23]	$f < n/3$	$O(\lambda n^4)$	$O(1)$	None	Auth. Channels	coll-resistant hashes
Abraham et al. [AJM+23]	$f < n/3$	$O(\lambda n^3)$	$O(1)$	$O(\lambda n^3)$	RO, PKI, PoT	optimal packed AVSS
Feng et al. [FT24, FGL+25]	$f < n/3$	$O(\lambda^2 n^2)$	$O(1)$	$O(\lambda n^3)$	RO, PKI, Silent	SNARKs, erasures
Abraham et al. [ABLS25]	$f < n/3$	$O(\lambda^2 n^{2+1/k})$	$O(k)$	$O(\lambda n^2)$	RO, PKI	SNARKs, erasures
Our ADKG	$f < n/3$	$O(\lambda^2 n^2)$	$O(1)$	$O(\lambda n^2)$	RO, PKI	SNARKs, erasures

Note. Communication and round complexity are given in expected numbers, as all these protocols are randomized. On **Setup Cost**: The silent setup in [FT24] requires each party to register an $O(\lambda n)$-bit sized public key and thus read $\Omega(\lambda n^2)$ bits during setup, yielding $\Omega(\lambda n^3)$ total setup communication cost. Moreover, the silent setup requires a powers-of-tau (PoT) setup, which itself requires $O(\lambda n^3)$ communication and $O(\log n)$ rounds in asynchrony [DXR24]. The other protocols (under plain PKI) register $O(\lambda)$-sized public keys during setup. Our protocol and [FT24, FGL+25, ABLS25] require erasures only for adaptive security. In [ABLS25], $k \leq \log n$ is a parameter fixed before protocol execution.

Due to asynchrony, different AVSS instances may complete at different times at different honest parties, so each party j maintains a local set $T_j \subseteq [n]$ of dealers whose AVSS instances it has completed. The purpose of the consensus phase is to reconcile these divergent local views by agreeing on a common subset $\mathcal{Q} \subseteq [n]$ of at least $f + 1$ dealers whose secrets are included in the final key. The resulting secret key is implicitly defined as $\mathsf{sk} := \sum_{i \in \mathcal{Q}} s_i$, with each party locally computing its share as $\mathsf{sk}_j := \sum_{i \in \mathcal{Q}} s_{i,j}$. The requirement that $|\mathcal{Q}| \geq f + 1$ guarantees that at least one honest secret contributes to the key, ensuring secrecy against an adversary corrupting at most f parties.

Communication bottleneck. This generic blueprint immediately exposes a fundamental efficiency barrier faced by ADKG protocols. The sharing phase consists of n independent AVSS executions, one per dealer. Since any AVSS protocol tolerating $f < n/3$ Byzantine faults necessarily requires $\Omega(n^2)$ communication per instance [ACD+23, BDK13], the sharing phase alone already incurs a total cost of $\Omega(n^3)$ communication. This explains why all classical ADKG constructions exhibit cubic or higher overall communication complexity, even before accounting for the cost of the consensus phase.

2.2 Recent Subcubic Approaches

We briefly recall the main ideas underlying the recent subcubic ADKG protocols in [FT24, ABLS25].

2.2.1 The Approach in [ABLS25].

Their construction departs from the classical AVSS-based paradigm by replacing the n parallel AVSS instances with a more compact committee-based sharing mechanism and a recursive consensus protocol driven by a weak coin.

Committee-based generation of hidden inputs. Instead of having each party directly contribute a secret, their protocol associates with every party i a small, randomly chosen hidden committee of $O(\lambda)$ parties. Each committee member locally generates an $O(\lambda n)$-sized PVSS transcript[3] for a random secret and sends it to $O(\lambda)$ other parties selected using a verifiable random function (VRF). Once a party has

[3] A publicly verifiable secret sharing (PVSS) scheme [Sta96] allows a dealer to threshold-share a secret s via a single public transcript Trx that anyone can verify for correctness, assuming only a plain PKI setup. Correctness means that each party i can extract its share s_i of s from Trx.

collected sufficiently many such transcripts, it aggregates them into a single PVSS transcript and disperses this aggregate using a provable information-dispersal[4] protocol. Crucially, as long as at least one honest committee member contributes to the aggregate, the resulting transcript is good: its embedded secret is uniformly random and hidden from the adversary. After dispersing an aggregated PVSS transcript, a party i multicasts (i.e., sends to all parties) a succinct proof π_i—including its identifier i—attesting that it has correctly dispersed a good transcript. Parties forward such proofs to ensure that all parties learn of at least one successful dispersal, even if they themselves did not manage to collect enough PVSS transcripts to form a good aggregate. The overall cost of this phase is $O(\lambda^2 n^2)$ communication and $O(1)$ rounds.

Recursive consensus via weak coin. To agree on one of the dispersed aggregated transcripts, the protocol employs a validated asynchronous Byzantine agreement (VABA) protocol whose liveness relies on a weak coin. The weak coin is implemented using a recursive construction. The set of parties is hierarchically partitioned into smaller subsets, forming a tree of degree $\ell := n^{1/k}$ and depth $k \leq \log n$. At each level, parties run a local consensus protocol within each subset—using a weak coin derived recursively from lower levels—to decide on a single candidate aggregated transcript, yielding at most ℓ candidates overall, one per child subset. Parties then execute a Weighted Gather protocol, similar to the Gather protocol in [AJM+21], that establishes a common core set V^* of candidates with large honest support. Each candidate transcript yields a random seed that is used to derive weight-many random ranks. The child whose transcript produces the highest rank is selected, and its transcript is then used once more to derive the final coin value r^*. With constant probability, the winning rank comes from the common core V^*, in which case all honest parties select the same transcript and hence agree on the same coin output.

2.2.2 The Approach in [FT24].

We now recall the construction of Feng and Tang.

Silent setup phase. Their work is built on Garg et al.'s silent threshold encryption (STE) [GKPW24]. In a silent setup, each party posts an $O(\lambda n)$ bit-sized public key on the public bulletin board. After verifying (and reading) the parties' linear-sized public keys against the powers-of-tau reference string [DXR24], each party can aggregate them into a single $O(\lambda n)$-sized group public key pk, which suffices for the silent-setup threshold encryption scheme. Feng and Tang extend STE into silent threshold encryption with tag-homomorphism (STHE), which supports homomorphic aggregation of ciphertexts encrypted under the same tag and tag-based threshold decryption.

Initial phase. As is common in consensus constructions, Feng and Tang construct a weak coin protocol and use it for their consensus phase. A common way to construct such a protocol involves every party secret sharing a random value for every other party. Parties then personally choose a subset of secrets, and then uses their sum as a random value.

In Feng and Tang's construction, instead of running n AVSS instances to commit to secrets, each party i encrypts a fresh random value under the STHE group public key pk and multicasts the $O(\lambda)$-sized ciphertext i together with a succinct proof of well-formedness π_i. A party i then aggregates $f+1$ such ciphertexts into a committed candidate coin c_i^*, accompanied by a succinct proof π_i^* showing correct aggregation of $f+1$ well-formed ciphertexts. This ensures that: (i) the aggregated ciphertext is decryptable later, and (ii) the encrypted value is uniformly random and hidden from the adversary.

Weak coin generation. To avoid the cubic communication overhead of classical Gather, Feng and Tang introduce a weaker primitive called *Align*. Align does not require all honest parties to output identical sets; instead, it guarantees (i) the existence of a well-covered subset of $n-f$ valid candidates, each observed by at least $f+1$ honest parties, and (ii) a binding cover, meaning that all valid candidates are fixed at the moment the first honest party outputs. Align is realized using several chained instances of provable broadcast [AS22], achieving $O(|m|n^2)$ communication for input size $|m|$.

[4] A provable information dispersal protocol [LLTW20] enables a sender to disperse data among parties together with a succinct proof that enough correct fragments were delivered to allow later reconstruction.

After Align, each party partially decrypts the candidates it observes using STHE and ranks them deterministically according to a random-oracle-based ranking rule. Each party locally selects the highest-ranked candidate it can open. With constant probability, the globally highest-ranked candidate lies in the well-covered subset and is therefore opened by at least $f+1$ honest parties, which suffices for all honest parties to agree on the same coin value. This yields a weak common coin with quadratic communication.

From weak coin to ADKG. This weak coin construction can be supplied to a quadratic-communication asynchronous consensus protocol [MR17], which makes black-box use of weak coins. The resulting protocol enables parties to agree on which committed candidate to open and to threshold-decrypt it using STHE to obtain a common unpredictable value (i.e., a strong coin). Finally, by plugging this strong coin into a recent coin-aided ADKG framework [FGL+25], they obtain an ADKG protocol with the desired properties.

2.3 Our Techniques: Weak Leader Election from Key Escrow

A weak leader election protocol allows all honest parties, with constant probability, to agree on the identity of a single party $i^* \in [n]$ (the leader), such that the leader is unpredictable to the adversary in advance and is honest with constant probability at the time of protocol initiation. Similarly, a weak coin outputs an unpredictable random value $\rho^* \in \{0,1\}^\lambda$ (or a bit) with constant probability of agreement. While seeming like a weaker primitive, weak leader election suffices for protocols that require selecting a proposer or dealer, e.g., in VABA or Byzantine fault-tolerant (BFT) leader rotation. In addition, electing a single honest party is at the core of many shared randomness constructions, such as common coin protocols.

To construct a weak leader election protocol, suppose we have a protocol, denoted *Key Escrow*, with the following properties. Each party $i \in [n]$ outputs an encryption key ek_i together with a succinct proof π_i attesting that the corresponding decryption key dk_i remains unknown to *any* set of f parties, including party i itself. At the same time, there exists an interactive protocol—denoted *Key Retrieve*—that allows any party i to reconstruct dk_i after any $f+1$ parties authorize that reconstruction. Using such a pair of protocols, we obtain the following *weak leader election* protocol in two phases.

Phase 1: Escrowed Key setup and encrypted rank dissemination. All parties first run the *Key Escrow* setup protocol, and each party i outputs an encryption key ek_i together with a proof π_i as described above. Party i multicasts the pair (ek_i, π_i) to all parties. Each party j computes its own rank $r_j := \mathsf{VRF}(\mathsf{sk}_j, tag)$ using a VRF[5] with the goal of disseminating its rank. Eventually, parties will elect the party with the highest rank.

Upon receiving a valid encryption key and proof (ek_i, π_i) from party i, party j encrypts its rank using i's key and sends the resulting ciphertext $c_{j,i} = \mathsf{Encrypt}(\mathsf{ek}_i, r_j)$ to i. Upon receiving $c_{j,i}$, party i sends party j a *stored* message confirming that it has received and stored the ciphertext. Each party j waits until it has $n-f$ parties sent it stored messages, and then multicasts a *ready* message. Each party that sends a ready message knows that its rank is stored by $f+1$ honest parties. Parties wait to hear $n-f$ such ready messages, guaranteeing that there exists a core of $f+1$ honest parties whose ranks are each stored by some set of $f+1$ honest parties.

To avoid the additional setup required for VRFs, we construct an interactive analogue of VRFs using a similar construction to that of the Key Escrow protocol, which we describe in Section 2.5.

Phase 2: Decrypt ranks and select winner. This phase consists of two parts. Each party i starts by invoking the *Key Retrieve* protocol to reconstruct its decryption key dk_i. This allows it to decrypt all ciphertexts it has stored and selects the index j^* with the highest rank r_{j^*} (among those it has opened and whose accompanying VRF proof μ_{j^*} verifies). In the second part, each party multicasts their local winner $(r_{j^*}, \mu_{j^*}, j^*)$ to all parties. Parties then wait for $n-f$ such winning candidates and output the index with the maximal rank among them as the final weak leader election outcome.

Security Intuition. At a high level, the protocol is secure due to two main reasons. Binding: By the end of Phase 1, the adversary must allow a core of $f+1$ parties honest to disseminate their encyrpted ranks to

[5] We assume that r_j implicitly includes the VRF proof μ_j of correct evaluation. Here, sk_j denotes the VRF secret key of party j, and tag can be viewed as a round or instance identifier (e.g., $tag = r$ for the r-th instance of Key Escrow).

$f + 1$ honest parties. This means that if one of them has the maximal rank, everybody will see that it is the winner. Hiding: At the time the adversary makes this choice, it does not know any of its decryption keys. This means that it does not know the ranks of the honest parties, and thus the choice of the core set is independent of the ranks themselves.

Since the core of honest parties is of size $f + 1 > n/3$, there is at least a $\frac{1}{3}$ probability that the winner is an honest party in the core, as required by a weak leader election protocol.

Efficiency. Excluding the cost of the Key Escrow and Key Retrieve protocol pair, parties simply send each other keys, encryptions, and proofs of size $O(\lambda)$ for a constant number of rounds. This results in $O(\lambda n^2)$ communication complexity and $O(1)$ round complexity.

2.4 Our Techniques: Key Escrow

Our Key Escrow protocol is implemented by having each party obtain an aggregated PVSS transcript that contains a public encryption key, together with sufficient information to reconstruct the secret key with the help of the network. In our implementation, each transcript has size $O(\lambda n)$.

In order to have every honest party obtain such a transcript, we have two steps: first, at least one honest party obtains a transcript, and following that, every party obtains a transcript. To implement the first step, parties call the PVSSExchange protocol of [ABLS25], which incurs $O(\lambda^2 n^2)$ communication costs. In the second step, each party that holds a transcript disseminates it in a call to a Disperse protocol and holds a proof that it has been disseminated. This proof guarantees that a valid aggregated transcript can be retrieved by any other party. Parties then forward these proofs to the network. Each party then retrieves a transcript based on the first verifying proof it hears.

At a later stage, parties should be able to retrieve their secret keys in the Key Retrieve protocol. Recall that PVSS transcripts contain n encrypted Shamir shares of the secret key. During the Key Retrieve protocol, each party sends each other party an encryption of its Shamir share. Parties then reply with the decrypted shares, which can then be combined to compute the shared secret key.

Efficiency. The PVSSExchange protocol incurs $O(\lambda^2 n^2)$ communication costs. Each call to the Disperse protocol costs $O(\lambda n + m)$, where $m = O(\lambda n)$ is the size of the dispersed message. Standard retrieval constructions have parties send each other parts of the message, allowing everybody to retrieve the message. We privately retrieve a message instead, with each party only receiving one message. Each such private retrieval also costs $O(\lambda n + m)$. Other than that, parties send each other messages of size $O(\lambda)$. In total, the Key Escrow protocol costs $O(\lambda^2 n^2)$ and the Key Retrieve protocol costs $O(\lambda n^2)$. Both protocols have constant round complexity.

2.5 Our Techniques: Provable Rank Protocol

In this section, we describe how to replace the VRFs that were used in the intuitive description above with an interactive analogue called a Provable Rank protocol. This interactive protocol can be constructed using a plain PKI setup without additional setup assumptions. Similarly to VRFs, a Provable Rank protocol allows parties to output independent pseudorandom values, along with proofs that these values are correctly computed.

As a first attempt, we could recall that parties could call the Key Escrow protocol to generate a public key with an associated random secret key. Parties can then call Key Retrieve to get that secret key and use it as a random value. Unfortunately, there is no guarantee that different parties generate different keys in the Key Escrow protocol, meaning that while each secret key is indeed random, the keys might not be independent.

We start in the same manner as the Key Escrow protocol by allowing each party to get an aggregated PVSS transcript with an unknown secret key. Now, instead of retrieving the secret keys, we view each secret key as a threshold VRF key. Each party i then collaboratively evaluates the threshold VRF on its own index, with the help of the network. To do so, i sends every other party its encrypted Shamir share contained in the PVSS transcript. Parties then reply with partial evaluations of the threshold VRF, which i combines to

obtain a pseudorandom rank and proof. Note that each party evaluates its threshold VRF on its *own index*, meaning that even if different parties use the same transcript, the random values are still independent.

2.6 Putting the Pieces Together: Our ADKG

We obtain our ADKG protocol as follows. First, parties call the PVSSExchange protocol [ABLS25] in order to obtain a possible valid output from the ADKG protocol for parties to agree on. Recall that at least one honest party outputs an aggregated PVSS transcript from the protocol. That party provably Disperses it, and sends the proof to all parties. The parties then execute a VABA[6] protocol to agree on which dispersed aggregated transcript to retrieve. This is done using the consensus protocol of [ABLS25], replacing their recursive weak coin generation procedure with our weak leader election (equivalently, weak coin[7]) protocol. Their VABA protocol proceeds in consecutive views, which can be thought of as attempts to reach consensus.

In each view, parties call the weak coin protocol once, and incur $O(\lambda n^2)$ communication and $O(1)$ rounds. Since our weak coin protocol has constant probability of success, the VABA protocol requires a constant expected number of views until all parties terminate. Finally, parties can Retrieve the agreed upon transcript, which can be output from the ADKG protocol. The Retrieve protocol requires an additional $O(\lambda n^2)$ cost and $O(1)$ rounds. Consequently, the overall expected communication cost of our ADKG protocol is $O(\lambda^2 n^2)$, and it completes in expected $O(1)$ rounds.

2.7 Open Questions

All existing works achieving subcubic communication under strong adaptive security require secure erasures. A major open question is therefore whether this requirement is inherent, or whether the assumption can be removed while maintaining subcubic communication (including setup costs). As a first step, one could explore this direction using heavy cryptographic machinery such as indistinguishability Obfuscation ($i\mathcal{O}$) or witness encryption for all NP.

2.8 More Related Work

We discuss further related work on DKG protocols.

Synchronous DKG protocols. Since the seminal work of Pedersen [Ped91], distributed key generation (DKG) protocols have been extensively studied. Most classical constructions [CGJ+99, GJKR07, GJM+21, Ped91] are designed for the synchronous network model, in which message delays are bounded by a known parameter Δ and parties are assumed to have synchronized clocks. At a high level, existing DKG protocols typically follow a two-phase structure consisting of a sharing phase followed by a consensus phase. During the sharing phase, each party samples a random secret and distributes it to all parties using a verifiable secret sharing (VSS) scheme. In the consensus phase, parties agree on a subset of at least $f+1$ parties—ensuring that an adversary corrupting f parties does not learn all secrets contributing to the final key—whose secrets are combined, typically by summation, to form the distributed secret key.

A fundamental bottleneck in this paradigm is that the sharing phase requires n parallel instances of Byzantine broadcast. Well-known lower bounds for Byzantine broadcast imply an inherent $\Omega(n^2)$ communication cost per instance [ACD+23, DR85] against strongly adaptive adversaries, even assuming a threshold key setup. As a result, the sharing phase alone incurs $\Omega(n^3)$ communication. Although several recent works [SBKN24, GJM+21, BLL+24, FLT24, FLMT24] depart from this classical blueprint, the protocols of [SBKN24, GJM+21] still exhibit cubic communication complexity, while [FLMT24] relies on strong common coins, which in turn require either a trusted dealer or a prior DKG. The protocol of [FLT24] employs network partitioning—viewable as recursion of depth one—to achieve $O(\lambda n^{2.5})$ communication and $O(n)$ rounds, whereas [BLL+24] applies recursion of depth $\log n$, achieving $O(\lambda n^2 \log n)$ communication and $O(n)$

[6] Also called *multivalued validated Byzantine agreement (MVBA)* in the literature.

[7] Our weak leader election protocol also directly yields a weak coin by applying an additional hash to the elected leader's rank.

rounds. It therefore remains an open question whether one can achieve $O(n^2)$ communication and $O(1)$ round complexity for DKG in the synchronous model.

Asynchronous DKG protocols. Beginning with the work of Kokoris-Kogias et al. [KKMS20], asynchronous DKG protocols have received considerable attention in recent years [AJM+23, AJM+21, DXKKR23, GLL+21, KKMS20]. In the asynchronous network model, messages may experience arbitrary delays, with the only guarantee being that messages exchanged between honest parties are eventually delivered. In particular, honest parties' clocks and processing speeds need not be synchronized, and parties may initiate protocol execution at different times. A central challenge in this setting arises from the inherent difficulty of reliably distinguishing between slow but honest parties and silent Byzantine parties.

All existing asynchronous DKG protocols retain the same two-phase structure of sharing followed by consensus. With the exception of the recent subcubic constructions [FT24, ABLS25], the sharing phase is implemented via n parallel instances of an asynchronous VSS (AVSS) protocol, each initiated by a different party. However, AVSS protocols are known to incur an inherent $\Omega(n^2)$ communication cost per instance [BDK13], resulting in an overall $\Omega(n^3)$ communication cost for the sharing phase alone. The subsequent consensus phase is typically realized via multivalued validated Byzantine agreement (MVBA) [CKPS01], which generally relies on common coins implemented using a threshold-key setup.[8] Prior to the recent works [FT24, ABLS25], all known MVBA protocols that do not assume a threshold key setup incurred $\Omega(\lambda n^3)$ communication, as the cost of establishing a common coin was itself cubic. Several additional works [BLSW24, BHK+24, GS24] propose batched asynchronous DKG protocols that generate multiple secret keys simultaneously using linear randomness extractors; however, these constructions likewise retain cubic communication complexity. Finally, [BCLZL23] introduces a network-agnostic DKG protocol that provides security guarantees for both synchronous and asynchronous networks under different threshold assumptions; their construction also has cubic communication complexity.

3 Preliminaries and Model

In this section, we provide the preliminaries and model for our paper.

General notation. We let λ denote the security parameter. We assume that the public parameters of all underlying cryptographic primitives, which are implicitly parameterized by λ, are fixed in advance and known to all parties. For integers $a \leq b$, we define the interval $[a, b] := \{a, \ldots, b\}$, and when $a = 1$ we write this set simply as $[b]$. We use $\mathbb{N}$ to denote the set of positive integers and $\mathbb{N}_0$ to denote the set of non-negative integers. We use standard cryptographic notions such as negligible functions and probabilistic polynomial-time (PPT) algorithms, and assume that all stated protocol properties hold except with negligible probability in λ. Throughout, we measure the communication cost of our distributed protocols in bits.

Adversarial and network model. We consider a system of n parties $\{1, \ldots, n\}$ connected by pairwise authenticated and private channels in a complete network. Communication takes place in an *asynchronous* network, where messages may be delayed arbitrarily, subject only to the guarantee that any message sent between honest parties is eventually delivered. We assume a Byzantine adversary that can corrupt up to $f < n/3$ parties and cause them to deviate arbitrarily from the protocol. The adversary is *strongly adaptive*: it may corrupt parties at any time during the execution and can delete or substitute any undelivered message that a party sent while it was still honest.

Public key infrastructure. As is standard in the literature on distributed cryptographic protocols [DY81], we assume a bulletin-board–based *public key infrastructure* (PKI) established before protocol execution. Each party $i \in [n]$ generates a public–secret key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ for a digital signature scheme DS, an aggregatable PVSS scheme PVSS, a verifiable random function VRF, and a forward-secure digital signature scheme KDS (see Appendix A). For notational simplicity, we assume that the pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ comprises the keys for all of these primitives. In the bulletin-board PKI model (also called the *plain* PKI model), each party generates

[8] Such threshold-key setups typically rely on a trusted dealer to threshold-share a secret key, enabling non-interactive threshold signatures and strong common coins. In the absence of a trusted dealer, however, this requires running a DKG.

its keys locally—where corrupted parties may choose their keys arbitrarily—and publishes its public keys on a public bulletin board. Throughout, we use $\vec{\mathsf{pk}} := (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ to denote the vector of all public keys.

3.1 Cryptographic Primitives

We briefly review the cryptographic tools used in this paper. Formal definitions are given in Appendix A.

Verifiable random function. A verifiable random function (VRF) [MVR99] enables the generation of a pseudorandom string u together with a proof of correctness π. Concretely, on input a message m and a secret key sk, the VRF outputs a value $u = \mathsf{VRF}(\mathsf{sk}, m)$ along with a proof π attesting to the correctness of this computation. Given the proof π and the corresponding public key pk, any party can verify that u was computed correctly, while learning no information about the secret key sk. Looking ahead, we employ the committee-sampling procedure of [ABLS25], which makes use of VRFs. Notably, their protocol assumes only a plain PKI setup and does not rely on any trusted VRF setup; these can be instantiated using [DY05].

Forward-secure digital signatures. In a forward-secure digital signature scheme [BM99, KO17], a party holds a base public-secret key pair $(\mathsf{pk}, \mathsf{sk})$, where the public key is fixed and the secret key evolves sequentially over time. To this end, the scheme provides an update algorithm Update that, on input the current secret key sk and an epoch number $r \in \mathbb{N}$, outputs the updated secret key sk for the next epoch $r + 1$. Looking ahead, we employ the committee-sampling procedure of [ABLS25], which makes use of forward-secure digital signatures; these can be instantiated using [KO17].

Public key encryption. In a public-key encryption (PKE) scheme [RSA78], a party holds a public-secret key pair $(\mathsf{ek}, \mathsf{dk})$. The public key ek is made available to everyone, while the secret key dk is kept private by its owner. Any party can encrypt a message m under ek to produce a ciphertext $c = \mathsf{Encrypt}(\mathsf{ek}, m)$. Only the holder of the corresponding secret key dk can decrypt c to recover the message, by computing $m = \mathsf{Decrypt}(\mathsf{dk}, c)$. Chosen-plaintext (CPA) security requires that no PPT adversary can distinguish encryptions of two messages of its choice, i.e., ciphertexts leak no information about the plaintext beyond its length. Looking ahead, we will use an encryption scheme to allow parties to encrypt their individual raw ranks; this can be instantiated using [BF01].

Non-interactive zero-knowledge proofs. A non-interactive zero-knowledge (NIZK) proof [BFM88] for an NP relation $\mathcal{R}$ with respect to a random oracle H consists of a pair of algorithms $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVerify})$. On input a statement x and a witness w such that $(x, w) \in \mathcal{R}$, the prover PProve outputs a proof π, and the verifier PVer takes (x, π) as input and outputs an accept/reject decision. Completeness requires that honestly generated proofs are always accepted, soundness that no accepting proof can be produced for a false statement, knowledge soundness requires that any prover that produces an accepting proof must know a corresponding witness, and zero knowledge that proofs can be simulated without knowing the witness by programming the random oracle. For readability, we omit the explicit dependence on H. Our NIZKs can be instantiated using zkSNARKs with transparent setup [Fu25, AGL+23]. Our protocols admit alternative instantiations, as long as the proof size remains at most $O(\lambda^2)$.

4 Background: Protocols from Prior Work

In this section, we recall several protocols from prior work that will be relevant for our construction.

4.1 Non-Equivocation

The protocol is from [ABLS25]. $\mathsf{NonEquiv}$ is a protocol for non-equivocation. Each party $i \in [n]$ participates with an input m_i and a tag tag, which can be viewed as a round or instance identifier. Upon completion, a party may output a proof π_i attesting that m_i is the unique value that it submitted to the protocol under the given tag. Any party can verify such a proof using the deterministic verification algorithm $\mathsf{NEVerify}$. The $\mathsf{NonEquiv}$ protocol is closely related to the provable broadcast primitive of [AMS19]. We restate the formal definition from [ABLS25].

Definition 1. *A non-equivocation protocol* $(\mathsf{NonEquiv}, \mathsf{NEVerify})$ *has the following properties, assuming all honest parties call the protocol with the same tag:*

- ***Validity.*** *If an honest party i outputs π_i, then* $\mathsf{NEVerify}(i, m_i, \pi_i, tag) = 1$.
- ***Non-Equivocation.*** *The adversary cannot generate two tuples (j, m, π) and (j, m', π') with $m \neq m'$ and* $\mathsf{NEVerify}(j, m, \pi, tag) = \mathsf{NEVerify}(j, m', \pi', tag) = 1$.
- ***Liveness.*** *Every honest party eventually outputs a proof.*

We recall their construction in Appendix B.1. The protocol tolerates up to $f < n/3$ Byzantine faults and achieves $O((\lambda + m)n^2)$ communication complexity in $O(1)$ rounds, where m denotes the input size. For the proof of correctness and security, we refer the reader to Section 4.1 of the original paper.

4.2 PVSS Exchange

The protocol is from [ABLS25]. The goal of the *PVSS Exchange* protocol is to produce an aggregated PVSS transcript that can be certified to contain at least one honest contribution, except with negligible probability. Each party locally generates a PVSS transcript and sends it to a set of $\Theta(\lambda)$ parties selected via a VRF evaluation. A party outputs an aggregated transcript after receiving a sufficiently large number of transcripts—approximately two thirds of the expected number—so as to ensure that the aggregation includes at least one transcript generated by an honest party. Concretely, each party invokes the $\mathsf{PVSSExchange}$ protocol with no input and may eventually output an aggregated transcript together with a proof. Given such an output, any party can invoke the verification algorithm $\mathsf{ExchangeVerify}$ to check that the aggregated transcript contains at least one honestly generated contribution, except with negligible probability. Importantly, the $\mathsf{PVSSExchange}$ protocol does not provide agreement: different parties may output different aggregated PVSS transcripts. Instead, it only guarantees that any transcript accepted by the verification procedure contains at least one honest contribution, except with negligible probability, and that at least one forever-honest party terminates and produces such an output, also except with negligible probability. We restate the formal definition from [ABLS25].

Definition 2. *A PVSS exchange protocol* $(\mathsf{PVSSExchange}, \mathsf{ExchangeVerify})$ *has the following properties if all honest parties call the protocol:*

- ***Verifiability.*** *If an honest party outputs* $(\mathsf{trans}_i, \pi_i)$*, then* $\mathsf{ExchangeVerify}(\mathsf{trans}_i, \pi_i) = 1$*. Moreover, if the adversary provides a pair* (trans, π) *such that* $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi) = 1$*, then* trans *includes at least one contribution generated by an honest party that erased its contents without being corrupted.*
- ***Termination.*** *At least one forever-honest party completes the protocol and outputs a value.*

We recall their construction in Appendix B.2. The protocol tolerates up to $f < n/3$ Byzantine faults and achieves $O(\lambda^2 n^2)$ communication complexity in $O(1)$ rounds, where m denotes the input size. For the proof of correctness and security, we refer the reader to Section 5.1 of the original paper. We note that their protocol assumes only a plain PKI setup and does not require any trusted VRF setup.

4.3 Provable AVID

The protocol is from [ABLS25]. A *provable asynchronous verifiable information dispersal (AVID)* protocol consists of two subprotocols, $\mathsf{Disperse}$ and $\mathsf{Retrieve}$, together with a deterministic verification algorithm $\mathsf{DisperseVerify}$. Parties additionally have access to an external validity function valid. In the $\mathsf{Disperse}$ protocol, a designated sender is given as input a message m such that $\mathsf{valid}(m) = 1$. The sender may produce a *proof of dispersal* π, which can be verified using $\mathsf{DisperseVerify}(m, \pi)$. A valid proof π certifies that the message m has been correctly dispersed and that the corresponding $\mathsf{Retrieve}$ protocol will succeed. In the $\mathsf{Retrieve}$ protocol, all parties can jointly reconstruct the dispersed message. Correctness requires that every honest party outputs the same message m with $\mathsf{valid}(m) = 1$. We restate the formal definition from [ABLS25].

Definition 3. *In a provable AVID protocol* (Disperse, DisperseVerify, Retrieve)*, a designated sender s has an input m, and all parties have access to an external validity function* valid*. Assuming that* $\mathsf{valid}(m) = 1$*, the protocol has the following properties:*

- ***Binding.*** *If an honest party receives a proof π such that* $\mathsf{DisperseVerify}(s, \pi) = 1$*, then there is a unique value m' such that every honest party that outputs a value from* Retrieve *protocol outputs m'.*
- ***External Validity.*** *For the m' defined in the binding property,* $\mathsf{valid}(m') = 1$*. Further, if the sender is honest, then the committed value is its input $m' = m$.*
- ***Termination.*** *If s is honest, then it completes the* Disperse *protocol and outputs a proof π.*
- ***Provability.*** *If s is honest and it outputs a proof π, then* $\mathsf{DisperseVerify}(s, \pi) = 1$*.*
- ***Totality.*** *If an honest party receives a proof π such that* $\mathsf{DisperseVerify}(s, \pi) = 1$ *and all honest parties call* Retrieve*, then they all complete* Retrieve*.*

We recall their construction in Appendix B.3. The protocol tolerates up to $f < n/3$ Byzantine faults and runs in $O(1)$ rounds. The Disperse subprotocol incurs $O(\lambda n + m)$ communication, while the Retrieve subprotocol incurs $O(\lambda n^2 + mn)$ communication, where m denotes the input size. For the proof of correctness and security, we refer the reader to Section 4.2 of the original paper.

4.4 Aggregatable PVSS

We use the scheme from [GJM+21, BL23], which operates over bilinear groups.[9] In the following, we recall the PVSS scheme and omit details that are not essential for our exposition. We further introduce notation that will be useful for our protocol constructions in subsequent sections.

- **Setup.** Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear pairing over cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order p, with generators $g \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$. Each party $i \in [n]$ holds a key pair (pk_i, sk_i) such that $pk_i := h^{sk_i} \in \mathbb{G}_2$ and $sk_i \in \mathbb{Z}_p$ for $i \in [n]$.
- **Secret Sharing.** The algorithm Dist (run by a dealing party $\ell \in [n]$) takes as input a secret key sk_ℓ and public keys $pk_1, \ldots, pk_n$. It outputs a PVSS transcript $\mathsf{trans}_\ell := \{\mathbf{C}, \mathbf{E}, \pi\}$, where

$$\mathbf{C} := (g^{s(0)}, g^{s(1)}, \ldots, g^{s(n)}), \quad \mathbf{E} := (pk_1^{s(1)}, \ldots, pk_n^{s(n)}),$$

  and $s \in \mathbb{Z}_p[X]$ is a uniformly random polynomial of degree f. Here, π denotes a signed proof of knowledge of $s(0)$ from party ℓ. For any transcript trans, we define

$$\mathsf{trans}[0] := C_0, \quad \forall j \in [n],\ \mathsf{trans}[j] := (C_j, E_j).$$

  We refer to $\mathsf{trans}[0]$ as the *transcript verification key*, and to $\mathsf{trans}[j]$ as the *j-th transcript share*.
- **Aggregation.** The algorithm Agg takes as input $k \geq 1$ transcripts $\{(\mathbf{C}_\ell, \mathbf{E}_\ell, \pi_\ell)\}_{\ell \in [k]}$ and outputs an aggregated transcript $\mathsf{trans} := \{\mathbf{C}, \mathbf{E}, \pi\}$, where

$$\mathbf{C} := \prod_{\ell \in [k]} \mathbf{C}_\ell, \quad \mathbf{E} := \prod_{\ell \in [k]} \mathbf{E}_\ell, \quad \pi := (\pi_1, \ldots, \pi_\ell),$$

  where the products are taken component-wise. Note that this is possible, as the corresponding entries have the same base g and pk_i, respectively. In particular, we emphasize that for any $k \in O(n)$, the size of an aggregated transcript is asymptotically the same as a single transcript. Further, we note that the (collective) proof π is used solely to verify that an aggregated transcript contains sufficiently many contributions—specifically, k contributions—from distinct parties.

[9] We note that this PVSS is a very basic construction derived from Scrape [CD17], and has also been used in several other works that leverage aggregatable PVSS, especially in the literature on distributed randomness beacons and DKGs [BLL+24, FLT24, BSKN23].

- **Transcript Verification.** The algorithm Ver takes as input public keys $pk_1, \ldots, pk_n$ and a transcript $\mathsf{trans} := \{\mathbf{C}, \mathbf{E}, \pi\}$, and outputs 1 (valid) or 0 (invalid). To this end, it checks correctness of each E_i with respect to C_i: for each $i \in [n]$, it checks $e(g, E_i) = e(C, pk_i)$. It further verifies π and that $\mathbf{C}$ is a commitment to a degree-f polynomial.
- **Share Extraction.** The algorithm $\mathsf{ShareExt}$ takes as input a secret key sk_i and transcript share (C_i, E_i), and outputs $S_i := E_i^{1/sk_i}$.
- **Share Verification.** The algorithm $\mathsf{ShareVer}$ takes as input a secret share S_j and a commitment C_j, and it checks $e(C_j, h) = e(g, S_j)$.
- **Reconstruction.** The algorithm $\mathsf{Reconstruction}$ takes as input any set of $f + 1$ valid secret shares $\{S_i\}_{i \in \mathcal{I}}$, and outputs the secret by Lagrange interpolation in the exponent. Concretely, the final reconstructed secret is then $S := h^{f(0)} \in \mathbb{G}_2$

4.5 Coin Generation from APVSS

The works [GJM+21, BLL+24] demonstrate how an aggregated PVSS transcript can be used to construct a one-round distributed randomness beacon protocol. Typically, an aggregated PVSS transcript is used to obtain a single coin flip by directly reconstructing the encoded secret S. However, as observed in prior work, interpreting the commitments $C_1, \ldots, C_n$ as public key shares of a public key C_0, and the secret shares $S_1, \ldots, S_n$ as secret key shares of a secret key S, enables the construction of a non-interactive threshold signature scheme with a unique signature component, analogous to threshold BLS.[10] A random coin is then derived by applying a random oracle to this unique signature component. Unforgeability of the threshold signature ensures that the resulting coin is uniformly random and independent in the random oracle model. Moreover, as shown in [BLL+24], unforgeability holds for any aggregated transcript containing at least one honest contribution. Finally, uniqueness of the signature component guarantees uniqueness of the derived coin for each message (and public key).

We briefly recall their construction and introduce notation that will be useful in subsequent protocol constructions. Let $\mathsf{trans} := \{\mathbf{C}, \mathbf{E}, \pi\}$ denote a PVSS transcript (as in Section 4.4) that serves as a threshold key setup for the threshold signature scheme of [GJM+21, BLL+24].

- **Setup.** As in Section 4.4: A bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ with generators $g \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$. Further, a random oracle $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{G}_1$ is used, which maps messages to group elements.
- **Partial Signing.** The $\mathsf{PartialCoin}$ algorithm takes as input a secret key share S_i and a message m, and outputs a partial signature $\sigma_i := (\sigma_i', \pi_i)$, where π_i is a proof of correctness for $\sigma_i' := e(\mathsf{H}_1(m), S_i)$.[11]
- **Partial Verification.** The $\mathsf{VerifyPartialCoin}$ algorithm takes as input a public key share C_i and a partial signature $\sigma_i = (\sigma_i', \pi_i)$, and outputs 1 (valid) or 0 (invalid). This partial signature verification check is performed using pairing equations only.
- **Reconstruction.** The $\mathsf{ReconstructCoin}$ algorithm takes as input a set $\{(\sigma_j', \pi_j)\}_{j \in \mathcal{I}}$ of $f + 1$ partial signatures, and outputs a full signature $\sigma := (\sigma', \pi)$ where

$$\sigma' := \prod_{j \in \mathcal{I}} \sigma_j'^{\lambda_{j,\mathcal{I}}} = e(\mathsf{H}_1(m), S), \quad \pi := \prod_{j \in \mathcal{I}} \pi_j,$$

  and $\lambda_{j,\mathcal{I}}$ denotes the j-th Lagrange coefficient for the set $\mathcal{I}$. We note that the proofs π_j, consisting of group elements, are also aggregatable.
- **Signature Verification.** The $\mathsf{VerifyCoin}$ algorithm takes as input a public key C_0, a message m, and a signature $\sigma = (\sigma', \pi)$, and outputs 1 (valid) or 0 (invalid). This verification check is performed using pairing equations only.
- **Coin Generation.** The per-message unique coin is computed as $\mathsf{coin} := \mathsf{H}(\sigma')$, where σ' denotes the unique component of the signature $\sigma = (\sigma', \pi)$ and $\mathsf{H} : \mathbb{G}_T \to \{0,1\}^\lambda$ is an additional random oracle.

[10] That is, for each message m, there exists a unique signature σ that verifies under the public key.

[11] The proof consists of several group elements from $\mathbb{G}_1$ and $\mathbb{G}_2$; its specific structure is not relevant for our purposes.

To build intuition for the threshold signature, it is helpful to view the construction through the lens of the underlying signature scheme. The public key is given by $\mathsf{pk} := g^s \in \mathbb{G}_1$, while the corresponding secret key is $\mathsf{sk} := h^s \in \mathbb{G}_2$, for a secret exponent $s \in \mathbb{Z}_p$.[12] To sign a message m, the signer computes $\sigma' = e(\mathsf{H}_1(m), \mathsf{sk})$ along with a proof of correctness π, which together constitute the signature $\sigma := (\sigma', \pi)$.

Remark 1. *The authors of [BLL+24] prove that the threshold signature scheme satisfies strong unforgeability for any aggregated transcript that includes the contribution of at least one honest party. Strong unforgeability allows the adversary to obtain up to f partial signatures on a target forgery message m^* without rendering the forgery trivial. More generally, they show that the threshold key setup can be securely used to derive coins that are uniformly random and mutually independent in the random oracle model. In particular, for any PPT adversary, it is computationally hard to forge the unique signature component σ'. Henceforth, we use the term* unforgeability of the threshold signature *to denote the computational hardness of forging this unique signature component.*

4.6 Encryption

This is essentially the Boneh–Franklin IBE [BF01], with a single identity. The scheme operates over bilinear groups and consists of the following algorithms.

- **Setup.** The system parameters are $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, p, g, h, \mathsf{H})$, where $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ are cyclic groups of prime order p, $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear pairing, $g \in \mathbb{G}_1$, $h \in \mathbb{G}_2$ are generators of the respective groups, and $\mathsf{H} : \mathbb{G}_T \to \{0,1\}^l$ is a cryptographic hash function.
- **Key Generation.** Sample $s \leftarrow \mathbb{Z}_p$ uniformly at random. Set the encryption key $\mathsf{ek} := g^s \in \mathbb{G}_1$, and the decryption key $\mathsf{dk} := h^s \in \mathbb{G}_2$.
- **Encrypt.** Take as input an encryption key $\mathsf{ek} \in \mathbb{G}_1$ and a message $m \in \{0,1\}^l$. Sample $r \leftarrow \mathbb{Z}_p$, and compute $U = g^r \in \mathbb{G}_1$ and $K = e(\mathsf{ek}, h)^r \in \mathbb{G}_T$. Output the ciphertext $c := (U, V) := \big(U,\ m \oplus \mathsf{H}(K)\big)$.
- **Decrypt.** Take as input a decryption key $\mathsf{dk} \in \mathbb{G}_2$ and a ciphertext $c = (U, V)$. Output the plaintext $m := V \oplus \mathsf{H}\big(e(U, \mathsf{dk})\big)$.

This scheme is secure against chosen-plaintext attacks (cf. Definition 11) [BF01].

5 Provable Rank

5.1 Definition

We provide a definition of a Provable Rank protocol. Our definition is split into two parts: first, a description of the protocol's *setup and syntax*, and second, a statement of the *security properties* it is required to satisfy. Our security properties are only required to hold except with negligible probability in λ.

Definition 4 (Provable Rank)**.** *Let $\mathcal{D}$ be a finite rank domain (e.g., $\mathcal{D} = \{0,1\}^{2\lambda}$ for security parameter λ). A **Provable Rank protocol** is defined as a tuple* $(\mathsf{ProvableRank}, \mathsf{RankVerify})$*, where* ProvableRank *is an interactive protocol executed by n parties, and* RankVerify *is a deterministic, non-interactive verification algorithm. To run the protocol, all parties share a public identifier called a tag, which uniquely identifies an execution of the protocol. Each party $i \in [n]$ executes* $\mathsf{ProvableRank}(tag)$ *and may output a pair* (rank_i, π_i)*, where $\mathsf{rank}_i \in \mathcal{D}$ and π_i is a proof. The verification algorithm* RankVerify *takes a tuple* $(\mathsf{rank}, i, \pi, tag)$ *as input, and outputs* 1 *(valid) or* 0 *(invalid).*

Assuming all parties execute the protocols with the same tag, a Provable Rank protocol must satisfy the following security properties in the presence of an adversary corrupting at most f parties.

- ***Correctness.*** *For any tag and any honest party i, whenever it completes an execution of* $\mathsf{ProvableRank}($*tag*$)$ *and outputs a pair* (rank_i, π_i)*, then* $\mathsf{RankVerify}(\mathsf{rank}_i, i, \pi_i, tag) = 1$*.*

[12] The corresponding public–secret key space is then $\mathcal{K} := \{(g^s, h^s) \in \mathbb{G}_1 \times \mathbb{G}_2 \mid s \in \mathbb{Z}_p\}$.

- ***Validity.*** *Let* $(\mathsf{rank}_i, i, \pi_i, tag)$ *be a valid tuple, i.e.,* $\mathsf{RankVerify}(\mathsf{rank}_i, i, \pi_i, tag) = 1$*, then* rank_i *is uniformly distributed over* $\mathcal{D}$ *and independently of all valid ranks with different indices or tags. We also define this via a security game in Definition 20.*
- ***Uniqueness.*** *The adversary cannot generate two tuples* $(\mathsf{rank}, i, \pi, tag)$ *and* $(\mathsf{rank}', i, \pi', tag)$ *such that* $\mathsf{rank} \neq \mathsf{rank}'$ *and* $\mathsf{RankVerify}(\mathsf{rank}_i, i, \pi_i, tag) = \mathsf{RankVerify}(\mathsf{rank}'_i, i, \pi'_i, tag) = 1$*. We also define this via a security game in Definition 19.*
- ***Secrecy.*** *For any tag and any honest party* i*, the adversary cannot distinguish the distribution of* (rank_i, π_i) *output by* $\mathsf{ProvableRank}(tag)$ *from* $(\tilde{r}, \tilde{\pi})$*, where* $\tilde{r} \leftarrow \mathcal{D}$ *and* $\mathsf{RankVerify}(\tilde{r}, i, \tilde{\pi}, tag) = 1$*, even when given all valid pairs* (rank_j, π_j) *from executions under any other tags and honest parties, as well as the adversary's full view of the protocol execution under that tag.*[13] *We also define this via a security game in Definition 18.*
- ***Termination.*** *For any tag, if all honest parties participate in* $\mathsf{ProvableRank}(tag)$*, then each of them eventually completes the protocol and outputs some pair* (rank, π)*.*

Intuitively, a Provable Rank protocol is designed to mimic a fair random draw from the space $\mathcal{D}$ among n participants in a distributed setting. Correctness ensures that whenever an honest participant completes the protocol and produces a pair (rank, π) the verifier will accept it—honest outputs are always valid. Validity captures the fairness of the draw: if two distinct participants obtain valid pairs, their ranks behave like independent uniform samples from the randomness space, so no adversary can bias one rank or correlate it with another. Uniqueness prevents equivocation by asserting that for each participant there is at most one valid (rank, π) pair; no one can produce different "winning tickets" for the same party. Secrecy hides the outcome: even though the protocol runs interactively, an adversary who sees all messages and proofs cannot tell an honest participant's rank from a fresh random element of $\mathcal{D}$. Termination guarantees liveness: if all honest parties engage, the protocol eventually finishes and yields a rank–proof pair for each of them, so the collective random draw cannot be indefinitely blocked by malicious behaviour.

5.2 Construction

We present our Provable Rank protocol as pseudocode in Algorithms 1 and 2 and give a verbal description next.

5.2.1 Protocol Description

Conceptually, the construction proceeds in three phases. In the first phase, each party establishes a private source of hidden randomness by obtaining an aggregated PVSS transcript that is guaranteed to contain at least one honest contribution. In the second phase, each party locally commits to its chosen transcript using non-equivocation and distributes the corresponding transcript shares to all other parties, together with proofs of consistency. In the final phase, each party's committed transcript is interpreted as a threshold signing key, which is then used to generate a unique threshold signature tied to that party; hashing this signature yields the party's individual rank. Each phase produces publicly verifiable artifacts, ensuring that ranks can later be verified non-interactively.

Phase 1: Per-party randomness setup (lines 1-17). In the first phase, every party participates in the PVSS Exchange protocol, whose goal is to produce an aggregated PVSS transcript trans together with a proof π_{trans} certifying that the aggregation contains at least one honestly generated contribution. Since PVSS Exchange guarantees termination only for at least one forever-honest party, the protocol is combined with provable AVID. Specifically, whenever a party completes PVSS Exchange, it disperses its resulting transcript and proof $(\mathsf{trans}, \pi_{\mathsf{trans}})$ and outputs a proof of dispersal π_{disp}, which certifies that the message $(\mathsf{trans}, \pi_{\mathsf{trans}})$ has been correctly dispersed and can later be retrieved by any party. The party then multicasts the dispersal proof π_{disp}, and upon receiving a valid dispersal proof, any party may invoke the private retrieval procedure

[13] That is, the adversary observes all rank–proof pairs and party i's proof π_i, but not the corresponding rank rank_i.

to locally reconstruct the dispersed message $(\mathsf{trans}, \pi_{\mathsf{trans}})$. As a result, each honest party i eventually obtains an aggregated PVSS transcript trans_i together with a corresponding proof $\pi_{\mathsf{trans},i}$.

Phase 2: Transcript commitment and sharing (lines 18-26). In the second phase, each party i commits to its chosen transcript trans_i and enforces consistency of all subsequent messages. To this end, party i invokes the non-equivocation protocol on the verification key $\mathsf{eph_vk}_i$ derived from trans_i and the protocol tag, producing a proof that it cannot later equivocate on this choice. This commitment ensures that party i is bound to a single transcript for the remainder of the execution. It is important to bind parties to a single transcript, as otherwise corrupt parties might ask to derive their ranks from several transcripts and choose the best rank among them. Party i then distributes its transcript to the rest of the system by sending each party j the designated transcript share $\mathsf{share}_j = \mathsf{trans}_i[j]$. Along with each share, i provides non-interactive proofs showing that the verification key is derived from a transcript that passed PVSS Exchange verification, and that the transmitted share is consistent with that same transcript. These proofs allow recipients to locally verify that all shares they receive from i are well-formed and tied to a single committed transcript, without any further interaction. At the end of this phase, every honest party holds a verifiable share of i's transcript.

Phase 3: Per-party coin generation (lines 27-38). In the final phase, party i's committed transcript trans_i is used to derive a random value uniquely associated with i. The transcript is interpreted as a threshold signing key: the commitments define public key shares, and each party j can locally extract its corresponding secret signing share S_j using its transcript share share_j and private key sk_j from the plain PKI setup. Each party j then uses its secret share to produce a partial threshold signature on the message (i, tag) and sends it to party i, provided all previously received proofs from i verify correctly. Once party i collects $f+1$ valid partial signatures $\{\sigma_j\}_j$, it reconstructs a full, unique threshold signature σ, hashes it, and outputs the resulting value as its rank rank_i. A final non-interactive proof $\pi_{\mathsf{rank},i}$ certifies that rank_i is the hash of a valid threshold signature under $\mathsf{eph_vk}_i$ on the message (i, tag). By the unforgeability of the threshold signature scheme, the ranks of honest parties are uniformly distributed and independent of the adversary's view.

5.3 Underlying Proof Systems

We make use of the following non-interactive proof system, which shows that $\mathsf{eph_vk}$ is the verification key from an transcript trans for which π_{trans} verifies. Concretely, we define the proof system $\mathsf{PS}_{\mathsf{key}} = (\mathsf{PProve}_{\mathsf{key}}, \mathsf{PVerify}_{\mathsf{key}})$ for the relation

$$\mathcal{R}_{\mathsf{key}} := \{((\mathsf{eph_vk}, tag, \pi_{\mathsf{trans}}), \mathsf{trans}) \mid \mathsf{eph_vk} = \mathsf{trans}[0], \mathsf{ExchangeVerify}(\mathsf{trans}, tag, \pi_{\mathsf{trans}}) = 1\}.$$

We further need a proof system showing that share_j is the j-th transcript share of an trans whose verification key is $\mathsf{eph_vk}$ and for which π_{key} verifies. We define the proof system $\mathsf{PS}_{\mathsf{share}} = (\mathsf{PProve}_{\mathsf{share}}, \mathsf{PVerify}_{\mathsf{share}})$ for the relation

$$\mathcal{R}_{\mathsf{share}} := \{((\mathsf{eph_vk}, \pi_{\mathsf{key}}, \mathsf{share}_j, j, tag), \mathsf{trans}) \mid \mathsf{share}_j = \mathsf{trans}[j], \mathsf{PVerify}_{\mathsf{key}}(\mathsf{trans}, tag, \pi_{\mathsf{key}}) = 1\}.$$

We further need a proof system showing that rank is the hash of a threshold signature[14] component σ' on message (i, tag) that verifies with respect to key $\mathsf{eph_vk}$. We define the proof system $\mathsf{PS}_{\mathsf{coin}} = (\mathsf{PProve}_{\mathsf{coin}}, \mathsf{PVerify}_{\mathsf{coin}})$ for the relation

$$\mathcal{R}_{\mathsf{coin}} := \{((\mathsf{eph_vk}, \mathsf{rank}, i, tag), (\sigma', \pi)) \mid \mathsf{rank} = \mathsf{Hash}(\sigma'), \mathsf{VerifyCoin}(\mathsf{eph_vk}, (i, tag), (\sigma', \pi)) = 1\}.$$

Algorithm 1 ProvableRank(tag)

$\mathsf{trans}_i \leftarrow \perp,\ \pi_{\mathsf{NE},i} \leftarrow \perp,\ \pi_{\mathsf{trans},i} \leftarrow \perp,\ \pi_{\mathsf{key},i} \leftarrow \perp,\ \mathsf{eph_vk}_i \leftarrow \perp,\ \mathsf{partials}_i \leftarrow \emptyset$
// Get a PVSS transcript, either from PVSSExchange*, or by retrieving somebody else's*
call PVSSExchange(tag)
upon PVSSExchange(tag) terminating with output (trans, π), **do**
  if $\mathsf{trans}_i = \perp$ **then**
    $\mathsf{trans}_i \leftarrow \mathsf{trans},\ \pi_{\mathsf{trans},i} \leftarrow \pi$
    call Disperse(trans, π) with external validity ExchangeVerify($\cdot, \cdot, tag$)
upon Disperse terminating with output π_{disp}, **do**
  send $\langle$"dispersed", $\pi_{\mathsf{disp}}\rangle$ to all parties
upon receiving the first $\langle$"dispersed", $\pi_{\mathsf{disp},j}\rangle$ message from j, **do**
  if i did not send a "retrieve" message and DisperseVerify($j, \pi_{\mathsf{disp},j}$) $= 1$ **then**
    send $\langle$"retrieve", $j\rangle$ to all parties
    participate in PrivateRetrieve(i) for retrieving j's dispersal
upon receiving the first $\langle$"retrieve", $k\rangle$ message from j, **do**
  participate in PrivateRetrieve(j) for letting j retrieve k's dispersal
upon PrivateRetrieve(i) outputting (trans, π), **do**
  if $\mathsf{trans}_i = \perp$ **then**
    $\mathsf{trans}_i \leftarrow \mathsf{trans},\ \pi_{\mathsf{trans},i} \leftarrow \pi$
// Once a transcript is received, commit to a single public key
upon $\mathsf{trans}_i \neq \perp$, **do**
  $\mathsf{eph_vk}_i \leftarrow \mathsf{trans}_i[0]$
  call NE($\mathsf{eph_vk}_i$, ("flip", tag))
// Ask others for help in computing a rank for my key
upon NE outputting π, **do**
  $\pi_{\mathsf{NE},i} \leftarrow \pi$
  $\pi_{\mathsf{key},i} \leftarrow \mathsf{PProve}_{\mathsf{key}}((\mathsf{eph_vk}_i, \pi_{\mathsf{trans},i}, tag), \mathsf{trans}_i)$
  for all $j \in [n]$ **do**
    $\mathsf{share}_j \leftarrow \mathsf{trans}_i[j]$
    $\pi_{\mathsf{share},j} \leftarrow \mathsf{PProve}_{\mathsf{share}}((\mathsf{eph_vk}_i, \pi_{\mathsf{key},i}, \mathsf{share}_j, j, tag), \mathsf{trans}_i)$
    send $\langle$"flip", $\mathsf{eph_vk}_i, \pi_{\mathsf{NE},i}, \pi_{\mathsf{trans},i}, \pi_{\mathsf{key},i}, \mathsf{share}_j, \pi_{\mathsf{share},j}\rangle$ to j
// Help others by sending them partial coins derived from their transcripts
upon receiving $\langle$"flip", $\mathsf{eph_vk}_j, \pi_{\mathsf{NE},j}, \pi_{\mathsf{trans},j}, \pi_{\mathsf{key},j}, \mathsf{share}_i, \pi_{\mathsf{share},i}\rangle$ from j, **do**
  if NEVerify($j, \mathsf{eph_vk}_j$, ("flip", tag), $\pi_{\mathsf{NE},j}$) $= 1$ **then**
    if $\mathsf{PVerify}_{\mathsf{key}}((\mathsf{eph_vk}_j, \pi_{\mathsf{trans},j}, tag), \pi_{\mathsf{key},j}) = \mathsf{PVerify}_{\mathsf{share}}((\mathsf{eph_vk}_j, \pi_{\mathsf{key},j}, \mathsf{share}_i, i, tag), \pi_{\mathsf{share},i}) = 1$ **then**
      $S_i \leftarrow \mathsf{ShareExt}(\mathsf{sk}_i, \mathsf{share}_i)$
      send $\langle$"flip share", PartialCoin($S_i, (j, tag)$)$\rangle$ to j
// Use other parties' partial coins to compute my rank
upon receiving $\langle$"flip share", $\sigma_j\rangle$ from j, **do**
  if VerifyPartialCoin($\mathsf{trans}_i[j], (i, tag), \sigma_j$) $= 1$ **then**
    $\mathsf{partials}_i \leftarrow \mathsf{partials}_i \cup \{(j, \sigma_j)\}$
    if $|\mathsf{partials}| = f + 1$ **then**
      $\sigma = (\sigma', \pi) \leftarrow \mathsf{ReconstructCoin}(\mathsf{partials}_i),\ \mathsf{rank}_i \leftarrow \mathsf{Hash}(\sigma')$
      $\pi_{\mathsf{coin},i} \leftarrow \mathsf{PProve}_{\mathsf{coin}}((\mathsf{eph_vk}_i, \mathsf{rank}_i, i, tag), \sigma)$
      output rank_i and $\pi_{\mathsf{rank},i} = (\mathsf{eph_vk}_i, \pi_{\mathsf{NE},i}, \pi_{\mathsf{coin},i}, \pi_{\mathsf{trans},i}, \pi_{\mathsf{key},i})$

Algorithm 2 $\mathsf{RankVerify}(\mathsf{rank}_i, i, (\mathsf{eph_vk}_i, \pi_{\mathsf{coin},i}, \pi_{\mathsf{trans},i}, \pi_{\mathsf{key},i}), tag)$

1: **output** $\mathsf{NEVerify}(i, \mathsf{eph_vk}_i, (\text{“flip”}, tag), \pi_{\mathsf{NE},i}) \wedge \mathsf{PVerify}_{\mathsf{key}}((\mathsf{eph_vk}_i, \pi_{\mathsf{trans},i}, tag), \pi_{\mathsf{key},i}) \wedge \mathsf{PVerify}_{\mathsf{coin}}((\mathsf{eph_vk}_i, \mathsf{rank}_i, i, tag), \pi_{\mathsf{coin},i})$

5.4 Construction

5.5 Analysis

Due to space constraints, we defer the proofs to Section C.1.

Theorem 1. *The pair* (ProvableRank, RankVerify) *(cf. Algorithms 1 and 2) is a Provable Rank protocol resilient to $f < n/3$ Byzantine parties, except with negligible probability in λ. Moreover, the* ProvableRank *protocol runs in $O(\lambda^2 n^2)$ communication and $O(1)$ rounds.*

6 Key Escrow

6.1 Definition

We provide a definition of a Key Escrow protocol. Our definition is split into two parts: first, a description of the protocol's *setup and syntax*, and second, a statement of the *security properties* it is required to satisfy. Our security properties are only required to hold except with negligible probability in λ.

For our definition, we assume a key space consisting of all valid public-secret key pairs output by the key generation algorithm of the underlying cryptographic scheme or application of interest (e.g., a signature or encryption scheme). We further assume a uniqueness property: for each public key in this space, there exists at most one corresponding secret key. For example, in a cyclic group $\mathbb{G}$ of prime order p with generator g, a natural key set is $\{(g^s, s) \in \mathbb{G} \times \mathbb{Z}_p \mid s \in \mathbb{Z}_p\}$, or $\mathcal{K} := \{(g^s, h^s) \in \mathbb{G}_1 \times \mathbb{G}_2 \mid s \in \mathbb{Z}_p\}$ in a bilinear pairing group with generators $g \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$.

Definition 5 (Key Escrow)**.** *Let $\mathcal{K}$ be a finite key space as specified above. A* ***Key Escrow protocol*** *is defined as a tuple* (KeyEscrow, KeyVerify, KeyRetrieve)*, where* KeyEscrow *and* KeyRetrieve *are interactive protocols executed by n parties, and* KeyVerify *is a deterministic, non-interactive verification algorithm. To run the protocol, all parties share a public identifier called a tag, which uniquely identifies an execution of the protocol. Each party $i \in [n]$ executes* $\mathsf{KeyEscrow}(tag)$ *and may output a pair* $(\mathsf{eph_pk}_i, \pi_i)$*, where* $\mathsf{eph_pk}_i$ *is a public key and π_i is a proof. The verification algorithm* KeyVerify *takes a tuple* $(\mathsf{eph_pk}, \pi, tag)$ *as input, and outputs 1 (valid) or 0 (invalid). Each party i may subsequently execute* $\mathsf{KeyRetrieve}(tag)$ *and output a value* $\mathsf{eph_sk}_i$*, intended to be the secret key corresponding to* $\mathsf{eph_pk}_i$*.*

Assuming all parties execute the protocols with the same tag, a Key Escrow protocol must satisfy the following security properties in the presence of an adversary corrupting at most f parties.

- ***Correctness.*** *For any tag and any honest party i, whenever it completes an execution of* $\mathsf{KeyEscrow}(tag)$ *and outputs* $(\mathsf{eph_pk}_i, \pi_i)$*, then there exists a corresponding secret key* $\mathsf{eph_sk}_i$ *such that* $(\mathsf{eph_pk}_i, \mathsf{eph_sk}_i) \in \mathcal{K}$*. Moreover, whenever i completes an execution of* $\mathsf{KeyRetrieve}(tag)$*, it outputs this secret key* $\mathsf{eph_sk}_i$*.*
- ***Provability.*** *For any tag and any honest party i, whenever it completes an execution of* $\mathsf{KeyEscrow}(tag)$ *and outputs* $(\mathsf{eph_pk}_i, \pi_i)$*, then* $\mathsf{KeyVerify}(\mathsf{eph_pk}_i, \pi_i, tag) = 1$*.*
- ***Validity.*** *For any tuple* $(\mathsf{eph_pk}, \pi, tag)$ *such that* $\mathsf{KeyVerify}(\mathsf{eph_pk}, \pi, tag) = 1$*, there exists a (unique) secret key* $\mathsf{eph_sk}$ *such that* $(\mathsf{eph_pk}, \mathsf{eph_sk}) \in \mathcal{K}$*. Moreover, prior to any execution of* $\mathsf{KeyRetrieve}(tag)$*,* $\mathsf{eph_sk}$ *remains unpredictable to the adversary, even given its joint view of all protocol executions across all tags (including executions of* KeyRetrieve *under any other tags). We define unpredictability via the security game in Definition 6 below.*

[14] The threshold signature is $\sigma = (\sigma', \pi)$, where π is a pairing-based proof of correctness for σ', see Sections 4.4 and 4.5.

- ***Termination.*** *For any tag, if all honest parties participate in* $\mathsf{KeyEscrow}(tag)$*, then each of them eventually outputs some pair* $(\mathsf{eph_pk}_i, \pi_i)$*. Moreover, if all honest parties subsequently participate in an execution of* $\mathsf{KeyRetrieve}(tag)$*, then each of them eventually outputs a value.*

Definition 6 (Unpredictability)**.** *For every efficient adversary* $\mathcal{A}$ *that corrupts at most* f *parties, the probability that* $\mathcal{A}$ *wins the following security experiment is negligible in* λ*.*

- ***Setup phase.*** *The challenger gives* $\mathcal{A}$ *all public parameters of the scheme and initializes two empty sets* $\mathcal{Q}_1, \mathcal{Q}_2 \leftarrow \emptyset$*.*
- ***Oracle*** $\textsc{KeyEscrow}(tag)$*. On input a tag, the challenger checks whether* $tag \in \mathcal{Q}_1$*. If so, it returns* $\perp$*. Otherwise, it initiates an execution of the* $\mathsf{KeyEscrow}(tag)$ *protocol among all parties, faithfully emulating the behavior of all honest parties, and records the tag by updating* $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 \cup \{tag\}$*.*
- ***Oracle*** $\textsc{KeyRetrieve}(tag)$*. On input a tag, the challenger checks whether* $tag \in \mathcal{Q}_1$*. If not, it returns* $\perp$*. Otherwise, it initiates an execution of the* $\mathsf{KeyRetrieve}(tag)$ *protocol among all parties, faithfully emulating the behavior of all honest parties, and records the tag by updating* $\mathcal{Q}_2 \leftarrow \mathcal{Q}_2 \cup \{tag\}$*.*
- ***Winning condition.*** *The adversary outputs a tuple* $(\mathsf{eph_pk}^*, \pi^*, tag^*)$ *and a claimed secret key* $\mathsf{eph_sk}^*$*. The adversary wins if and only if*

$$\mathsf{KeyVerify}(\mathsf{eph_pk}^*, \pi^*, tag^*) = 1, \quad (\mathsf{eph_pk}^*, \mathsf{eph_sk}^*) \in \mathcal{K}, \quad tag^* \notin \mathcal{Q}_2.^{15}$$

Intuitively, a Key Escrow protocol enables a group of n participants to jointly generate public keys in a distributed manner, together with verifiable certificates, while ensuring that the corresponding secret keys can later be securely recovered. Correctness ensures that whenever an honest participant completes the escrow phase and produces a pair $(\mathsf{eph_pk}, \pi)$, the public key is *valid* in the sense that there exists a corresponding secret key in the underlying key space, and that this secret key is correctly recovered when the retrieval protocol is executed. Provability guarantees that any public key produced by an honest participant is accompanied by a proof that can be efficiently verified, so malformed or invalid keys are rejected. Validity captures soundness and secrecy: any public key that verifies must be valid (i.e., have a corresponding secret key in the key space), and prior to the execution of the retrieval protocol this secret key remains unpredictable to the adversary, even after observing the entire escrow execution. Finally, termination guarantees liveness: if all honest parties participate, the escrow protocol eventually produces a public-key–proof pair for each of them, and if they later invoke the retrieval protocol, the corresponding secret keys are eventually recovered, ensuring that key generation and recovery cannot be indefinitely blocked by malicious behavior.

6.2 Construction

We present our Key Escrow protocol as pseudocode in Algorithms 3 to 5 and give a verbal description next.

6.2.1 Protocol Description

The Key Escrow protocol consists of two interactive protocols and a non-interactive verification algorithm. We will describe the interactive protocols, KeyEscrow and KeyRetrieve. In KeyEscrow, a party establishes an escrowed public key together with a publicly verifiable proof certifying that the key was derived from an aggregated PVSS transcript containing at least one honest contribution. In KeyRetrieve, the parties collaboratively reconstruct the corresponding escrowed secret key by exchanging and validating transcript shares. Both protocols are accompanied by non-interactive proofs, ensuring that correctness can be verified without further interaction.

Protocol 1: KeyEscrow. This phase proceeds exactly as Phase 1 of the ProvableRank protocol (cf. Section 5.2), with the following additional step. Recall that after this phase, each party i holds a tuple $(\mathsf{trans}_i, \pi_{\mathsf{trans},i})$, where trans_i is an aggregated PVSS transcript and π_{trans} is a proof certifying that trans_i contains at

[15] We assume that it is possible to efficiently check whether a key pair lies in the key space (i.e., whether the key pair is valid). This is typically the case anyway (e.g., for the key spaces suggested above).

least one honest contributor. Party i then derives its escrowed public key $\mathsf{eph_pk}_i$ from trans_i. It produces a non-interactive proof $\pi_{\mathsf{key},i}$ certifying that $\mathsf{eph_pk}_i$ is derived from a transcript that passed the PVSS Exchange verification. Party i outputs the escrowed public key $\mathsf{eph_pk}_i$ together with the proof $\pi_{\mathsf{eph_pk},i} = (\pi_{\mathsf{trans},i}, \pi_{\mathsf{key},i})$, allowing any party to verify the correctness of the escrowed key.

Protocol 2: KeyRetrieve. The protocol enables reconstruction of the (unique) secret key corresponding to an escrowed public key produced in the Key Escrow phase. Holding an escrowed public key $\mathsf{eph_pk}_i$ and its associated transcript trans_i, party i distributes the transcript by sending each party j its designated transcript share $\mathsf{share}_j = \mathsf{trans}_i[j]$, together with a non-interactive proof $\pi_{\mathsf{share},i,j}$ certifying that the share is consistent with $\mathsf{eph_pk}_i$ and the verified transcript. These proofs allow recipients to locally verify that the received shares are well formed and correspond to the correct escrow instance. Upon receiving a valid transcript share, each party j locally extracts its corresponding secret share S_j using its private key sk_j from the plain PKI setup and returns the extracted value to party i. Party i verifies each returned value against the corresponding commitment in the transcript and collects all valid responses. Once i obtains $f+1$ valid secret shares, it reconstructs the escrowed secret key $\mathsf{eph_sk}_i$ using the reconstruction algorithm of the underlying PVSS scheme and outputs the result.

6.3 Construction

Algorithm 3 KeyEscrow(tag)

$\mathsf{trans}_i \leftarrow \bot,\ \pi_{\mathsf{trans},i} \leftarrow \bot,\ \mathsf{eph_pk}_i \leftarrow \bot,\ \pi_{\mathsf{key},i} \leftarrow \bot$
// Get a PVSS transcript, either from PVSSExchange*, or by retrieving somebody else's*
call PVSSExchange(tag)
upon PVSSExchange(tag) terminating with output (trans, π), **do**
  if $\mathsf{trans}_i = \bot$ **then**
    $\mathsf{trans}_i \leftarrow \mathsf{trans},\ \pi_{\mathsf{trans},i} \leftarrow \pi$
    call Disperse(trans, π) with external validity ExchangeVerify($\cdot, \cdot, tag$)
upon Disperse terminating with output π_{disp}, **do**
  send $\langle$"dispersed", $\pi_{\mathsf{disp}}\rangle$ to all parties
upon receiving the first $\langle$"dispersed", $\pi_{\mathsf{disp},j}\rangle$ message from j, **do**
  if i did not send a "retrieve" message and DisperseVerify($j, \pi_{\mathsf{disp},j}$) $= 1$ **then**
    send $\langle$"retrieve", $j\rangle$ to all parties
    participate in PrivateRetrieve(i) for retrieving j's dispersal
upon receiving the first $\langle$"retrieve", $k\rangle$ message from j, **do**
  participate in PrivateRetrieve(j) for letting j retrieve k's dispersal
upon PrivateRetrieve(i) outputting (trans, π), **do**
  if $\mathsf{trans}_i = \bot$ **then**
    $\mathsf{trans}_i \leftarrow \mathsf{trans},\ \pi_{\mathsf{trans},i} \leftarrow \pi$
// After getting a transcript, output its public key
upon $\mathsf{trans}_i \neq \bot$, **do**
  $\mathsf{eph_pk}_i \leftarrow \mathsf{trans}_i[0]$
  $\pi_{\mathsf{key},i} \leftarrow \mathsf{PProve}_{\mathsf{key}}((\mathsf{eph_pk}_i, \pi_{\mathsf{trans},i}, tag), \mathsf{trans}_i)$
  output $\mathsf{eph_pk}_i$ and $\pi_{\mathsf{eph_pk},i} = (\pi_{\mathsf{trans},i}, \pi_{\mathsf{key},i})$

6.4 Analysis

Due to space constraints, we defer the proofs to Section C.2.

Theorem 2. *The tuple* (KeyEscrow, KeyVerify, KeyRetrieve) *(cf. Algorithms 3 to 5) is a Key Escrow protocol*

Algorithm 4 KeyRetrieve(tag)

$\mathsf{shares}_i \leftarrow \emptyset$
// Ask others for help in reconstructing my secret key
for all $j \in [n]$ **do**
  $\mathsf{share}_j \leftarrow \mathsf{trans}_i[j]$
  $\pi_{\mathsf{share},j} \leftarrow \mathsf{PProve}_{\mathsf{share}}((\mathsf{eph_pk}_i, \pi_{\mathsf{key},i}, \mathsf{share}_j, j, tag), \mathsf{trans}_i)$
  send $\langle$"share", $\mathsf{share}_j, \pi_{\mathsf{share},j}, \mathsf{eph_pk}_i, \pi_{\mathsf{key},i}\rangle$ to j
// Help others by sending them shares of their keys
upon receiving the first $\langle$"share", $\mathsf{share}_i, \pi_{\mathsf{share},i}, \mathsf{eph_pk}_j, \pi_{\mathsf{key},j}\rangle$ message from j, **do**
  if $\mathsf{PVerify}_{\mathsf{share}}((\mathsf{eph_pk}_j, \pi_{\mathsf{key},j}, \mathsf{share}_i, i, tag), \pi_{\mathsf{share},i})$ **then**
    send $\langle$"dec", $\mathsf{ExtShare}(\mathsf{sk}_i, \mathsf{share})\rangle$ to i
// After receiving enough shares, output the reconstructed secret key
upon receiving the first $\langle$"dec", $S_j\rangle$ from j, **do**
  if $\mathsf{ShareVer}(S_j, \mathsf{share}_j) = 1$ **then**
    $\mathsf{shares}_i \leftarrow \mathsf{shares}_i \cup \{(j, S_j)\}$
    if $|\mathsf{shares}_i| = f + 1$ **then**
      $\mathsf{eph_sk}_i \leftarrow \mathsf{Reconstruct}(\mathsf{shares}_i)$
      output $\mathsf{eph_sk}_i$

Algorithm 5 KeyVerify($\mathsf{eph_pk}, (\pi_{\mathsf{trans}}, \pi_{\mathsf{key}}), tag$)

output $\mathsf{PVerify}_{\mathsf{key}}((\mathsf{eph_vk}, \pi_{\mathsf{trans}}, tag), \pi_{\mathsf{key}})$

resilient to $f < n/3$ Byzantine parties, except with negligible probability in λ. Moreover, the KeyEscrow *and* KeyRetrieve *protocols run in $O(\lambda^2 n^2)$ and $O(\lambda n^2)$ communication, respectively, and $O(1)$ rounds.*

7 Weak Leader Election

In this section, we construct a Weak Leader Election protocol. The protocol allows all parties to attempt to elect the same party that was honest when it *started* the protocol (the adversary can always corrupt a party after it is elected). The protocol is *weak* in the sense that there is some constant probability that all honest parties output the same honest index. With the remaining probability, parties might output the index of a party that was corrupted before starting the protocol, or even output different indices.

This primitive might seem too weak, allowing the adversary to corrupt the party after it is elected and thus making it an ineffective leader. However, this primitive allows parties to first act as leaders, and then retroactively elect one party after it has already played its role. In addition, the act of electing a single honest party is at the core of other important primitives for generating shared randomness.

Using one additional round of local computation, our weak leader election protocol yields a weak coin (see Remark 2) with the same complexities. Using a weak coin, [FT24, FGL+25] show how to construct a consensus protocol, a strong common coin, and an ADKG protocol. The ADKG protocol requires an expected constant number of calls to the weak coin protocol and an additional expected $O(\lambda^2 n^2)$ communication. In addition, their protocol has $O(1)$ expected round complexity. Using a weak coin, [ABLS25] also construct a consensus protocol and use it to construct an ADKG protocol with the same overhead as [FT24, FGL+25]. Combining our protocols with previous ADKG constructions, we achieve constant-round ADKG protocols with total costs $O(\lambda^2 n^2)$.

7.1 Definition

We provide a definition of a Weak Leader Election protocol. Our definition is split into two parts: first, a description of the protocol's *syntax*, and second, a statement of the *security properties* it is required to satisfy. Our security properties are only required to hold except with negligible probability in λ.

Definition 7 (Weak Leader Election)**.** *A* ***Weak Leader Election protocol*** *is an interactive protocol* $\mathsf{Election}$ *executed by n parties. To run the protocol, all parties share a public identifier called a tag, which uniquely identifies an execution of the election. Each party $i \in [n]$ executes* $\mathsf{Election}(tag)$ *and may output an index $\ell_i \in [n]$.*

Assuming all parties execute the protocol with the same tag, a Weak Leader Election protocol must satisfy the following properties in the presence of an adversary corrupting at most f parties:

- α-***Correctness.*** *With probability at least α, all honest parties that output an index output the same index $\ell \in [n]$, and the party indexed by ℓ was honest at the beginning of the protocol execution.*
- ***Termination.*** *If all honest parties participate in* $\mathsf{Election}(tag)$*, then each of them eventually completes the protocol and outputs an index.*

Intuitively, a Weak Leader Election protocol is designed to select a single honest leader among n participants in a distributed setting. α-correctness guarantees that, with probability at least α, all honest participants agree on the same leader and that the selected leader was honest at the start of the protocol. Termination ensures liveness: if all honest parties participate, the protocol eventually completes and each honest party outputs a leader, so the election cannot be indefinitely stalled by malicious behavior.

7.2 Construction

We present our Weak Leader Election protocol as pseudocode in Algorithm 6 and give a verbal description next.

7.2.1 Protocol Description

Conceptually, the Weak Leader Election protocol proceeds in four phases. In the first phase, each party locally generates a provably random rank and an escrowed encryption key. In the second phase, parties privately disseminate their ranks by encrypting them under each recipient's escrowed encryption key. In the third phase, parties coordinate to ensure that sufficiently many encrypted ranks have been reliably stored before allowing any decryption to occur. In the final phase, parties decrypt, locally determine the maximal rank, and converge on a common elected leader through a final voting step. Each phase produces publicly verifiable artifacts, ensuring that the elected leader and its rank can be validated non-interactively.

Phase 1: Rank generation and key escrow (lines 1-6). Each party i invokes $\mathsf{ProvableRank}$ to obtain a random value r_i together with a proof $\pi_{\mathsf{rank},i}$ certifying that the rank was correctly generated. The party also invokes $\mathsf{KeyEscrow}$ to generate an ephemeral public key $\mathsf{eph_pk}_i$ and a corresponding proof $\pi_{\mathsf{eph_pk},i}$ certifying that the key is well formed and that the associated secret key will remain hidden until $\mathsf{KeyRetrieve}$ is called. Upon completion of $\mathsf{KeyEscrow}$, party i send its public key and proof $(\mathsf{eph_pk}_i, \pi_{\mathsf{eph_pk},i})$ to all parties. Recipients verify the proof before accepting the key.

Phase 2: Encrypted rank dissemination (lines 7-15). After generating its rank r_i, party i disseminates it privately to every other party. Specifically, upon receiving a verified public key from party j, it encrypts its rank under $\mathsf{eph_pk}_j$ and sends the resulting ciphertext $c_{j,i}$ together with $\pi_{\mathsf{rank},i}$ to j. When j receives the first encrypted rank/ciphertext from i, it stores the ciphertext and replies with a *stored* acknowledgment. Because ranks remain encrypted until the escrowed secret keys are retrieved, the adversary cannot bias the election outcome by learning honest ranks before deciding whether to deliver or delay them.

Phase 3: Storage confirmation and coordinated decryption (lines 16-25). After receiving *stored* acknowledgments from $n - f$ parties, party i knows that its encrypted rank is widely stored and sends a *ready* message to all parties. Once a party receives $n - f$ *ready* messages and completes $\mathsf{KeyEscrow}$, it invokes

KeyRetrieve to obtain its secret key. At that point, at least $f+1$ honest parties have their encrypted ranks stored by at least $f+1$ honest parties. Since the $f+1$ honest parties whose ranks are widely stored form more than one third of the system, with probability at least $\frac{1}{3}$ one of them holds the global maximal rank. In that case, the maximal rank belongs to an honest party whose value is durably stored and will be decrypted by sufficiently many honest parties.

Phase 4: Decryption and leader election (lines 26-38). After retrieving its secret key $\mathsf{eph_sk}_i$ and storing at least $n-f$ encrypted ranks, party i decrypts each ciphertext to recover r_j, verifies the proof $\pi_{\mathsf{rank},j}$, and computes $\mathsf{rank}_j = \mathsf{Hash}(j, r_j)$. It then selects the party ℓ with the maximal valid rank and sends ℓ together with rank_ℓ and $(r_\ell, \pi_{\mathsf{rank},\ell})$ to all parties. Upon receiving $n-f$ valid election messages, each party outputs the identifier ℓ^* corresponding to the highest reported rank. If the global maximal rank belongs to an honest party whose rank was well propagated, every honest party that terminates receives a report from one of the $f+1$ honest parties that stored ℓ^*'s rank and therefore outputs ℓ^* as leader.

Algorithm 6 Election(tag)

1: $\mathsf{ranks}_i \leftarrow \emptyset,\ \mathsf{candidates}_i \leftarrow \emptyset$
// Generate rank and encryption key, send encryption key to everybody
2: call ProvableRank$((tag, \text{“rank”}))$ and KeyEscrow$((tag, \text{“key”}))$
// Send parties my rank encrypted with their encryption key
3: **upon** KeyEscrow$((tag, \text{“key”}))$ terminating with output $(\mathsf{eph_pk}_i, \pi_{\mathsf{eph_pk},i})$, **do**
4: send $\langle \text{“key”}, \mathsf{eph_pk}_i, \pi_{\mathsf{eph_pk},i} \rangle$ to all parties
5: **upon** ProvableRank$((tag, \text{“rank”}))$ terminating with output $(r_i, \pi_{\mathsf{rank},i})$, **do**
6: **upon** receiving the first $\langle \text{“key”}, \mathsf{eph_pk}_j, \pi_{\mathsf{eph_pk},j} \rangle$ message from j, **do**
7: **if** KeyVerify$(\mathsf{eph_pk}_j, \pi_{\mathsf{eph_pk},j}, (tag, \text{“key”})) = 1$ **then**
8: send $\langle \text{“rank”}, \mathsf{Encrypt}(\mathsf{eph_pk}_j, r_i), \pi_{\mathsf{rank},i} \rangle$ to j
// Inform senders about ranks being received, ready to proceed after many others store my rank
9: **upon** receiving the first $\langle \text{“rank”}, c_j, \pi_{\mathsf{rank},j} \rangle$ message from j, **do**
10: $\mathsf{ranks}_i \leftarrow \mathsf{ranks}_i \cup \{(j, c_j, \pi_{\mathsf{rank},j})\}$
11: send $\langle \text{“stored”} \rangle$ to j
12: **upon** receiving $\langle \text{“stored”} \rangle$ from $n-f$ parties, **do**
13: send $\langle \text{“ready”} \rangle$ to all parties
// After many parties are ready to proceed, retrieve keys
14: **upon** receiving $\langle \text{“ready”} \rangle$ from $n-f$ parties and completing the KeyEscrow$((tag, \text{“key”}))$ protocol, **do**
15: call KeyRetrieve$((tag, \text{“key”}))$
// Decrypt all ranks using retrieved decryption key, send highest-ranked party as my elected party
16: **upon** KeyRetrieve$((tag, \text{“key”}))$ returning a key $\mathsf{eph_sk}_i$ and $|\mathsf{ranks}_i| \geq n-f$, **do**
17: **for all** $(j, c_j, \pi_{\mathsf{rank},j}) \in \mathsf{ranks}_i$ **do**
18: $r_j \leftarrow \mathsf{Decrypt}(\mathsf{eph_pk}_i, c_j),\ \mathsf{rank}_j \leftarrow \mathsf{Hash}(j, r_j)$
19: $\ell = \mathsf{argmax}\{\mathsf{rank}_j \mid j \ s.t.\ (j, c_j, \pi_{\mathsf{rank},j}) \in \mathsf{ranks}_i \ \wedge\ \mathsf{RankVerify}(r_j, j, \pi_{\mathsf{rank},j}, (tag, \text{“rank”})) = 1\}$
20: send $\langle \text{“elected”}, \ell, \mathsf{rank}_\ell, r_\ell, \pi_{\mathsf{rank},\ell} \rangle$ to all parties
// Pick the highest-ranked party among all parties' elected parties
21: **upon** receiving $\langle \text{“elected”}, \ell_j, \mathsf{rank}_{\ell_j}, r_{\ell_j}, \pi_{\mathsf{rank},\ell_j} \rangle$ from j, **do**
22: **if** $\mathsf{rank}_j = \mathsf{Hash}(j, r_j)$ and $\mathsf{RankVerify}(r_{\ell_j}, \ell_j, \pi_{\mathsf{rank},\ell_j}, (tag, \text{“rank”})) = 1$ **then**
23: $\mathsf{candidates}_i \leftarrow \mathsf{candidates}_i \cup \{(j, \ell_j, \mathsf{rank}_{\ell_j})\}$
24: **if** $|\mathsf{candidates}_i| = n-f$ **then**
25: $(k, \ell^*, \mathsf{rank}_{\ell^*}) \leftarrow \mathsf{argmax}\{\mathsf{rank}_{\ell_k} \mid (k, \ell_k, \mathsf{rank}_{\ell_k}) \in \mathsf{candidates}_i\}$
26: **output** ℓ^*

7.3 Analysis

Due to space constraints, we defer the proofs to Section C.3.

Theorem 3. *The protocol* Election *(cf. Algorithm 6) is a Weak Leader Election protocol with* $\alpha = \frac{1}{3}$ *resilient to* $f < n/3$ *Byzantine parties, except with negligible probability in* λ*. Moreover, the* Election *protocol runs in* $O(\lambda^2 n^2)$ *communication and* $O(1)$ *rounds.*

Remark 2. *A Weak Common Coin protocol lets parties jointly sample a value from a distribution* $\mathcal{D}$*. With constant probability (the "success event"), all parties output the same sampled value; otherwise, no guarantees are provided. Additionally, the coin's value must be unpredictable before any honest party invokes the protocol (see [ABLS25, FT24] for a formal definition). A coin can be obtained by slightly adapting our* Election *protocol: parties output* $\mathsf{Hash}(\ell^*, \mathsf{rank}_{\ell^*})$ *in the end instead of just* ℓ^**. As shown in the proof of Theorem 3, in the good event where an honest party is elected, its rank is unpredictable to the adversary before any party calls* KeyRetrieve*, after which the leader is fixed. Hence,* $\mathsf{Hash}(\ell^*, \mathsf{rank}_{\ell^*})$ *is random and independent of the adversary's view at that time.*

7.4 Putting the Pieces Together: Our ADKG

We obtain our ADKG protocol as follows. First, parties call the PVSSExchange protocol [ABLS25] in order to obtain a possible valid output from the ADKG protocol for parties to agree on. Recall that at least one honest party outputs an aggregated PVSS transcript from the protocol. That party provably disperses it, and sends the proof to all parties. The parties then execute a VABA protocol to agree on which dispersed aggregated transcript to retrieve. This is done using the consensus protocol of Abraham et al. [ABLS25], replacing their recursive weak coin generation procedure with our weak leader election (equivalently, weak coin) protocol. Their VABA protocol proceeds in consecutive views, which can be thought of as attempts to reach consensus.

In each view, parties call the weak coin protocol once, and incur $O(\lambda n^2)$ communication and $O(1)$ rounds. Since our weak coin protocol has constant probability of success, the VABA protocol requires a constant expected number of views until all parties terminate. Finally, parties can retrieve the agreed upon transcript, which can be output from the ADKG protocol. The Retrieve protocol requires an additional $O(\lambda n^2)$ cost and $O(1)$ rounds. Consequently, the overall expected communication cost of our ADKG protocol is $O(\lambda^2 n^2)$, and it completes in expected $O(1)$ rounds.

On Forward-Secure Signatures. In our protocol, we use forward-secure signatures whose keys are updated during the PVSS Exchange protocol. This is done to prevent the adversary from learning who is allowed to send messages to whom and then adaptively corrupting them while they still hold the signing keys. Parties update their keys in the PVSS Exchange protocol and erase the old key, and only then send the first message of the protocol. However, to verify signatures generated with a signing key, parties need to know how many times the key was updated.

Thankfully, at each point throughout the ADKG protocol, parties know exactly how many times the key was updated. Each call to the election protocol (once per view) starts by calling the Key Escrow and Provable Rank protocols once, both beginning with a call to the PVSS Exchange protocol. Parties update their signing key once in each one of these calls. In addition, parties call the PVSS Exchange protocol once at the beginning of the ADKG protocol, and thus update the key one more time. This means that parties can know how many times the signing key was updated as a function of the view number, which is attached to every message in consensus protocols.

On Multi-Instance ADKG. While our construction allows parties to run an ADKG protocol to set up threshold cryptosystems, parties need to update their secret key from the forward-secure signature scheme in the process. For many purposes, simply having a single threshold system setup is enough (and it can also be used as a standard PKI setup for a non-threshold signature scheme). However, if one wishes for parties to keep their original secret signing key without erasing it after an update, e.g., for running many instances of ADKG, one can add another round before executing each ADKG instance. Parties start by generating

another key pair for a forward-secure signature scheme and signing the public key. This signing key pair will be used as an ephemeral key pair for the ADKG instance. Crucially, parties can update this ephemeral secret key without changing their permanent signing key. Parties then execute a Non-Equivocation round (Algorithm 7), inputting their ephemeral public key, to generate a proof that this key is the unique key used in the given ADKG instance. To allow verification of signatures using this key, parties also send their ephemeral public key and a proof of its uniqueness whenever signing or forwarding a signature.

References

[ABLS25] Ittai Abraham, Renas Bacho, Julian Loss, and Gilad Stern. Nearly quadratic asynchronous distributed key generation. Cryptology ePrint Archive, Paper 2025/006, 2025. 2, 3, 6, 7, 8, 9, 10, 20, 23, 29, 35, 36

[ACD+23] Ittai Abraham, T.-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. *Distributed Comput.*, 36(1):3–28, 2023. 3, 7

[AGL+23] Arasu Arun, Chaya Ganesh, Satya Lokam, Tushar Mopuri, and Sriram Sridhar. Dew: A transparent constant-sized polynomial commitment scheme. In *Public-Key Cryptography – PKC 2023: 26th IACR International Conference on Practice and Theory of Public-Key Cryptography, Atlanta, GA, USA, May 7–10, 2023, Proceedings, Part II*, page 542–571, Berlin, Heidelberg, 2023. Springer-Verlag. 9, 39, 50, 56

[AJM+21] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 363–373. ACM, 2021. 1, 3, 4, 8

[AJM+23] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, and Gilad Stern. Bingo: Adaptivity and asynchrony in verifiable secret sharing and distributed key generation. In *Annual International Cryptology Conference*, pages 39–70. Springer, 2023. 3, 8

[AMS19] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 337–346. ACM, 2019. 9

[AS22] Ittai Abraham and Alexander Spiegelman. Provable broadcast, 2022. 4

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I*, page 561–586, Berlin, Heidelberg, 2019. Springer-Verlag. 37

[BCLZL23] Renas Bacho, Daniel Collins, Chen-Da Liu-Zhang, and Julian Loss. Network-agnostic security comes (almost) for free in dkg and mpc. In *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part I*, page 71–106, Berlin, Heidelberg, 2023. Springer-Verlag. 8

[BDK13] Michael Backes, Amit Datta, and Aniket Kate. Asynchronous computational vss with reduced communication complexity. In *Proceedings of the 13th International Conference on Topics in Cryptology*, CT-RSA'13, page 259–276, Berlin, Heidelberg, 2013. Springer-Verlag. 3, 8

[BdM94] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 274–285, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. 29, 36

[BF01] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, page 213–229, Berlin, Heidelberg, 2001. Springer-Verlag. 9, 13

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 103–112, New York, NY, USA, 1988. Association for Computing Machinery. 9, 32

[BHK+24] Fabrice Benhamouda, Shai Halevi, Hugo Krawczyk, Yiping Ma, and Tal Rabin. Sprint: High-throughput robust distributed schnorr signatures. In *Advances in Cryptology – EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26–30, 2024, Proceedings, Part V*, page 62–91, Berlin, Heidelberg, 2024. Springer-Verlag. 8

[BK25] Renas Bacho and Alireza Kavousi. SoK: Dlog-Based Distributed Key Generation . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 614–632, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. 2

[BL23] Renas Bacho and Julian Loss. Adaptively secure (aggregatable) pvss and application to distributed randomness beacons. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1791–1804, New York, NY, USA, 2023. Association for Computing Machinery. 11, 30, 52, 54

[BLL+24] Renas Bacho, Christoph Lenzen, Julian Loss, Simon Ochsenreither, and Dimitrios Papachristoudis. Grandline: Adaptively secure dkg and randomness beacon with (log-)quadratic communication complexity. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 941–955, New York, NY, USA, 2024. Association for Computing Machinery. 7, 11, 12, 13, 45, 60

[BLSW24] Renas Bacho, Julian Loss, Gilad Stern, and Benedikt Wagner. Harts: High-threshold, adaptively secure, and robust threshold schnorr signatures. In *Advances in Cryptology – ASIACRYPT 2024: 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9–13, 2024, Proceedings, Part III*, page 104–140, Berlin, Heidelberg, 2024. Springer-Verlag. 8

[BM99] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 431–448, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. 9, 30

[BNPS03] Mihir Bellare, Chanathip Namprempre, David Pointcheval, and Michael Semanko. The one-more-rsa-inversion problems and the security of chaum's blind signature scheme. *Journal of Cryptology*, 16:185–215, 2003. 33

[BSKN23] Adithya Bhat, Nibesh Shrestha, Aniket Kate, and Kartik Nayak. Optrand: Optimistically responsive reconfigurable distributed randomness. *Proceedings 2023 Network and Distributed System Security Symposium*, 2023. 11

[CD17] Ignacio Cascudo and Bernardo David. Scrape: Scalable randomness attested by public entities. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *Proceedings of the 15th International Conference on Applied Cryptography and Network Security (ACNS 2017)*, volume 10355 of *Lecture Notes in Computer Science*, pages 537–556. Springer, Cham, 2017. 11

[CGJ+99] Ran Canetti, Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Adaptive security for threshold cryptosystems. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 98–116, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. 7

[CKPS01] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. *IACR Cryptol. ePrint Arch.*, page 6, 2001. 8

[DR85] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, 1985. 7

[DXKKR23] Sourav Das, Zhuolun Xiang, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5359–5376, Anaheim, CA, August 2023. USENIX Association. 3, 8

[DXR24] Sourav Das, Zhuolun Xiang, and Ling Ren. Powers of tau in asynchrony. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024. 3, 4

[DY81] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, SFCS '81, page 350–357, USA, 1981. IEEE Computer Society. 8

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *Public Key Cryptography - PKC 2005*, pages 416–431, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. 9

[DYX+22] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew K. Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2518–2534, 2022. 3

[FGL+25] Hanwen Feng, Yingzi Gao, Yuan Lu, Qiang Tang, and Jing Xu. Practical asynchronous distributed key reconfiguration and its applications. *Cryptology ePrint Archive*, 2025. 2, 3, 5, 20

[FLMT24] Hanwen Feng, Zhenliang Lu, Tiancheng Mai, and Qiang Tang. Making hash-based MVBA great again. *IACR Cryptol. ePrint Arch.*, page 479, 2024. 7

[FLT24] Hanwen Feng, Zhenliang Lu, and Qiang Tang. Dragon: Decentralization at the cost of representation after arbitrary grouping and its applications to sub-cubic dkg and interactive consistency. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, PODC '24, page 469–479, New York, NY, USA, 2024. Association for Computing Machinery. 7, 11

[FT24] Hanwen Feng and Qiang Tang. Asymptotically optimal adaptive asynchronous common coin and DKG with silent setup. *IACR Cryptol. ePrint Arch.*, page 2098, 2024. 2, 3, 4, 8, 20, 23

[Fu25] Shihui Fu. Improved constant-sized polynomial commitment schemes without trusted setup. *IACR Communications in Cryptology*, 2(2), 2025. 9, 39, 50, 56

[GJKR07] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptol.*, 20(1):51–83, January 2007. 7

[GJM+21] Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. In *Advances in Cryptology – EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I*, page 147–176, Berlin, Heidelberg, 2021. Springer-Verlag. 7, 11, 12

[GKKS+22] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In Ittay Eyal and Juan Garay, editors, *Financial Cryptography and Data Security*, pages 296–315, Cham, 2022. Springer International Publishing. 1

[GKPW24] Sanjam Garg, Dimitris Kolonelos, Guru-Vamsi Policharla, and Mingyuan Wang. Threshold encryption with silent setup. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part VII*, volume 14926 of *Lecture Notes in Computer Science*, pages 352–386. Springer, 2024. 4

[GLL+21] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. *CoRR*, abs/2106.07831, 2021. 3, 8

[GS24] Jens Groth and Victor Shoup. Fast batched asynchronous distributed key generation. In *Advances in Cryptology – EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26–30, 2024, Proceedings, Part V*, page 370–400, Berlin, Heidelberg, 2024. Springer-Verlag. 8

[KKMS20] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1751–1767, New York, NY, USA, 2020. Association for Computing Machinery. 1, 3, 8

[KO17] Jihye Kim and Hyunok Oh. Forward-secure digital signature schemes with optimal computation and storage of signers. In Sabrina De Capitani di Vimercati and Fabio Martinelli, editors, *ICT Systems Security and Privacy Protection*, pages 523–537, Cham, 2017. Springer International Publishing. 9

[LLTW20] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 129–138. ACM, 2020. 4

[Mer87] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, page 369–378, Berlin, Heidelberg, 1987. Springer-Verlag. 37

[MR17] Achour Mostéfaoui and Michel Raynal. Signature-free asynchronous byzantine systems: from multivalued to binary consensus with t < n/3, o(n2) messages, and constant time. *Acta Inf.*, 54(5):501–520, August 2017. 5

[MVR99] Silvio Micali, Salil Vadhan, and Michael Rabin. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, page 120, USA, 1999. IEEE Computer Society. 9, 31

[Ped91] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In Donald W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, pages 522–526, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. 1, 7

[RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960. 29, 36

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. 9, 29

[SBKN24] Nibesh Shrestha, Adithya Bhat, Aniket Kate, and Kartik Nayak. Synchronous distributed key generation without broadcasts. *IACR Communications in Cryptology*, 07 2024. 7

[Sta96] Markus Stadler. Publicly verifiable secret sharing. In *International Conference on the Theory and Application of Cryptographic Techniques*, 1996. 3

[YMR+19] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery. 1

[ZDL+23] Haibin Zhang, Sisi Duan, Chao Liu, Boxin Zhao, Xuanji Meng, Shengli Liu, Yong Yu, Fangguo Zhang, and Liehuang Zhu. Practical asynchronous distributed key generation: Improved efficiency, weaker assumption, and standard model. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–581, 2023. 3

A Formal Definitions

In this section, we recall the formal definitions of the primitives used in this paper, many of which are taken verbatim from prior work [ABLS25].

Definition 8 (Reed-Solomon Code [RS60])**.** *A Reed-Solomon code with parameters $(q, b) \in \mathbb{N}^2$ is a tuple of algorithms $\Sigma = (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Decode})$ with the following properties:*

- $\mathsf{Setup}(1^\lambda, q, b) \to par$: *The* setup algorithm *takes as input the security parameter 1^λ and parameters q and b. It outputs system parameters par. All other algorithms implicitly take par as input.*
- $\mathsf{Encode}(m) \to (s_1, \ldots, s_q)$: *The deterministic* encoding algorithm *takes as input a message m (which is internally split into b data symbols $m_1, \ldots, m_b$). It outputs a code word $(s_1, \ldots, s_q)$ of length q. Knowledge of any b elements of the code word uniquely determines the input message and the remaining of the code word.*
- $\mathsf{Decode}(s_1, \ldots, s_q) \to m$: *The deterministic* decoding algorithm *takes as input a code word $(s_1, \ldots, s_q)$ of length q. It outputs a decoded message m. This algorithm tolerates up to c errors and d erasures in a code word $(s_1, \ldots, s_q)$ if and only if $q - b \geq 2c + d$.*

Definition 9 (Cryptographic Accumulator [BdM94])**.** *A cryptographic accumulator scheme is a tuple of algorithms $\Sigma = (\mathsf{Setup}_{\mathsf{Acc}}, \mathsf{Gen}_{\mathsf{Acc}}, \mathsf{Eval}_{\mathsf{Acc}}, \mathsf{Witness}_{\mathsf{Acc}}, \mathsf{Verify}_{\mathsf{Acc}})$ with the following properties:*

- $\mathsf{Setup}_{\mathsf{Acc}}(1^\lambda, n) \to par$: *The* setup algorithm *takes as input the security parameter 1^λ and an accumulation threshold n. It outputs system parameters par. All other algorithms implicitly take par as input.*
- $\mathsf{Gen}_{\mathsf{Acc}}(par) \to ak$: *The randomized* accumulator key generation algorithm *takes as input system parameters par. It outputs a public accumulator key ak.*
- $\mathsf{Eval}_{\mathsf{Acc}}(ak, D) \to z$: *The deterministic* evaluation algorithm *takes as input an accumulator key ak and a list of n elements $D := (d_1, \ldots, d_n)$. It outputs an accumulation value z for D.*
- $\mathsf{Witness}_{\mathsf{Acc}}(ak, z, d, D) \to w/\perp$: *The possibly randomized* witness generation algorithm *takes as input an accumulator key ak, an accumulation value z, an element d, and a list of elements D. It outputs $\perp$ if $d \notin D$, and a witness w otherwise.*
- $\mathsf{Verify}_{\mathsf{Acc}}(ak, z, d, w, D) \to 0/1$: *The deterministic* verification algorithm *takes as input an accumulator key ak, an accumulation value z, an element d, a witness w, and a list of element D. It outputs 1 (accept) if w is a valid proof for membership $d \in D$ and 0 (reject) otherwise.*

Definition 10 (Public-Key Encryption Scheme [RSA78])**.** *A public-key encryption scheme is a tuple of algorithms $\mathsf{PKE} = (\mathsf{Setup}, \mathsf{KGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ with the following properties:*

- $\mathsf{Setup}(1^\lambda) \to par$: *The* setup algorithm *takes as input the security parameter 1^λ. It outputs system parameters par. All other algorithms implicitly take par as input.*
- $\mathsf{KGen}(par) \to (\mathsf{ek}, \mathsf{dk})$: *The randomized* key generation algorithm *takes as input system parameters par. It outputs an encryption key ek and a decryption key dk.*
- $\mathsf{Encrypt}(\mathsf{ek}, m)$*:* *The possibly randomized* encryption algorithm *takes as input an encryption key ek and a message m. It outputs a ciphertext c.*
- $\mathsf{Decrypt}(\mathsf{dk}, c)$*:* *The deterministic* decryption algorithm *takes as input a decryption key dk and a ciphertext c. It outputs a message m or a special symbol $\perp$ indicating failure.*

Definition 11 (CPA Security)**.** *A public-key encryption scheme $\mathsf{PKE} = (\mathsf{Setup}, \mathsf{KGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ is said to be indistinguishable under chosen-plaintext attack (CPA) secure if for every PPT adversary $\mathcal{A}$, the following advantage is negligible in λ:*

$$\mathsf{Adv}^{\mathsf{CPA}}_{\mathsf{PKE},\mathcal{A}}(\lambda) := \left| \Pr\left[b' = b\right] - \frac{1}{2} \right|,$$

where the probability is taken over the following security experiment:

1. *Run the key generation algorithm and get* $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KGen}(1^\lambda)$.
2. $\mathcal{A}(\mathsf{ek})$ *outputs two messages* (m_0, m_1) *of equal length.*
3. *A bit* $b \leftarrow \{0,1\}$ *is sampled uniformly at random.*
4. *The challenger computes* $c^* \leftarrow \mathsf{Encrypt}(\mathsf{ek}, m_b)$ *and gives* c^* *to* $\mathcal{A}$.
5. *When* $\mathcal{A}$ *outputs a bit* b'*, the experiment returns* b'.

Definition 12 (Key-Evolving Digital Signature Scheme [BM99])**.** *A key-evolving digital signature scheme is a tuple of algorithms* $\mathsf{KDS} = (\mathsf{Setup}_{\mathsf{KDS}}, \mathsf{KGen}_{\mathsf{KDS}}, \mathsf{Update}_{\mathsf{KDS}}, \mathsf{Sign}_{\mathsf{KDS}}, \mathsf{Verify}_{\mathsf{KDS}})$ *with the following properties:*

- $\mathsf{Setup}_{\mathsf{KDS}}(1^\lambda, T) \to par$: *The* setup algorithm *takes as input the security parameter* 1^λ *and the total number of periods* $T \in \mathbb{N}$ *over which the scheme will operate. It outputs system parameters par. All other algorithms implicitly take par as input.*
- $\mathsf{KGen}_{\mathsf{KDS}}(par) \to (\mathsf{pk}, \mathsf{sk}_0)$: *The randomized* key generation algorithm *takes as input system parameters par. It outputs a base public key* pk *and corresponding base secret key* sk_0.
- $\mathsf{Update}_{\mathsf{KDS}}(\mathsf{sk}_r) \to \mathsf{sk}_{r+1}$: *The deterministic* secret key update algorithm *takes as input a secret key* sk_r *of some period* $r \in \mathbb{N}_0$*. It outputs an updated secret key* sk_{r+1} *for the period* $r+1$.
- $\mathsf{Sign}_{\mathsf{KDS}}(\mathsf{sk}_r, m) \to \sigma$: *The possibly randomized* signature generation algorithm *takes as input the current secret key* sk_r *and a message* m*. It outputs a signature* σ_r *for the period* r.
- $\mathsf{Verify}_{\mathsf{KDS}}(\mathsf{pk}, m, \sigma_r) \to 0/1$: *The deterministic* verification algorithm *takes as input the public key* pk*, a message* m*, and a signature* σ_r*. It outputs 1 (accept) if* σ_r *is a valid signature for period* r *and 0 (reject) otherwise.*

Definition 13 (Aggregatable PVSS Scheme [BL23])**.** *A* (t,n)-threshold aggregatable PVSS (APVSS) scheme over group $\mathbb{G}$ *is a tuple of algorithms* $\mathsf{APVSS} = (\mathsf{Setup}, \mathsf{KGen}, \mathsf{Encr}, \mathsf{Decr}, \mathsf{Dist}, \mathsf{Agg}, \mathsf{Contr}, \mathsf{Ver}, \mathsf{Rec})$ *such that:*

- $\mathsf{Setup}(1^\lambda) \to par$: *The* setup algorithm *takes as input the security parameter* 1^λ*. It outputs system parameters par. All other algorithms implicitly take par as input.*
- $\mathsf{KGen}(par) \to (\mathsf{pk}, \mathsf{sk})$: *The randomized* key generation algorithm *takes as input system parameters par. It outputs a public encryption key* pk *and a secret decryption key* sk.
- $\mathsf{Encr}(\mathsf{pk}, m) \to c$: *The randomized* encryption algorithm *takes as input an encryption key* pk *and a message* m*. It outputs a ciphertext* c*. We may also write* $\mathsf{Encr}_{\mathsf{pk}}(m)$ *instead of* $\mathsf{Encr}(\mathsf{pk}, m)$.
- $\mathsf{Decr}(\mathsf{sk}, c) \to m$: *The deterministic* decryption algorithm *takes as input a decryption key* sk *and a ciphertext* c*. It outputs a message* m *(optionally with a proof of correct decryption). We may sometimes also write* $\mathsf{Decr}_{\mathsf{sk}}(c)$ *instead of* $\mathsf{Decr}(\mathsf{sk}, c)$*. Further, for all messages* m *and keys* $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{KGen}(par)$*, we require that* $\Pr\left[\mathsf{Decr}_{\mathsf{sk}}(\mathsf{Encr}_{ek}(m)) = m\right] = 1$.
- $\mathsf{Dist}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \to (\vec{E}, \pi)$: *The randomized* secret sharing algorithm *takes as input a list of* n *encryption keys* $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$*. It outputs a vector of encrypted shares* $\vec{E} := (\mathsf{Encr}_{\mathsf{pk}_1}(S_1), \ldots, \mathsf{Encr}_{\mathsf{pk}_n}(S_n))$ *along with a proof* π*, where* $S_1, \ldots, S_n$ *are* (t,n)*-threshold shares of a secret* $S = S_0 \in \mathbb{G}$*. We also refer to the tuple* $T := (\vec{E}, \pi)$ *as a* PVSS transcript.
- $\mathsf{Agg}(\{(\vec{E}_i, \pi_i)\}_{i \in [k]}) \to ((\vec{E}, \pi), \vec{w})$: *The deterministic* aggregation algorithm *takes as input PVSS transcripts* $(\vec{E}_1, \pi_1), \ldots, (\vec{E}_k, \pi_k)$ *for* $k \in \mathbb{N}$*. It outputs an* (aggregated) PVSS transcript $T := (\vec{E}, \pi)$ *along with a weight vector* $\vec{w} \in \mathbb{N}_0^n$ *(indication contributing parties and their weights).*
- $\mathsf{Contr}((\vec{E}, \pi), \vec{w}) \to I$: *The deterministic* contributor identifier algorithm *takes as input an (aggregated) PVSS transcript* $T = (\vec{E}, \pi)$ *with weight vector* $\vec{w}$*. It outputs a set of indices* $I \subseteq [n]$ *specifying the contributors.*

- $\mathsf{Ver}((\vec{E}, \pi), \vec{w}, (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)) \to 0/1$: *The deterministic* verification algorithm *takes as input an (aggregated) PVSS transcript* $T = (\vec{E}, \pi)$ *with weight vector* $\vec{w}$*, and a list of* n *encryption keys* $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$*. It outputs 1 (valid transcript) or 0 (invalid transcript).*
- $\mathsf{Rec}(S_1, \ldots, S_{t+1}) \to S$: *The deterministic* reconstruction algorithm *takes as input* $t+1$ *(decrypted) shares* $S_1, \ldots, S_{t+1}$*. It outputs a secret* $S \in \mathbb{G}$*. (If there are more inputs, choose the first* $t+1$ *lexicographically.)*

Definition 14 (Aggregated Unpredictability)**.** *Let* APVSS *be a* (t, n)*-threshold aggregatable PVSS scheme. For any adversary* $\mathcal{A}$ *corrupting at most* f *parties, its probability of winning in the following security experiment* AggUnpred *is negligible in* λ*.*

- ***Setup Phase.*** *For all* $i \in [n]$*, run* $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{Keys}(\mathsf{par}, i)$*. On input* par *and* $\{\mathsf{pk}_i\}_{i \in [n]}$*, the adversary* $\mathcal{A}$ *outputs an index set* $\mathcal{C} \subseteq [n]$ *with* $|\mathcal{C}| \leq f$ *of initially corrupted parties, together with updated public keys* $\{\hat{\mathsf{pk}}_j\}_{j \in \mathcal{C}}$*. For all* $j \in \mathcal{C}$*, set* $\mathsf{pk}_j := \hat{\mathsf{pk}}_j$*. Let* $\mathcal{H} := [n] \setminus \mathcal{C}$ *denote the set of honest parties. Let* $\mathcal{L} := \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ *be the public key list.*
- ***Transcript Oracle.*** *The adversary* $\mathcal{A}$ *is given oracle access to* $\mathcal{O}_{\mathsf{PVSS}}$*. On query* $(\mathsf{givePVSS}, i)$ *for some* $i \in \mathcal{H}$*, the oracle runs* $\mathsf{ADist}(\mathsf{sk}_i, \mathcal{L})$*, and returns the resulting transcript* T *to* $\mathcal{A}$*.*
- ***Corruption Oracle.*** *The adversary* $\mathcal{A}$ *is given oracle access to* $\mathcal{O}_{\mathsf{corr}}$*. On query* $i \in \mathcal{H}$*, the oracle returns party* i*'s internal state, including the secret key. Update* $\mathcal{C} := \mathcal{C} \cup \{i\}$ *and* $\mathcal{H} := \mathcal{H} \setminus \{i\}$*.*
- ***Winning Condition.*** *The adversary outputs an aggregated transcript* $T^* = ((\vec{E}^*, \pi^*), \vec{w}^*)$ *and a value* S^**. The adversary wins if and only if all of the following conditions hold:*
  1. $\mathsf{AVer}(\mathcal{L}, T^*) = 1$*,*
  2. $\mathsf{Contr}(T^*) \cap \mathcal{H} \neq \emptyset$*,*
  3. $S^* = \mathsf{Rec}\big(\{\mathsf{Dec}_{\mathsf{sk}_i}(E_i^*)\}_{i \in \mathcal{H}}\big)$*.*

  Informally, these conditions require that the aggregated transcript is valid, that it contains at least one contribution from an honest party, and that S^* *is exactly the secret encoded by the transcript.*

Definition 15 (Verifiable Random Function [MVR99])**.** *A verifiable random function is a tuple of algorithms* $\mathsf{VRF} = (\mathsf{Setup}_{\mathsf{VRF}}, \mathsf{Gen}_{\mathsf{VRF}}, \mathsf{Eval}_{\mathsf{VRF}}, \mathsf{Verify}_{\mathsf{VRF}})$ *with the following properties:*

- $\mathsf{Setup}_{\mathsf{VRF}}(1^\lambda) \to par$: *The* setup algorithm *takes as input the security parameter* 1^λ*. It outputs system parameters* par*. All other algorithms implicitly take* par *as input.*
- $\mathsf{Gen}_{\mathsf{VRF}}(par) \to (\mathsf{pk}, \mathsf{sk})$: *The randomized* key generation algorithm *takes as input system parameters* par*. It outputs a public key* pk *and a corresponding secret key* sk*.*
- $\mathsf{Eval}_{\mathsf{VRF}}(\mathsf{sk}, m) \to (u, \pi)$: *The deterministic* function evaluation algorithm *takes as input a secret key* sk *and a message* m*. It outputs a function value* $u \in S$ *along with a proof* π*. Here,* S *is a finite set denoting the codomain of the function.*
- $\mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}, m, u, \pi) \to 0/1$: *The deterministic* verification algorithm *takes as input the public key* pk*, a message* m*, an output* u*, and a proof* π*. It outputs 1 (accept) or 0 (reject).*

Additionally, VRF *must satisfy the following security properties.*

- ***Correctness.*** *For all* $\lambda \in \mathbb{N}$*, for all* $par \in \mathsf{Setup}(1^\lambda)$*, for all* $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{Gen}_{\mathsf{VRF}}(par)$*, for all* m*, and for all* (u, π) *in the image of* $\mathsf{Eval}_{\mathsf{VRF}}(\mathsf{sk}, m)$*, it holds* $\mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}, m, u, \pi) = 1$*.*
- ***Unique Provability.*** *For all possible* pk *(not necessarily in the image of* $\mathsf{Gen}_{\mathsf{VRF}}$*), for all* m*, for all* $u_1, u_2 \in S$*, and for all possible proofs* π_1, π_2*, the following implication holds:*

$$(\mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}, m, u_1, \pi_1) = \mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}, m, u_2, \pi_2) = 1) \implies (u_1 = u_2)\,.$$

  Informally, this means that for every message m*, there exists a valid proof* π *for at most one function value* u*.*

- ***Pseudorandomness.*** *For all PPT adversaries $\mathcal{A}$, its advantage in the pseudorandomness experiment defined hereafter is negligible:*
  - *Offline Phase. Initialize set $\mathcal{M} := \emptyset$. Run the setup algorithm on input 1^λ to obtain system parameters $par \leftarrow \mathsf{Setup}_{\mathsf{VRF}}(1^\lambda)$. Then, run the key generation algorithm on input par to obtain keys $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}_{\mathsf{VRF}}(par)$. Run $\mathcal{A}$ on input pk.*
  - *Evaluation Queries. At any point of the experiment, $\mathcal{A}$ may submit a message m. In this case, return $(u, \pi) \leftarrow \mathsf{Eval}_{\mathsf{VRF}}(\mathsf{sk}, m)$ and update $\mathcal{M} := \mathcal{M} \cup \{m\}$.*
  - *Online Phase. When $\mathcal{A}$ outputs a message m^*, run the evaluation algorithm on it to obtain $(u_0, \pi) \leftarrow \mathsf{Eval}_{\mathsf{VRF}}(\mathsf{sk}, m^*)$. Sample a bit $b \leftarrow\!\!\$\ \{0,1\}$ and a value $u_1 \leftarrow\!\!\$\ S$ uniformly at random.*
  - *Winning Condition. On input u_b, $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$. Return 1 if $b' = b$ and $m^* \notin \mathcal{M}$. Otherwise, return 0.*

Definition 16 (Non-Interactive Proof System [BFM88])**.** *Let $\mathcal{R}$ be an* **NP** *relation and* H *be a random oracle. A non-interactive proof system for $\mathcal{R}$ with respect to* H *is a tuple of PPT algorithms* $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVerify})$ *with oracle access to* H *with the following properties:*

- $\mathsf{PProve}^{\mathsf{H}}(x, w) \to \pi$: *The randomized* proof generation algorithm *takes as input a statement x and a witness w. It outputs a proof π.*
- $\mathsf{PVerify}^{\mathsf{H}}(x, \pi) \to 0/1$: *The deterministic* verification algorithm *takes as input a statement x and a proof π. It outputs 1 (accept) or 0 (reject).*

We require completeness *to hold: For any $(x, w) \in \mathcal{R}$, we have*

$$\Pr\left[\mathsf{PVerify}^{\mathsf{H}}(x, \pi) = 1 \;\middle|\; \pi \leftarrow \mathsf{PProve}^{\mathsf{H}}(x, w)\right] = 1.$$

We also require soundness *to hold. Concretely, we say that* PS *is sound if for every PPT algorithm $\mathcal{A}$, the following advantage is negligible:*

$$\Pr\left[b = 1 \;\wedge\; \forall w : (x, w) \notin \mathcal{R} \;\middle|\; (x, \pi) \leftarrow \mathcal{A}^{\mathsf{H}},\; b := \mathsf{PVerify}^{\mathsf{H}}(x, \pi)\right].$$

We also require some of our NIZKs to satisfy knowledge soundness*. Informally, knowledge soundness guarantees that any prover that produces an accepting proof must "know" a corresponding witness.*

Formally, we say that PS *is knowledge-sound if there exists a (possibly stateful) PPT extractor* Ext *such that for every non-uniform PPT adversary $\mathcal{A}$, the following advantage is negligible:*

$$\Pr\left[(x^*, \pi^*, w^*) \leftarrow \mathsf{Ext}^{\mathsf{H}}(\mathcal{A}) \;\middle|\; \mathsf{PVerify}^{\mathsf{H}}(x^*, \pi^*) = 1,\; (x^*, w^*) \notin \mathcal{R}\right].$$

That is, whenever $\mathcal{A}$ outputs a statement–proof pair (x^*, π^*) that verifies, the extractor Ext *is able to extract a witness w^* such that $(x^*, w^*) \in \mathcal{R}$, except with negligible probability.*

Further, the notion of simulation-soundness *strengthens standard soundness. Concretely, we say that* PS *is simulation-sound, if there exists a (possibly stateful) PPT simulator* PSim *such that for every non-uniform PPT adversary $\mathcal{A}$, the following advantage is negligible:*

$$\Pr\left[(x^*, \pi^*) \leftarrow \mathcal{A}^{\mathsf{H}^{\mathsf{PSim}}, \mathsf{Oracle}_1} \;\middle|\; \mathsf{PVerify}^{\mathsf{H}^{\mathsf{PSim}}}(x^*, \pi^*) = 1,\; x^* \notin \mathcal{L},\; x^* \notin Q\right],$$

where $\mathsf{H}^{\mathsf{PSim}}$ *denotes a random oracle simulated by* PSim*,* Oracle_1 *is an oracle that on input $(x, w) \in \mathcal{R}$ returns $\pi \leftarrow \mathsf{PSim}(x)$, and Q denotes the set of statements queried to* Oracle_1*.*

We also require the proof system to be zero-knowledge*. Concretely, we say that* PS *is zero-knowledge, if there exists a (possibly stateful) PPT algorithm* PSim *such that for every (potentially unbounded) adversary $\mathcal{A}$ with polynomial-time access to* H*, the following advantage is negligible:*

$$\left|\Pr\left[\mathcal{A}^{\mathsf{H}, \mathsf{Oracle}_0} = 1\right] - \Pr\left[\mathcal{A}^{\mathsf{H}^{\mathsf{PSim}}, \mathsf{Oracle}_1} = 1\right]\right|,$$

where $\mathsf{H}^{\mathsf{PSim}}$ *denotes a random oracle simulated by* PSim*, and* Oracle_b *for $b \in \{0,1\}$ takes as input pairs $(x, w) \in \mathcal{R}$ and outputs $\pi \leftarrow \mathsf{PProve}^{\mathsf{H}}(x, w)$ if $b = 0$ and $\pi \leftarrow \mathsf{PSim}(x)$ if $b = 1$.*

Definition 17 (OMDL Assumption [BNPS03])**.** *Let $(\mathbb{G}, p, g)$ be a cyclic group of prime order p with generator $g \in \mathbb{G}$. We denote by DL_g an oracle that on input $\xi := g^z \in \mathbb{G}$ returns the discrete logarithm $z \in \mathbb{Z}_p$ of ξ (to the base g). For an algorithm $\mathcal{A}$ and $k \in \mathbb{N}$, we consider the following security game:*

- Setup phase. *Sample $(z_1, \ldots, z_k) \leftarrow \mathbb{Z}_p^k$ and set $\xi_i := g^{z_i} \in \mathbb{G}$ for all $i \in [k]$.*
- Challenge phase. *Run $\mathcal{A}$ on input $(\mathbb{G}, p, g, h)$ and $(\xi_1, \ldots, \xi_k)$. Here, $\mathcal{A}$ gets access to the oracle DL_g.*
- Winning condition. *Let $(z'_1, \ldots, z'_k)$ denote the output of $\mathcal{A}$. Return 1 if (i) $z'_i = z_i$ for all $i \in [k]$, and (ii) DL_g was queried at most $k-1$ times during the online phase. Otherwise, return 0.*

We say that the OMDL assumption of degree k holds relative to $(\mathbb{G}, p, g)$ if for any probabilistic polynomial-time algorithm $\mathcal{A}$, the probability that the above game outputs 1 is negligible in λ.

A.1 Game-Based Definitions

We give a game-based definition for the secrecy property of Provable Rank (see Section 5).

Definition 18 (Secrecy of Provable Rank)**.** *For every efficient adversary $\mathcal{A}$ that corrupts at most f parties, the probability that $\mathcal{A}$ wins the following security experiment is negligible in λ.*

- ***Setup phase.*** *The challenger gives $\mathcal{A}$ all public parameters of the protocol and initializes two empty sets $\mathcal{T} := \emptyset$ and $\mathcal{O} := \emptyset$.*
- ***Oracle*** $\textsc{ProvableRank}(tag)$*. On input a tag, the challenger checks whether $tag \in \mathcal{T}$. If so, it returns $\perp$. Otherwise, it initiates an execution of the $\mathsf{ProvableRank}(tag)$ protocol among all parties, faithfully emulating the behavior of all honest parties, and records the tag by updating $\mathcal{T} := \mathcal{T} \cup \{tag\}$. For every honest party $j \in \mathcal{H}$, let (rank_j, π_j) denote its output in this execution. The challenger records all such pairs by updating*
$$\mathcal{O} := \mathcal{O} \cup \{(j, tag, \mathsf{rank}_j, \pi_j)\}_{j \in \mathcal{H}}.$$
- ***Challenge phase.*** *The adversary outputs a challenge pair (i, tag^*) such that i is honest and $tag^* \in \mathcal{T}$. The challenger samples a random bit $b \leftarrow \{0,1\}$.*
  - *If $b = 0$: set $(R, \Pi) := (\mathsf{rank}_i, \pi_i)$.*
  - *If $b = 1$: sample $\tilde{r} \leftarrow \mathcal{D}$ uniformly at random and generate $\tilde{\pi}$ such that $\mathsf{RankVerify}(\tilde{r}, i, \tilde{\pi}, tag^*) = 1$, and set $(R, \Pi) := (\tilde{r}, \tilde{\pi})$.*
- ***Output phase.*** *The challenger gives $\mathcal{A}$:*
$$\Big(R, \Pi, \ \{(\mathsf{rank}_j, \pi_j)\}_{j \neq i \in \mathcal{H}}\Big).$$
- ***Winning condition.*** *The adversary outputs a bit $b' \in \{0,1\}$ and wins if and only if $b' = b$.*

We next give a game-based definition for the uniqueness property of Provable Rank (see Section 5).

Definition 19 (Uniqueness of Provable Rank)**.** *For every efficient adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins the following security experiment is negligible in λ.*

- ***Setup phase.*** *The challenger gives $\mathcal{A}$ all public parameters of the protocol.*
- ***Output phase.*** *The adversary $\mathcal{A}$ outputs two tuples*
$$(\mathsf{rank}, i, \pi, tag) \quad \text{and} \quad (\mathsf{rank}', i, \pi', tag).$$
- ***Winning condition.*** *The adversary wins if and only if $\mathsf{rank} \neq \mathsf{rank}'$ and*
$$\mathsf{RankVerify}(\mathsf{rank}, i, \pi, tag) = 1, \quad \mathsf{RankVerify}(\mathsf{rank}', i, \pi', tag) = 1.$$

We next give a game-based definition for the validity property of Provable Rank (see Section 5).

Definition 20 (Validity of Provable Rank)**.** *For any efficient adversary $\mathcal{A}$ corrupting at most f parties and any efficient distinguisher $\mathcal{B}$, the probability that the following security experiment returns 1 is negligibly close to $\frac{1}{2}$.*

- ***Setup phase.*** *The challenger gives $\mathcal{A}$ all public parameters of the protocol and initializes an empty set $\mathcal{T} := \emptyset$ and an empty multiset $\mathcal{O} := \emptyset$.*
- ***Oracle*** $\textsc{ProvableRank}(tag)$*. On input a tag, the challenger checks whether $tag \in \mathcal{T}$. If so, it returns $\perp$. Otherwise, it initiates an execution of the* $\mathsf{ProvableRank}(tag)$ *protocol among all parties, faithfully emulating the behavior of all honest parties, and records the tag by updating $\mathcal{T} := \mathcal{T} \cup \{tag\}$. For every honest party $j \in \mathcal{H}$, let (rank_j, π_j) denote its output in this execution. The challenger records all such tuples by updating*
$$\mathcal{O} := \mathcal{O} \cup \{(j, tag, \mathsf{rank}_j, \pi_j)\}_{j \in \mathcal{H}}.$$
- ***Challenge phase I.*** *The challenger gives $\mathcal{A}$ the multiset $\mathcal{O}$. The adversary outputs a multiset*
$$\mathcal{P} := \{(j, tag, \mathsf{rank}_j, \pi_j)\}_{j \in \mathcal{C}},$$
where $\mathcal{C}$ denotes the set of corrupted parties. The challenger checks that for every $(j, tag, \mathsf{rank}_j, \pi_j) \in \mathcal{P}$,
$$\mathsf{RankVerify}(\mathsf{rank}_j, j, \pi_j, tag) = 1.$$
If any check fails, the experiment aborts.[16]
- ***Challenge phase II.*** *Let $\mathcal{S} := \mathcal{O} \cup \mathcal{P}$ and $k := |\mathcal{S}|$. Enumerate the elements of $\mathcal{S}$ in any fixed canonical order as*
$$\mathcal{S} = \{(j_\ell, tag_\ell, \mathsf{rank}_\ell, \pi_\ell)\}_{\ell \in [k]}.$$
Sample $(U_1, \ldots, U_k) \leftarrow \mathcal{D}^k$ uniformly and independently. Moreover, sample a bit $b \leftarrow \{0,1\}$ uniformly at random. If $b = 0$, give $(U_1, \ldots, U_k)$ to $\mathcal{B}$; otherwise, give $(\mathsf{rank}_1, \ldots, \mathsf{rank}_k)$ to $\mathcal{B}$.
- ***Winning condition.*** *The distinguisher $\mathcal{B}$ outputs a bit $b' \in \{0,1\}$. The experiment outputs 1 if and only if $b' = b$.*

[16] Alternatively, the challenger can proceed without aborting and set the final guess b' to a uniform bit whenever the verification checks above fail.

B Protocols from Prior Work (Section 4)

B.1 Non-Equivocation

We recall the protocol from [ABLS25]. The $\mathsf{NonEquiv}$ protocol proceeds in two logical rounds. In the first round, each party multicasts a "help" message containing its input m_i to all other parties. Upon receiving such a message, a party responds with a "sig" message containing a signature on the first "help" message it receives from any sender. Each party waits until it has collected $n - f$ valid "sig" messages corresponding to the same "help" message, and then outputs a proof attesting to the receipt of these signatures. Because any set of $n - f$ parties forms a quorum, any two such sets intersect in at least one honest party. Since an honest party signs at most one message per sender, this intersection property ensures non-equivocation.

A formal pseudocode description of the protocol is given in Figures 7 and 8. The protocol makes use of the non-interactive proof system $\mathsf{PS}_{\mathsf{TS}} = (\mathsf{PProve}_{\mathsf{TS}}, \mathsf{PVerify}_{\mathsf{TS}})$ for the relation

$$\mathcal{R}_{\mathsf{TS}} := \left\{ \left(\vec{\mathsf{pk}}, m; S, (\sigma_i)_{i \in S} \right) \;\middle|\; |S| = n - f,\ \forall i \in S : \mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, m, \sigma_i) = 1 \right\},$$

showing knowledge of $n - f$ valid signatures on a common message m under distinct public keys.

Algorithm 7 $\mathsf{NonEquiv}(m_i, tag)$

$\mathsf{sigs}_i \leftarrow \emptyset$
send $\langle$"help", $m_i, tag\rangle$ to all parties
// Sign received messages
upon receiving the first $\langle$"help", $m_i, tag\rangle$ message from j, **do**
    send $\langle$"sig", $\mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, (j, m_j, tag))\rangle$ to j
// Collect signatures, output a proof once $n - f$ have been received
upon receiving the first $\langle$"sig", $\sigma_j\rangle$ message from j, **do**
    if $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_j, (i, m_i, tag), \sigma_j) = 1$ **then**
        $\mathsf{sigs}_i \leftarrow \mathsf{sigs}_i \cup \{(j, \sigma_j)\}$
        if $|\mathsf{sigs}_i| = n - f$ **then**
            output $\mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (i, m_i, tag); \mathsf{sigs}_i)$, but continue sending messages

Algorithm 8 $\mathsf{NEVerify}(i, m, \pi, tag)$

return $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (i, m, (\text{"lockSig"}, tag)); \pi)$

B.2 PVSS Exchange

We recall the protocol from [ABLS25]. In their construction, each party uses a VRF that, on input a secret key and an index, outputs a selection bit together with a publicly verifiable proof. The VRF outputs *true* with probability $\frac{\lambda c}{n}$ for a fixed constant $c \geq 10$, and $false$ otherwise. Parties wait to receive messages from a sufficiently large fraction of the selected parties, namely a $(\frac{2}{3} - \epsilon)$-fraction of the expected number, where $\epsilon \in (0, \frac{1}{50})$ is an agreed-upon slack parameter. Upon receiving valid VRF proofs from $(\frac{2}{3} - \epsilon)\lambda c$ distinct parties, a party aggregates the corresponding PVSS transcripts and outputs the resulting aggregate together with a proof certifying that it contains a sufficient number of transcripts. The security intuition is that any sufficiently large randomly sampled subset of parties contains an honest participant except with negligible probability. For liveness, the authors show that for any fixed corruption pattern, the probability that no honest party terminates is negligible; moreover, even after taking a union bound over all possible choices of corrupted parties, at least one forever-honest party terminates with overwhelming probability.

A formal pseudocode description of the protocol is given in Figures 9 and 10. The protocol makes use of the non-interactive proof system $\mathsf{PS}_{\mathsf{exch}} = (\mathsf{PProve}_{\mathsf{exch}}, \mathsf{PVerify}_{\mathsf{exch}})$ for the relation

$$\mathcal{R}_{\mathsf{exch}} := \left\{ \begin{array}{l|l} (\vec{\mathsf{pk}}, \mathsf{trans}, j; & |S| = (\frac{2}{3} - \epsilon)\lambda c,\ \forall i \in S : \\ S, \{(\sigma_i, \mathsf{trans}_i, \pi_i)\}_{i \in S}) & \mathsf{Verify}_{\mathsf{PVSS}}(\mathsf{trans}_i, \vec{\mathsf{pk}}) = 1, \\ & \mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}_i, true, j, \pi_i) = 1, \\ & \mathsf{Verify}_{\mathsf{KDS}}(\mathsf{pk}_i, (j, \mathsf{trans}_i), \sigma_i) = 1 \end{array} \right\},$$

showing receipt of $(\frac{2}{3} - \epsilon)\lambda c$ valid PVSS transcripts from distinct parties, together with corresponding valid signatures and VRF proofs.

Algorithm 9 $\mathsf{PVSSExchange}(tag)$

$\mathsf{messages} \leftarrow \emptyset,\ \mathsf{transcripts} \leftarrow \emptyset$
// Check which parties to send messages to, generate transcripts, and store them
for $j \in [n]$ **do**
  $(\mathsf{sendFlag}_j, \pi_j) \leftarrow \mathsf{VRF}(\mathsf{sk}_i, (j, tag))$
  if $\mathsf{sendFlag}_j = true$ **then**
    $\mathsf{trans}_j \leftarrow\!\!\$\ \mathsf{Dist}_{\mathsf{PVSS}}(\vec{\mathsf{pk}})$
    $\mathsf{messages} \leftarrow \mathsf{messages} \cup \{(j, \mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \mathsf{Sign}_{\mathsf{KDS}}(\mathsf{sk}_i, (j, \mathsf{trans}_j)))\}$
// Erase keys and then send messages to all parties
update $\vec{\mathsf{sk}}_i$ via Update and **erase** the old key and randomness used for $\mathsf{Dist}_{\mathsf{PVSS}}()$
for all $(j, \mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \sigma_j) \in \mathsf{messages}$ **do**
  send $\langle$"trans", $\mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \sigma_j\rangle$ to j
// Collect transcripts and output an aggregated transcript after receiving enough of them
upon receiving a $\langle$"trans", $\mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \sigma_j\rangle$ message from j, **do**
  if $\mathsf{Verify}_{\mathsf{VRF}}(\mathsf{pk}_j, \mathsf{sendFlag}, (i, tag), \pi_j) = \mathsf{Verify}_{\mathsf{KDS}}(\mathsf{pk}_j, (i, \mathsf{trans}_j), \sigma_j) = 1$ and $\mathsf{sendFlag} = true$ **then**
    $\mathsf{transcripts} \leftarrow \mathsf{transcripts} \cup \{(j, \mathsf{sendFlag}, \pi_j, \mathsf{trans}_j, \sigma_j)\}$
    if $|\mathsf{transcripts}| \geq (\frac{2}{3} - \epsilon)\lambda c$ **then**
      $\mathsf{trans} \leftarrow \mathsf{Aggregate}_{\mathsf{PVSS}}(\{\mathsf{trans}_j \mid \exists (j, \mathsf{sendFlag}_j, \pi_j, \mathsf{trans}_j, \sigma_j) \in \mathsf{transcripts}\})$
      $\pi \leftarrow \mathsf{PProve}_{\mathsf{exch}}(\vec{\mathsf{pk}}, \mathsf{trans}, tag; \mathsf{transcripts})$
      output (trans, π) and **terminate**

Algorithm 10 $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi, tag)$

return $\mathsf{PVerify}_{\mathsf{exch}}(\vec{\mathsf{pk}}, \mathsf{trans}, tag; \pi)$

B.3 Provable AVID

We recall the protocol from [ABLS25]. Their protocol makes use of linear erasure codes [RS60] and cryptographic accumulators [BdM94]. We briefly recall these primitives below (see Section A for formal definitions).

Linear erasure codes. The authors employ standard (q, b) Reed–Solomon codes [RS60], which encode b data symbols into a codeword of q symbols via the Encode algorithm, with the property that any subset of b codeword symbols suffices to reconstruct the original data using the Decode algorithm. In our setting, we use parameters $(q, b) = (n, f + 1)$, which guarantee decodability in the presence of Byzantine faults while maintaining robustness.

Cryptographic accumulators. A cryptographic accumulator [BdM94] is a primitive that compresses a set of elements D into a single accumulation value z using the $\mathsf{Eval}_{\mathsf{Acc}}$ algorithm. For each element in D, the

accumulator supports the generation of a succinct membership witness via the $\mathsf{Witness}_{\mathsf{Acc}}$ algorithm, which can be efficiently verified using $\mathsf{Verify}_{\mathsf{Acc}}$. The standard security requirement for accumulators is collision resistance, ensuring that no adversary can produce a valid witness for an element that was not accumulated. The authors employ cryptographic accumulators with constant-size accumulation values and membership proofs [BBF19]. We note, however, that these can alternatively be instantiated using Merkle trees [Mer87], at the cost of an additional $\log(n)$ factor in communication complexity.

Construction. For their protocol, the authors assume access to an external validity predicate valid. They further employ the non-interactive proof system $\mathsf{PS}_{\mathsf{acc}} = (\mathsf{PProve}_{\mathsf{acc}}, \mathsf{PVerify}_{\mathsf{acc}})$ for the relation

$$\mathcal{R}_{\mathsf{acc}} := \left\{ \left(\vec{\mathsf{pk}}, \mathsf{acc}, j; m, \pi_{\mathsf{TS}}\right) \;\middle|\; \begin{array}{l} \mathsf{acc} = \mathsf{Eval}_{\mathsf{Acc}}(ak, \mathsf{Encode}(m; n, f+1)), \\ \mathsf{valid}(m) = 1,\ \mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (j, \mathsf{acc}); \pi_{\mathsf{TS}}) = 1 \end{array} \right\},$$

showing knowledge of $n - f$ valid signatures under distinct public keys on an accumulation value acc of an externally valid message m. Their provable AVID protocol proceeds in a dispersal phase followed by a retrieval phase. A formal pseudocode description of the protocol is given in Figures 11, 12, and 13.

In the dispersal phase, a designated sender $s \in [n]$ with input m first encodes m into n shares $(m'_1, \ldots, m'_n)$ using an $(n, f+1)$-erasure code. The sender accumulates the resulting codeword into an accumulation value acc and collects $n - f$ signatures on acc to obtain a non-equivocation proof π_{TS}. It additionally produces a proof $\pi_{\mathsf{acc},s}$ showing that acc corresponds to an encoding of a message m with $\mathsf{valid}(m) = 1$. The sender then distributes to each party i the tuple $(\mathsf{acc}, \pi_{\mathsf{acc},s}, m'_i, w_i)$, where w_i is a membership witness for m'_i with respect to acc. Upon receiving a valid tuple, each party returns a signature on the sender's index s. The sender aggregates $n - f$ such signatures into a final proof of dispersal π_{tot} and terminates. Verification $\mathsf{Disperse}_{\mathsf{Verify}}$ consists of checking that π_{tot} is a valid proof of dispersal. Then, in the retrieval phase, each party multicasts its share (m'_i, w_i). Upon receiving $f+1$ valid shares consistent with the same accumulation value acc and dispersal proof π_{tot}, a party reconstructs m using the decoding algorithm Decode.

Algorithm 11 $\mathsf{Disperse}(m)$

$\mathsf{sigs} \leftarrow \emptyset,\ \mathsf{sigs}' \leftarrow \emptyset,\ \pi_{tot} \leftarrow \bot,\ M' \leftarrow \bot$
$m_i \leftarrow \bot,\ w_i \leftarrow \bot,\ \mathsf{acc}_i \leftarrow \bot,\ \pi_{acc,i} \leftarrow \bot$
// Lines 3-7 are only run by the sender s
// Encode the message in an error-correcting code and compute an accumulator of the encoding
$M' \leftarrow (m'_1, \ldots, m'_n) \leftarrow \mathsf{Encode}(m; n, f+1)$
$\mathsf{acc} \leftarrow \mathsf{Eval}_{\mathsf{Acc}}(ak, M')$
for $i \in [n]$ **do**
    $w_i \leftarrow \mathsf{Witness}_{\mathsf{Acc}}(ak, \mathsf{acc}, m'_i, M')$
// Non-equivocation of accumulator
send $\langle$"help", $\mathsf{acc}, \pi\rangle$ to all parties
upon receiving the first $\langle$"help", $\mathsf{acc}\rangle$ message from the sender s, **do**
    send $\langle$"sig", $\mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, (s, \mathsf{acc}))\rangle$ to s
upon the sender s receiving the first $\langle$"sig", $\sigma_i\rangle$ message from i, **do**
    if $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, (s, \mathsf{acc})) = 1$ **then**
        $\mathsf{sigs} \leftarrow \mathsf{sigs} \cup \{(i, \sigma_i)\}$
        if $|\mathsf{sigs}| = n - f$ **then**
            $\pi_{\mathsf{TS}} \leftarrow \mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (s, \mathsf{acc}); \mathsf{sigs})$
            $\pi_{acc,s} \leftarrow \mathsf{PProve}_{\mathsf{acc}}(\vec{\mathsf{pk}}, (s, \mathsf{acc}); m, \pi_{\mathsf{TS}})$
// Send shares of the encoding to each party, parties reply once they receive their shares
upon the sender s having $\pi_{acc,s} \neq \bot$, **do**
    for $i \in [n]$ **do**
        send $\langle$"share", $\mathsf{acc}_s, \pi_{acc,s}, m', w\rangle$ to i
upon receiving the first $\langle$"share", $\mathsf{acc}, \pi_{acc}, m'_i, w_i\rangle$ message from j, **do**
    if $\mathsf{PVerify}_{\mathsf{acc}}(\vec{\mathsf{pk}}, (j, \mathsf{acc}); \pi_{acc}) = 1$ and $\mathsf{Verify}_{\mathsf{Acc}}(ak, \mathsf{acc}, m', w) = 1$ **then**
        $m'_i \leftarrow m',\ w'_i \leftarrow w,\ \mathsf{acc}_i \leftarrow \mathsf{acc},\ \pi_{\mathsf{acc},i} \leftarrow \pi_{\mathsf{acc}}$
        send $\langle$"total", $\mathsf{Sign}_{\mathsf{DS}}(\mathsf{sk}_i, j)\rangle$ to j
// Wait until $n - f$ parties received their shares, and output a proof
upon the sender s receiving the first $\langle$"total", $\sigma_i\rangle$ message from i, **do**
    if $\mathsf{Verify}_{\mathsf{DS}}(\mathsf{pk}_i, i) = 1$ **then**
        $\mathsf{sigs}' \leftarrow \mathsf{sigs}' \cup \{(i, \sigma_i)\}$
        if $|\mathsf{sigs}'| = n - f$ **then**
            $\pi_{tot} \leftarrow \mathsf{PProve}_{\mathsf{TS}}(\vec{\mathsf{pk}}, i; \mathsf{sigs}')$
            output π_{tot} and **terminate**

Algorithm 12 $\mathsf{Retrieve}()$

$\mathsf{shares} \leftarrow \emptyset,\ m \leftarrow \bot$
send $\langle$"retrieve", $\mathsf{acc}_i, \pi_{\mathsf{acc},i}, m'_i, w_i\rangle$ to all parties
upon receiving the first $\langle$"retrieve", $\mathsf{acc}, \pi_{\mathsf{acc}}, m'_i, w_i\rangle$ message, **do**
    if $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (s, \mathsf{acc}_i); \pi_{\mathsf{acc},i}) = 1$ and $\mathsf{Verify}_{\mathsf{Acc}}(ak, \mathsf{acc}, m'_i, w_i) = 1$ **then**
        $\mathsf{shares} \leftarrow \mathsf{shares} \cup \{(i, m'_i)\}$
        if $|\mathsf{shares}| = f + 1$ **then**
            $m \leftarrow \mathsf{Decode}(\mathsf{shares}; n, f+1)$
            output m

Algorithm 13 $\mathsf{DisperseVerify}(i, \pi)$

return $\mathsf{PVerify}_{\mathsf{TS}}(\vec{\mathsf{pk}}, (i, \text{"total"}); \pi)$

C Deferred Proofs

C.1 Provable Rank (Section 5)

Henceforth, we denote by $|\pi|$ the (maximum) size of our underlying zkSNARKs. While our zkSNARKs are instantiated with proof size $O(\lambda)$ [Fu25, AGL+23], our protocols admit alternative instantiations provided that the proof size remains $O(\lambda^2)$. Our complexity analysis is stated in terms of the proof length $|\pi|$ to keep the presentation generic and to accommodate such alternative instantiations.

Theorem 4 (Complexity)**.** *The* ProvableRank *protocol has $O((\lambda^2 + |\pi|)n^2)$ communication complexity. In particular, when $|\pi| \in O(\lambda^2)$, this is $O(\lambda^2 n^2)$. In addition, the protocol has $O(1)$ round complexity.*

Proof. During the protocol, parties do the following:

- call the PVSSExchange costing $O(\lambda^2 n^2)$;
- each disperses a transcript and privately retrieves a single transcript for a total cost of $O(\lambda n^2 + n \cdot (\lambda n))$;
- call NE with a secret key of size $O(\lambda)$ as input for a cost of $O(\lambda n^2)$;
- send each other a private key with a share, each of size $O(\lambda)$, as well as a constant number of proofs of size $O(|\pi|)$, for a total of $O(|\pi| n^2)$; and,
- send each other decrypted shares of size $O(\lambda)$ for a total of $O(\lambda n^2)$.

In total, the communication costs add up to $O((\lambda^2 + |\pi|)n^2)$. In particular, when $|\pi| \in O(\lambda^2)$, this is $O(\lambda^2 n^2)$.

In addition, the protocol consists of a constant number of all-to-all rounds and constant-round protocols, yielding $O(1)$ round complexity. □

Theorem 5 (Security)**.** *The pair* (ProvableRank, RankVerify) *(Algorithms 1 and 2) is a Provable Rank protocol resilient to $f < n/3$ Byzantine parties, except with negligible probability in λ.*

Proof. We prove each property in a separate lemma and explicitly state the required properties of the underlying primitives. Although all primitives in our construction satisfy these requirements, this modular presentation highlights which assumptions are needed for each guarantee and facilitates replacing components with alternative implementations satisfying the same properties.

Lemma 1 (Correctness)**.** *Assume that: (i)* PVSSExchange *is verifiable, (ii)* Disperse *satisfies external validity with respect to the predicate* ExchangeVerify*, (iii) the non-equivocation protocol* NE *is valid, (iv) the proof systems $\mathsf{PS}_{\mathsf{key}}$ and $\mathsf{PS}_{\mathsf{coin}}$ are complete, and (v) the underlying threshold signature scheme is correct.*

Then for any tag and any honest party i, if i outputs $(\mathsf{rank}_i, \pi_{\mathsf{rank},i})$ in an execution of ProvableRank(tag)*, it holds that* RankVerify$(\mathsf{rank}_i, i, \pi_{\mathsf{rank},i}, tag) = 1$*, except with negligible probability in λ. In particular, the protocol pair satisfies the correctness property.*

Proof. Assume that an honest party i completes the protocol ProvableRank(tag) and outputs $(\mathsf{rank}_i, \pi_{\mathsf{rank},i})$, where

$$\pi_{\mathsf{rank},i} = (\mathsf{eph_vk}_i, \pi_{\mathsf{NE},i}, \pi_{\mathsf{coin},i}, \pi_{\mathsf{trans},i}, \pi_{\mathsf{key},i}).$$

We show that RankVerify$(\mathsf{rank}_i, i, \pi_{\mathsf{rank},i}, tag) = 1$.

Party i invokes the non-equivocation protocol NE only after setting $\mathsf{trans}_i \neq \perp$ (line 18–20). The variable trans_i is assigned either upon termination of protocol PVSSExchange(tag) (lines 4–6, line 5) or upon successful completion of PrivateRetrieve(i) (lines 15–18, line 17). In both cases, the associated proof $\pi_{\mathsf{trans},i}$ satisfies

$$\mathsf{ExchangeVerify}(\mathsf{trans}_i, tag, \pi_{\mathsf{trans},i}) = 1,$$

by verifiability of the PVSSExchange protocol and the external validity guarantee of the Disperse protocol.

After setting trans_i, party i defines $\mathsf{eph_vk}_i := \mathsf{trans}_i[0]$ and invokes $\mathsf{NE}(\mathsf{eph_vk}_i, (\text{“flip”}, tag))$, storing the resulting proof $\pi_{\mathsf{NE},i}$. By validity of the non-equivocation protocol,

$$\mathsf{NEVerify}(i, \mathsf{eph_vk}_i, (\text{“flip”}, tag), \pi_{\mathsf{NE},i}) = 1.$$

Party i then computes

$$\pi_{\mathsf{key},i} \leftarrow \mathsf{PProve}_{\mathsf{key}}((\mathsf{eph_vk}_i, \pi_{\mathsf{trans},i}, tag), \mathsf{trans}_i).$$

Since $\mathsf{eph_vk}_i = \mathsf{trans}_i[0]$ and $\mathsf{ExchangeVerify}(\mathsf{trans}_i, tag, \pi_{\mathsf{trans},i}) = 1$, completeness of the proof system $\mathsf{PS}_{\mathsf{key}}$ implies

$$\mathsf{PVerify}_{\mathsf{key}}((\mathsf{eph_vk}_i, \pi_{\mathsf{trans},i}, tag), \pi_{\mathsf{key},i}) = 1.$$

In the final phase, party i collects partial signatures $\{(j, \sigma_j)\}_j$ and adds a tuple (j, σ_j) to $\mathsf{partials}_i$ only if

$$\mathsf{VerifyPartialCoin}(\mathsf{trans}_i[j], (i, tag), \sigma_j) = 1.$$

Once $|\mathsf{partials}_i| = f + 1$, party i reconstructs a threshold signature

$$\sigma = (\sigma', \pi) \leftarrow \mathsf{ReconstructCoin}(\mathsf{partials}_i).$$

By correctness of the threshold signature scheme, we have

$$\mathsf{VerifyCoin}(\mathsf{eph_vk}_i, (i, tag), (\sigma', \pi)) = 1.$$

Party i then sets $\mathsf{rank}_i := \mathsf{Hash}(\sigma')$ and computes a proof

$$\pi_{\mathsf{coin},i} \leftarrow \mathsf{PProve}_{\mathsf{coin}}((\mathsf{eph_vk}_i, \mathsf{rank}_i, i, tag), (\sigma', \pi)).$$

By completeness of the proof system $\mathsf{PS}_{\mathsf{coin}}$, we have

$$\mathsf{PVerify}_{\mathsf{coin}}((\mathsf{eph_vk}_i, \mathsf{rank}_i, i, tag), \pi_{\mathsf{coin},i}) = 1.$$

Finally, the verifier $\mathsf{RankVerify}$ checks that the conjunction of $\mathsf{NEVerify}$, $\mathsf{PVerify}_{\mathsf{key}}$, and $\mathsf{PVerify}_{\mathsf{coin}}$ holds (cf. Algorithm 2). Since all three checks succeed, it follows that

$$\mathsf{RankVerify}(\mathsf{rank}_i, i, \pi_{\mathsf{rank},i}, tag) = 1,$$

which completes the proof. □

Lemma 2 (Uniqueness)**.** *Assume that (i) the non-equivocation protocol* NE *satisfies non-equivocation, (ii) the proof system* $\mathsf{PS}_{\mathsf{coin}}$ *is knowledge-sound with extractor* $\mathsf{Ext}_{\mathsf{coin}}$*, and (iii) the underlying threshold signature scheme has a unique signature component* σ' *per public key and message.*

Then the uniqueness security experiment in Definition 19 is won with at most negligible probability in λ*. In particular, the protocol pair satisfies the uniqueness property.*

Proof. Suppose, for contradiction, that there exists an efficient adversary $\mathcal{A}$ that wins the experiment in Definition 19 with non-negligible probability. Thus, with non-negligible probability, $\mathcal{A}$ outputs two tuples

$$(\mathsf{rank}, i, \pi, tag) \quad \text{and} \quad (\mathsf{rank}', i, \pi', tag)$$

such that $\mathsf{rank} \neq \mathsf{rank}'$ and

$$\mathsf{RankVerify}(\mathsf{rank}, i, \pi, tag) = 1 \quad \text{and} \quad \mathsf{RankVerify}(\mathsf{rank}', i, \pi', tag) = 1.$$

Parse

$$\pi = (\mathsf{eph_vk}, \pi_{\mathsf{NE}}, \pi_{\mathsf{coin}}, \pi_{\mathsf{trans}}, \pi_{\mathsf{key}}) \quad \text{and} \quad \pi' = (\mathsf{eph_vk}', \pi'_{\mathsf{NE}}, \pi'_{\mathsf{coin}}, \pi'_{\mathsf{trans}}, \pi'_{\mathsf{key}}).$$

Step 1: The verification keys must coincide. Since both tuples verify, in particular

$$\mathsf{NEVerify}(i, \mathsf{eph_vk}, (\text{“flip”}, tag), \pi_{\mathsf{NE}}) = 1$$
$$\text{and} \quad \mathsf{NEVerify}(i, \mathsf{eph_vk}', (\text{“flip”}, tag), \pi'_{\mathsf{NE}}) = 1.$$

By the non-equivocation property of NE, it follows that $\mathsf{eph_vk} = \mathsf{eph_vk}'$.

Step 2: Extract verifying threshold signatures. Since both tuples verify, we also have

$$\begin{aligned} &\mathsf{PVerify}_{\mathsf{coin}}((\mathsf{eph_vk}, \mathsf{rank}, i, tag), \pi_{\mathsf{coin}}) = 1 \\ \text{and} \quad &\mathsf{PVerify}_{\mathsf{coin}}((\mathsf{eph_vk}', \mathsf{rank}', i, tag), \pi'_{\mathsf{coin}}) = 1. \end{aligned}$$

By knowledge soundness of $\mathsf{PS}_{\mathsf{coin}}$, there exists an efficient extractor $\mathsf{Ext}_{\mathsf{coin}}$ that, given the internal randomness of the prover used to produce π_{coin}, outputs a witness (σ', π^*) such that

$$\mathsf{VerifyCoin}(\mathsf{eph_vk}, (i, tag), (\sigma', \pi^*)) = 1 \quad \text{and} \quad \mathsf{rank} = \mathsf{Hash}(\sigma').$$

Analogously, from π'_{coin} the extractor outputs $(\sigma'', \pi^{*\prime})$ such that

$$\mathsf{VerifyCoin}(\mathsf{eph_vk}', (i, tag), (\sigma'', \pi^{*\prime})) = 1 \quad \text{and} \quad \mathsf{rank}' = \mathsf{Hash}(\sigma'').$$

Using Step 1, we may replace $\mathsf{eph_vk}' = \mathsf{eph_vk}$ in the second verification equation.

Step 3: Uniqueness of the signature component implies equal ranks. By uniqueness of the threshold signature component, for fixed public key $\mathsf{eph_vk}$ and message (i, tag), there exists at most one σ' such that $\mathsf{VerifyCoin}(\mathsf{eph_vk}, (i, tag), (\sigma', \cdot)) = 1$. Therefore, $\sigma' = \sigma''$, and we get

$$\mathsf{rank} = \mathsf{Hash}(\sigma') = \mathsf{Hash}(\sigma'') = \mathsf{rank}',$$

contradicting the winning condition $\mathsf{rank} \neq \mathsf{rank}'$. □

Lemma 3 (Secrecy)**.** *Assume that: (i)* Disperse *satisfies external validity with respect to the predicate* ExchangeVerify*, (ii)* PVSSExchange *is verifiable, (iii) the underlying threshold signature scheme is unforgeable and has a unique signature component σ' per public key and message, (iv) the proof system $\mathsf{PS}_{\mathsf{coin}}$ is zero-knowledge, and (v)* Hash *is modeled as a random oracle.*

Then the secrecy security experiment in Definition 18 is won by any efficient adversary with at most negligible probability in λ. In particular, the protocol pair satisfies the secrecy property.

Proof. Fix any PPT adversary $\mathcal{A}$ corrupting at most f parties. We prove that $\mathcal{A}$ wins the secrecy experiment of Definition 18 with probability at most $\frac{1}{2} + \mathsf{negl}(\lambda)$ via a sequence of games.

Throughout, let $Q_{\mathsf{Hash}}(\lambda)$ be an a priori polynomial upper bound on the number of queries $\mathcal{A}$ makes to the random oracle Hash, and let $Q_{\mathrm{exec}}(\lambda)$ be a polynomial upper bound on the number of distinct tags on which $\mathcal{A}$ invokes ProvableRank$(\cdot)$ (equivalently, the number of protocol executions created by the oracle in the experiment).

Game $\mathbf{G}_0$ (real secrecy game). This is the secrecy experiment in Definition 18. In particular, after $\mathcal{A}$ outputs a challenge pair (i, tag^*) with i honest and $tag^* \in \mathcal{T}$, the challenger samples $b \leftarrow \{0, 1\}$ and sets

$$(R, \Pi) := \begin{cases} (\mathsf{rank}_i, \pi_i) & \text{if } b = 0, \\ (\tilde{r}, \tilde{\pi}) \text{ s.t. } \tilde{r} \leftarrow \mathcal{D},\ \mathsf{RankVerify}(\tilde{r}, i, \tilde{\pi}, tag^*) = 1 & \text{if } b = 1. \end{cases}$$

It then gives $\mathcal{A}$ the tuple

$$\Big(R, \Pi, \{(\mathsf{rank}_j, \pi_j)\}_{j \neq i \in \mathcal{H}}\Big),$$

and $\mathcal{A}$ outputs a bit b'. The game outputs 1 if and only if $b' = b$.

Game $\mathbf{G}_1$ (guess the challenged party). The challenger samples an index $i^* \leftarrow [n]$ uniformly at random and aborts unless $\mathcal{A}$'s challenge index equals i^*, i.e., unless $\mathcal{A}$ outputs (i, tag^*) with $i = i^*$. Let Good_1 denote the event that $\mathcal{A}$ indeed chooses $i = i^*$. Then $\Pr[\mathsf{Good}_1] = 1/n$, and conditioned on Good_1 the view of $\mathcal{A}$ is distributed exactly as in $\mathbf{G}_0$. We therefore get

$$\Pr[\mathbf{G}_1 \Rightarrow 1] \geq \frac{1}{n} \cdot \Pr[\mathbf{G}_0 \Rightarrow 1],$$

where $\Pr[\mathbf{G}_i \Rightarrow 1]$ denotes the probability that the game returns 1, i.e., the adversary wins.

Game $\mathbf{G}_2$ (guess the challenged execution/tag). The challenger samples an index $t^* \leftarrow [Q_{\text{exec}}]$ uniformly at random, intended to identify which oracle call $\textsc{ProvableRank}(\cdot)$ will correspond to the challenge tag tag^*. Concretely, the challenger maintains the ordered list of tags $(tag^{(1)}, \ldots, tag^{(q)})$ as oracle calls arrive; it aborts unless the challenge tag equals $tag^{(t^*)}$, i.e., unless $\mathcal{A}$ outputs $tag^* = tag^{(t^*)}$. Let Good_2 be the event that this guess is correct. Then $\Pr[\mathsf{Good}_2] \geq 1/Q_{\text{exec}}$, and conditioned on Good_2 the distribution is as in $\mathbf{G}_1$. Therefore, we get

$$\Pr[\mathbf{G}_2 \Rightarrow 1] \geq \frac{1}{Q_{\text{exec}}} \cdot \Pr[\mathbf{G}_1 \Rightarrow 1],$$

Game $\mathbf{G}_3$ (guess the random oracle query defining the rank). The challenger samples an index $q^* \leftarrow [Q_{\mathsf{Hash}}]$ uniformly at random. It answers Hash queries by lazy sampling as usual, while recording the sequence of *inputs* queried to Hash as $(z_1, z_2, \ldots)$, where z_q denotes the q-th input queried by *any* party (including the adversary and all simulated honest parties). The challenger aborts unless the rank output by the challenged honest party i^* is computed as $\mathsf{rank}_{i^*} = \mathsf{Hash}(z_{q^*})$ in the execution under tag^*. Let Good_3 denote the event that this guess is correct, and define

$$\sigma'^* := z_{q^*} \quad \text{so that} \quad \mathsf{rank}_{i^*} = \mathsf{Hash}(\sigma'^*).$$

Since Hash is queried at most Q_{Hash} times, we have $\Pr[\mathsf{Good}_3] \geq 1/Q_{\mathsf{Hash}}$, and conditioned on Good_3 the distribution is as in $\mathbf{G}_2$. Thus, we obtain

$$\Pr[\mathbf{G}_3 \Rightarrow 1] \geq \frac{1}{Q_{\mathsf{Hash}}} \cdot \Pr[\mathbf{G}_2 \Rightarrow 1],$$

Game $\mathbf{G}_4$ (program random oracle at σ'^*). We modify the simulation of the random oracle Hash as follows: the first time the honest party i^* queries $\mathsf{Hash}(\sigma'^*)$ in order to compute its rank in the execution under tag^*, the challenger samples $\tilde{r} \leftarrow \mathcal{D}$ uniformly and sets $\mathsf{Hash}(\sigma'^*) := \tilde{r}$. All subsequent queries to $\mathsf{Hash}(\sigma'^*)$ are answered consistently with $\tilde{r}$, and all other Hash queries are answered by lazy sampling as usual.

Let Bad be the event that $\mathcal{A}$ queries $\mathsf{Hash}(\sigma'^*)$ *before* the challenger programs it (i.e., before the first query to $\mathsf{Hash}(\sigma'^*)$ made by the honest party i^* when computing its rank in the execution under tag^*). Conditioned on $\neg\mathsf{Bad}$, the change from $\mathbf{G}_3$ to $\mathbf{G}_4$ is consistent with the random oracle distribution, and thus

$$\Pr[\mathbf{G}_4 \Rightarrow 1 \mid \neg\mathsf{Bad}] = \Pr[\mathbf{G}_3 \Rightarrow 1 \mid \neg\mathsf{Bad}].$$

Therefore, we get

$$|\Pr[\mathbf{G}_4 \Rightarrow 1] - \Pr[\mathbf{G}_3 \Rightarrow 1]| \leq \Pr[\mathsf{Bad}].$$

Bounding $\Pr[\mathsf{Bad}]$ *via threshold signature unforgeability.* We construct a PPT forger $\mathcal{F}$ against the (strong) unforgeability of the underlying threshold signature scheme, where unforgeability refers to the computational hardness of producing a signature component σ' for a fresh message under a verification key derived from an aggregated transcript containing at least one honest contribution. Specifically, we describe the forger $\mathcal{F}$ that simulates $\mathbf{G}_3$ to the adversary $\mathcal{A}$ while interacting with the threshold signature unforgeability challenger $\mathcal{U}$.

Access to challenger $\mathcal{U}$. The challenger $\mathcal{U}$ runs the setup for the plain PKI and the underlying threshold signature scheme. In particular, it fixes keys for all parties and supports a corruption oracle $\textsc{Corrupt}(j)$ that reveals the secret state of party j upon corruption. The forger $\mathcal{F}$ must output a triple $(\mathsf{eph_vk}^*, m^*, \sigma'^*)$ that constitutes a successful break according to the unforgeability definition (i.e., σ'^* is the signature component associated with $(\mathsf{eph_vk}^*, m^*)$).

More initial guesses. In addition to the guesses in $\mathbf{G}_3$, $\mathcal{F}$ makes two further uniform guesses:

- an index $j^* \leftarrow [n]$, intended to be an honest contributor that appears in the aggregated transcript trans^* eventually adopted by the challenge party i^*; and

- an index $a^* \leftarrow [n]$, intended to be the (first) party whose PVSS Exchange output is dispersed and later adopted as the aggregated transcript trans^* by i^*.

Let Good_4 denote the event that j^* is an honest contributor contained in trans^*, and let Good_5 denote the event that a^* is the party whose dispersed transcript is the one ultimately adopted by i^*. Each event holds with probability at least $1/n$.

Simulating the plain PKI and corruption. $\mathcal{F}$ gives $\mathcal{A}$ the public parameters obtained from $\mathcal{U}$, i.e., the setup for the plain PKI and threshold signature scheme. For each honest party $j \in \mathcal{H}$, $\mathcal{F}$ sets the public PKI key used by the protocol to be the public key provided by $\mathcal{U}$. For each corrupt party $j \in \mathcal{C}$, $\mathcal{F}$ forwards to $\mathcal{U}$ the public key chosen by $\mathcal{A}$ for j during the PKI setup phase. In this way, all public keys used in the simulation are exactly those known to $\mathcal{U}$, ensuring that the simulated protocol execution is consistent with the threshold signature unforgeability experiment. Moreover, whenever $\mathcal{A}$ corrupts a party j, $\mathcal{F}$ forwards this request to $\mathcal{U}$ via $\textsc{Corrupt}(j)$, obtains the corresponding secret state (including its plain PKI secret key), and continues the simulation consistently. This ensures that the simulation is perfect from $\mathcal{A}$'s viewpoint.

Simulating $\mathsf{PVSSExchange}$ and forcing an honest contribution. $\mathcal{F}$ simulates each execution of the protocol $\mathsf{PVSSExchange}(tag)$ honestly for all parties, with one exception: for the guessed party a^* and the challenge tag tag^*, the forger $\mathcal{F}$ ensures that the aggregated transcript trans^* ultimately adopted by party i^* contains a contribution from the guessed honest party j^*. Concretely, $\mathcal{F}$ proceeds as follows.

In the threshold signature unforgeability experiment, the challenger $\mathcal{U}$ provides an honestly generated contribution transcript trans_{j^*} for party j^*. By definition of the unforgeability game, the adversary may specify any aggregated transcript trans as long as it includes trans_{j^*} as one of its components. Accordingly, in the simulation of $\mathsf{PVSSExchange}(tag^*)$, the forger $\mathcal{F}$ adopts trans_{j^*} as the honest contribution of party j^* and arranges that the guessed party a^* includes trans_{j^*} in its aggregated transcript. Conditioned on the event that a^* is indeed the party whose dispersed transcript is later adopted by the honest party i^*, the resulting aggregated transcript trans^* used by i^* contains trans_{j^*} as an honest contribution.

By the guarantees of PVSS Exchange and dispersal/retrieval, the transcript adopted by an honest party contains at least one honest contribution; conditioned on the above guessing events, this contribution is exactly trans_{j^*}. After completion of Phase 1 of $\mathsf{ProvableRank}(tag^*)$, the transcript trans^* is therefore fixed and consistent with both the $\mathsf{ProvableRank}$ protocol execution and the requirements of the threshold signature unforgeability experiment.

Simulating Phase 2 and Phase 3 and handling partial signatures. The forger $\mathcal{F}$ simulates the second phase of $\mathsf{ProvableRank}(\cdot)$ faithfully, including the non-equivocation protocol and all proof system messages. That is, it derives the transcript verification keys, collects a non-equivocation proof, sends each party its transcript share along with proofs. This is done faithfully on behalf of all honest parties. Note that this phase does not involve the threshold signature yet; it simply distributes the transcripts among parties. Moreover, it answers all queries to the random oracle Hash as in $\mathbf{G}_3$, while recording the queried inputs $(z_1, z_2, \ldots)$. We now describe how $\mathcal{F}$ simulates the third phase, i.e., the generation of partial threshold signatures for *all* transcript verification keys arising in the execution. Note that the partial signatures are used so that each party can derive its local individual rank. Before we do this, we first introduce some notation and recall some useful facts related to aggregated PVSS transcripts.

Aggregated transcripts and secret keys. Let trans be any aggregated transcript that appears in the simulation, and let $\mathsf{eph_vk} := \mathsf{trans}[0]$ be its verification key. We write $\mathsf{eph_sk}(\mathsf{trans})$ for the secret key defined by trans (i.e., the secret encoded by the aggregated transcript). In the unforgeability experiment, $\mathcal{U}$ provides $\mathcal{F}$ with an honestly generated contribution transcript trans_{j^*} for the guessed honest party j^*; we denote by $\mathsf{eph_sk}_{j^*}$ the (unknown) secret key encoded by trans_{j^*}.

Extracting contributions from an aggregated transcript. By design, any aggregated transcript trans contains (i) auxiliary data that identifies the set of included contributions and (ii) proofs of knowledge that allow extracting the secret-key contributions of corrupted parties from trans. Moreover, for every honest party $h \neq j^*$, recall that $\mathcal{F}$ generated h's individual contribution/transcript honestly during the PVSS Exchange simulation and therefore knows its corresponding secret-key contribution.

Handling transcripts/keys other than trans^*. Consider any aggregated transcript $\mathsf{trans} \neq \mathsf{trans}^*$ with verification key $\mathsf{eph_vk} = \mathsf{trans}[0]$ that appears during the simulation. There are two cases.

- **Case (i): trans does *not* include trans_{j^*}.** Then all honest contributions contained in trans were generated by $\mathcal{F}$ itself, and all corrupted party contributions can be extracted from trans using the auxiliary data and proofs of knowledge, as discussed above. Hence $\mathcal{F}$ can reconstruct the entire secret key $\mathsf{eph_sk}(\mathsf{trans})$ explicitly. In particular, it can generate partial signatures on any message for these transcripts/keys.
- **Case (ii): trans *does* include trans_{j^*}.** Then $\mathcal{F}$ knows all contributions to trans except the (unknown) secret contribution corresponding to trans_{j^*}. Therefore, $\mathcal{F}$ can compute a deterministic value $\mathsf{shift}(\mathsf{trans})$ such that
$$\mathsf{eph_sk}(\mathsf{trans}) \;=\; \mathsf{eph_sk}_{j^*} \;+\; \mathsf{shift}(\mathsf{trans}).$$
Here $\mathsf{shift}(\mathsf{trans})$ is obtained by summing all known honest contributions (for parties $h \neq j^*$) together with the extracted corrupted party contributions, as prescribed by the aggregation rule of PVSS.

Computing a shift relative to the challenge key $\mathsf{eph_vk}^* = \mathsf{trans}[0]$. Apply the same procedure to the challenge transcript trans^* as discussed in Case (ii) above. Since trans^* contains trans_{j^*} by construction, we obtain $\mathsf{shift}(\mathsf{trans}^*)$ such that
$$\mathsf{eph_sk}^* \;=\; \mathsf{eph_sk}(\mathsf{trans}^*) \;=\; \mathsf{eph_sk}_{j^*} \;+\; \mathsf{shift}(\mathsf{trans}^*).$$
Now consider any other transcript $\mathsf{trans} \neq \mathsf{trans}^*$ of Case (ii). Define the relative offset
$$\widetilde{\mathsf{shift}}(\mathsf{trans}) \;:=\; \mathsf{shift}(\mathsf{trans}) \;-\; \mathsf{shift}(\mathsf{trans}^*).$$
Then, we have
$$\begin{aligned}\mathsf{eph_sk}(\mathsf{trans}) \;&=\; \mathsf{eph_sk}_{j^*} + \mathsf{shift}(\mathsf{trans})\\ &=\; (\mathsf{eph_sk}_{j^*} + \mathsf{shift}(\mathsf{trans}^*)) + \widetilde{\mathsf{shift}}(\mathsf{trans})\\ &=\; \mathsf{eph_sk}^* + \widetilde{\mathsf{shift}}(\mathsf{trans}).\end{aligned}$$

Translating partial signatures. We use the linearity of the underlying threshold signature scheme in the secret key: given a partial signature under secret key $\mathsf{eph_sk}^*$ on message m, and a known offset Δ (as above), one can efficiently derive a corresponding partial signature under secret key $\mathsf{eph_sk}^* + \Delta$ on the same message m.[17]

Accordingly, to simulate partial signatures under a transcript key $\mathsf{eph_vk} = \mathsf{trans}[0]$:

- if trans falls under Case (i), $\mathcal{F}$ directly computes partial signatures using the explicitly known $\mathsf{eph_sk}(\mathsf{trans})$; and
- if trans falls under Case (ii), $\mathcal{F}$ obtains partial signatures under the challenge key $\mathsf{eph_sk}^*$ from $\mathcal{U}$ (as permitted by the unforgeability experiment) and translates them using $\widetilde{\mathsf{shift}}(\mathsf{trans})$ into partial signatures under $\mathsf{eph_sk}(\mathsf{trans})$.

This allows $\mathcal{F}$ to simulate the Phase 3 partial-signature messages—and hence all ranks—for all parties and tags other than the challenge pair (i^*, tag^*).

Handling partial signatures for (i^*, tag^*). $\mathcal{F}$ must simulate the phase in which parties send partial signatures on (i^*, tag^*) to i^* over private channels. For any honest party whose secret state is not yet known to $\mathcal{F}$ (because it has not been corrupted), $\mathcal{F}$ sends a syntactically correct dummy private-channel message of the appropriate form to i^*. If such a party is later corrupted, $\mathcal{F}$ forwards the corruption request to $\mathcal{U}$, obtains the necessary secret state, and retroactively explains the previously sent dummy message by fixing randomness consistent with the private channel transcript. This is possible under the private channel assumption and preserves a faithful simulation from $\mathcal{A}$'s point of view.

[17] This is the standard linearity property of pairing-based threshold signatures; the translation is deterministic given Δ.

Finally, we note that $\mathcal{F}$ does not need to explicitly generate the proof π_{coin,i^*} corresponding to rank_{i^*} in the real execution. Indeed, the event Bad occurs only if the adversary queries $\mathsf{Hash}(\sigma'^*)$ *before* the honest party i^* reconstructs its full threshold signature. Once i^* reconstructs the signature, it deterministically queries $\mathsf{Hash}(\sigma'^*)$ itself, and hence no distinguishing power is gained by considering later queries. In any case, even if required, $\mathcal{F}$ could generate π_{coin,i^*} using the zero-knowledge simulator for the proof system $\mathsf{PS}_{\mathsf{coin}}$, given only the public statement $(\mathsf{eph_vk}^*, \mathsf{rank}_{i^*}, i^*, tag^*)$.

All other proofs appearing in the simulation are generated honestly. This is because $\mathcal{F}$ either explicitly knows the corresponding secret keys (for transcripts of Case (i)) or can derive the required partial signatures and ranks via key-translation from $\mathsf{eph_sk}^*$ (for transcripts of Case (ii)), as described above. Consequently, the simulated protocol execution is indistinguishable from a real execution from the adversary's point of view, conditioned on all guesses Good_i were correct.

Using Bad to break unforgeability. Recall that in $\mathbf{G}_3$ we defined $\sigma'^* := z_{q^*}$ and that Good_3 means that $\mathsf{rank}_{i^*} = \mathsf{Hash}(\sigma'^*)$ in the challenge execution. If Bad occurs, then $\mathcal{A}$ queries $\mathsf{Hash}(\sigma'^*)$ before the honest party i^* makes its first query to $\mathsf{Hash}(\sigma'^*)$ when computing its rank. Under the simulation above and conditioned on $\bigwedge_{i=1}^{5} \mathsf{Good}_i$, the value σ'^* equals the signature component associated with the key $\mathsf{eph_vk}^* := \mathsf{trans}^*[0]$ and message $m^* := (i^*, tag^*)$ in the underlying threshold-signature scheme. In this case, $\mathcal{F}$ aborts and outputs $(\mathsf{eph_vk}^*, m^*, \sigma'^*)$ to $\mathcal{U}$. Therefore, $\mathcal{F}$ succeeds whenever $\mathcal{A}$ succeeds in producing such a σ'^*, which happens with probability

$$\Pr[\mathsf{Bad} \wedge \bigwedge_{i=1}^{5} \mathsf{Good}_i] \leq \Pr[\mathsf{Bad}] \cdot \frac{1}{\mathsf{poly}(\lambda)} \leq \mathsf{Adv}^{\mathrm{uf}}_{\mathrm{TS}}(\mathcal{F}) \cdot \frac{1}{\mathsf{poly}(\lambda)} = \mathsf{negl}(\lambda),$$

as we assume it is computationally hard to forge the threshold signature (which is shown in [BLL+24]).

Game $\mathbf{G}_5$ (simulate the coin proof). We modify the generation of the challenge proof π_{coin,i^*}. In $\mathbf{G}_4$, after the challenger programs $\mathsf{Hash}(\sigma'^*) = \tilde{r}$, the honest party i^* outputs $\mathsf{rank}_{i^*} = \tilde{r}$ together with a proof π_{coin,i^*} attesting that

$$\mathsf{rank}_{i^*} = \mathsf{Hash}(\sigma'^*) \ \wedge \ \mathsf{VerifyCoin}(\mathsf{eph_vk}^*, m^*, (\sigma'^*, \pi)) = 1 \quad \text{for some } \pi.$$

In $\mathbf{G}_5$, the challenger instead generates π_{coin,i^*} using the zero-knowledge simulator for the proof system $\mathsf{PS}_{\mathsf{coin}}$, given only the public statement $(\mathsf{eph_vk}^*, \tilde{r}, i^*, tag^*)$. If the simulator fails, the game aborts; we denote this event by ZKfail. By the zero-knowledge property of $\mathsf{PS}_{\mathsf{coin}}$, we have

$$|\Pr[\mathbf{G}_5 \Rightarrow 1] - \Pr[\mathbf{G}_4 \Rightarrow 1]| \ \leq \ \Pr[\mathsf{ZKfail}] \ = \ \mathsf{negl}(\lambda).$$

Game $\mathbf{G}_6$ (ideal secrecy game). In $\mathbf{G}_6$, the challenger sets $(R, \Pi) := (\tilde{r}, \tilde{\pi}_{i^*})$, where $\tilde{r} \leftarrow \mathcal{D}$ is sampled uniformly at random, and returns this pair together with the set $\{(\mathsf{rank}_j, \pi_j)\}_{j \neq i^* \in \mathcal{H}}$. By construction of $\tilde{\pi}_{i^*}$, we have

$$\mathsf{RankVerify}(\tilde{r}, i^*, \tilde{\pi}_{i^*}, tag^*) = 1.$$

Therefore, the distribution of the challenger's response in $\mathbf{G}_6$ is identical to that of the $b = 1$ branch of Definition 18.

Conclusion. By a standard hybrid argument over the sequence of games $\mathbf{G}_0, \ldots, \mathbf{G}_6$, we obtain

$$\Pr[\mathbf{G}_0 \Rightarrow 1] \ \leq \ \mathsf{negl}(\lambda).$$

It follows that $\mathcal{A}$'s advantage in the secrecy experiment of Definition 18 is negligible.

□

Lemma 4 (Validity)**.** *Assume that: (i)* $\mathsf{Disperse}$ *satisfies external validity with respect to the predicate* $\mathsf{ExchangeVerify}$*, (ii)* $\mathsf{PVSSExchange}$ *is verifiable, (iii) the non-equivocation protocol* NE *is complete and satisfies non-equivocation, (iv) the underlying threshold signature scheme is unforgeable and has a unique signature*

component σ' per public key and message, (v) the proof system $\mathsf{PS}_{\mathsf{coin}}$ is sound, and (vi) Hash is modeled as a random oracle.

Then the validity security experiment in Definition 20 returns 1 with probability negligibly close to $\frac{1}{2}$. In particular, the protocol pair satisfies the validity property.

Proof. Fix any PPT adversary $\mathcal{A}$ corrupting at most f parties and any PPT distinguisher $\mathcal{D}$. Let $k := |\mathcal{S}|$ be defined as in Definition 20, where $\mathcal{S} := \mathcal{O} \cup \mathcal{P}$ and $\mathcal{S} = \{(j_\ell, tag_\ell, \mathsf{rank}_\ell, \pi_\ell)\}_{\ell \in [k]}$ is enumerated in a canonical order.

We prove by a sequence of hybrid games that

$$(\mathsf{rank}_1, \ldots, \mathsf{rank}_k) \approx_c (U_1, \ldots, U_k),$$

where $(U_1, \ldots, U_k) \leftarrow \mathcal{D}^k$ are uniform and independent. This implies that $\mathcal{D}$'s distinguishing advantage in Definition 20 is negligible.

Game $\mathbf{G}_0$ (real validity game). This is the real experiment of Definition 20, including faithful emulation of all honest parties in each oracle execution of the $\mathsf{ProvableRank}(tag)$.

Game $\mathbf{G}_1$ (after Phase 1, reconstruct transcript secret keys). We modify the challenger's oracle $\mathsf{ProvableRank}(tag)$ as follows. At the moment Phase 1 completes (i.e., after dispersal/retrieval and transcript adoption), each party j (honest or corrupt) has adopted some aggregated transcript $\mathsf{trans}_{j,tag}$, with ephemeral verification key $\mathsf{eph_vk}_{j,tag} := \mathsf{trans}_{j,tag}[0]$.

In $\mathbf{G}_1$, immediately after Phase 1 finishes for this execution tag, the challenger reconstructs the secret key $\mathsf{eph_sk}(\mathsf{trans}_{j,tag})$ encoded by each $\mathsf{trans}_{j,tag}$, for every party j, as follows:

- For honest contributions: since the challenger emulates all honest parties, it knows their PVSS contributions and hence their secrets contributing to any aggregated transcript.
- For corrupted contributions included in $\mathsf{trans}_{j,tag}$: using the auxiliary data and proof of knowledge proofs, the challenger extracts the corrupted parties' secret contributions from $\mathsf{trans}_{j,tag}$.

Therefore, the challenger can compute the encoded secret key $\mathsf{eph_sk}(\mathsf{trans}_{j,tag})$ for each adopted transcript. This change is purely internal to the challenger and does not affect the messages shown to $\mathcal{A}$. Thus, both games are identically distributed and we get

$$\Pr[\mathbf{G}_1 \Rightarrow 1] = \Pr[\mathbf{G}_0 \Rightarrow 1].$$

Game $\mathbf{G}_2$ (precompute all σ' and program Hash right after Phase 1). We further modify the oracle $\textsc{ProvableRank}(tag)$. Right after Phase 1 completes (and after reconstructing each $\mathsf{eph_sk}(\mathsf{trans}_{j,tag})$ as in $\mathbf{G}_1$), the challenger precomputes, for every party j, the unique threshold signature component

$$\sigma'_{j,tag} \leftarrow \mathsf{PartialCoin}(\mathsf{eph_sk}(\mathsf{trans}_{j,tag}), (j, tag)).$$

Then the challenger programs the random oracle on each such $\sigma'_{j,tag}$ by lazy assignment: Sample $U_{j,tag} \leftarrow \mathcal{D}$ uniformly and independently, and set $\mathsf{Hash}(\sigma'_{j,tag}) := U_{j,tag}$. Store $\mathsf{rank}_{j,tag} := \mathsf{Hash}(\sigma'_{j,tag})$.

In Phase 3, whenever a party reconstructs its coin and queries $\mathsf{Hash}(\sigma'_{j,tag})$, the challenger answers consistently with the already-programmed value. All other protocol messages are emulated faithfully.

Bad event. Let Bad be the event that $\mathcal{A}$ (or any corrupted party) queries $\mathsf{Hash}(\sigma'_{j,tag})$ for some execution tag and some party j *before* the challenger programs it, i.e., before the end of Phase 1 of that execution when $\sigma'_{j,tag}$ is first computed in $\mathbf{G}_2$. Conditioned on $\neg\mathsf{Bad}$, the change from $\mathbf{G}_1$ to $\mathbf{G}_2$ is consistent with a true random oracle distribution, hence

$$\begin{aligned} &\Pr[\mathbf{G}_2 \Rightarrow 1 \mid \neg\mathsf{Bad}] = \Pr[\mathbf{G}_1 \Rightarrow 1 \mid \neg\mathsf{Bad}] \\ \Longrightarrow \quad &\big|\Pr[\mathbf{G}_2 \Rightarrow 1] - \Pr[\mathbf{G}_1 \Rightarrow 1]\big| \leq \Pr[\mathsf{Bad}]. \end{aligned}$$

We now show $\Pr[\mathsf{Bad}] \leq \mathsf{negl}(\lambda)$.

Bounding $\Pr[\mathsf{Bad}]$ **via threshold signature unforgeability.** Let $Q_{\mathsf{Hash}}(\lambda)$ be an a priori polynomial upper bound on the total number of Hash queries made by all parties, and let $Q_{\mathrm{exec}}(\lambda)$ bound the number of distinct tags queried to ProvableRank$(\cdot)$. We construct a PPT forger $\mathcal{F}$ against the underlying threshold signature scheme.

Guesses. $\mathcal{F}$ makes the following uniform guesses:

- $t^* \leftarrow [Q_{\mathrm{exec}}]$, the index of the execution/tag in which Bad occurs;
- $i^* \leftarrow [n]$, the party index for which the relevant σ'_{i^*,tag^*} is hit early;
- $q^* \leftarrow [Q_{\mathsf{Hash}}]$, the random oracle query index that equals σ'_{i^*,tag^*}; and
- $j^* \leftarrow [n]$, an honest contributor that appears in the transcript adopted by i^* in that execution.

Let Good be the event that all guesses are correct; clearly $\Pr[\mathsf{Good}] \geq 1/\mathsf{poly}(\lambda)$.

After the above guesses (t^*, i^*, q^*, j^*), the bad event Bad defined here is, up to notation, identical to the bad event analyzed in the secrecy proof of the Provable Rank protocol (see Lemma 3). In that proof, the challenger first performs a sequence of guessing hybrids to isolate a single challenge execution, a single honest party, and a single random oracle query, and then defines Bad as the event that the adversary queries Hash on the corresponding signature component before it is determined by the honest protocol execution. Here, we define Bad directly, but implicitly condition on the same successful guesses. Conditioned on these guesses being correct, the structure of the execution at the point Bad occurs is the same as in the secrecy proof: the adversary queries $\mathsf{Hash}(\sigma'^*)$ for the unique threshold signature component σ'^* corresponding to a public key $\mathsf{eph_vk}^*$ derived from an aggregated transcript that contains at least one honest contribution, and does so before any party can reconstruct the signature. Therefore, the simulation and reduction used to bound $\Pr[\mathsf{Bad}]$ are essentially identical to those in the secrecy proof. We thus give only a high-level idea of it.

Access to the unforgeability challenger. $\mathcal{F}$ receives the public parameters/PKI material from the unforgeability challenger $\mathcal{U}$, and simulates the protocol for $\mathcal{A}$ exactly as in the real game, forwarding corruption requests to $\mathcal{U}$ so that corrupt parties' secret states are obtained consistently.

Embedding an honest contribution in the guessed execution. Let tag^* be the t^*-th distinct tag on which $\mathcal{A}$ invokes ProvableRank$(\cdot)$ (as in our secrecy proof). In the simulation of Phase 1 for the execution tag^*, $\mathcal{F}$ ensures that the aggregated transcript trans^* ultimately adopted by party i^* contains an honestly generated contribution transcript trans_{j^*} provided by $\mathcal{U}$ for the (guessed) honest party j^*. This is done exactly as in our secrecy reduction: $\mathcal{F}$ generates all other honest contributions itself, and extracts corrupted contributions from the aggregated transcript. Conditioned on Good, the transcript adopted by i^* in execution tag^* is a transcript trans^* whose set of included contributions contains j^*, and thus $\mathsf{eph_vk}^* := \mathsf{trans}^*[0]$ is derived from an aggregated transcript containing at least one honest contribution.

Random oracle and the Bad *event.* $\mathcal{F}$ answers Hash queries by lazy sampling while recording the queried inputs as $(z_1, z_2, \ldots)$. If Bad occurs and Good holds, then before Phase 1 of execution tag^* completes, the adversary makes the query $z_{q^*} = \sigma'_{i^*,tag^*}$, where σ'_{i^*,tag^*} is the unique signature component that verifies for message $m^* := (i^*, tag^*)$ under verification key $\mathsf{eph_vk}^*$. In this case, $\mathcal{F}$ outputs the forgery $(\mathsf{eph_vk}^*, m^*, \sigma'^*) := (\mathsf{eph_vk}^*, (i^*, tag^*), z_{q^*})$ to $\mathcal{U}$.

Simulation of Phase 2 and of all other parties' ranks. The simulation of Phase 2 and of the ranks of all parties other than the challenged one i^* is performed exactly as in the secrecy proof of the Provable Rank protocol. Specifically, Phase 2 is simulated honestly for all parties except i^*, including the non-equivocation protocol and the dissemination of transcript shares together with the corresponding proofs. For Phase 3, the ranks and partial threshold signatures of all parties $(j, tag) \neq (i^*, tag^*)$ can be derived exactly as in the secrecy proof: since the unforgeability challenger $\mathcal{U}$ allows partial signature queries on any message other than (i^*, tag^*), the simulator can obtain the required partial signatures under the relevant transcript verification keys and translate them as needed. The corresponding proofs are also generated honestly. Consequently, the simulation of all protocol messages and outputs for parties other than i^* is faithful and indistinguishable from a real execution from the adversary's point of view.

Success probability. Whenever $\mathsf{Bad} \wedge \mathsf{Good}$ occurs, $\sigma'^* = z_{q^*}$ is a signature component for $(\mathsf{eph_vk}^*, m^*)$,

with $\mathsf{eph_vk}^*$ derived from an aggregated transcript containing the challenger-provided honest contribution. Thus $\mathcal{F}$ breaks unforgeability. Therefore,

$$\Pr[\mathsf{Bad}] \cdot \frac{1}{\mathsf{poly}(\lambda)} \leq \Pr[\mathsf{Bad} \wedge \mathsf{Good}] \leq \mathsf{Adv}^{\mathrm{uf}}_{\mathrm{TS}}(\mathcal{F}) = \mathsf{negl}(\lambda),$$

implying $\Pr[\mathsf{Bad}] \leq \mathsf{negl}(\lambda)$ and hence

$$\left|\Pr[\mathbf{G}_2 \Rightarrow 1] - \Pr[\mathbf{G}_1 \Rightarrow 1]\right| \leq \mathsf{negl}(\lambda).$$

Game $\mathbf{G}_3$ (consistency check for adversarial tuples at end of Challenge Phase I). Recall that in $\mathbf{G}_2$, for every oracle execution on tag tag and every party j, once Phase 1 completes and $\mathsf{trans}_{j,tag}$ (and hence $\mathsf{eph_vk}_{j,tag} := \mathsf{trans}_{j,tag}[0]$) is fixed, the challenger precomputes the unique signature component

$$\sigma'_{j,tag} \leftarrow \mathsf{PartialCoin}(\mathsf{eph_sk}(\mathsf{trans}_{j,tag}), (j, tag))$$

and programs the random oracle so that $\mathsf{rank}_{j,tag} := \mathsf{Hash}(\sigma'_{j,tag})$ is fixed thereafter.

In $\mathbf{G}_3$, we modify the challenger only at the end of *Challenge phase I*, when the adversary outputs $\mathcal{P} = \{(j, tag, \mathsf{rank}, \pi)\}_{j \in \mathcal{C}}$. For each tuple $(j, tag, \mathsf{rank}, \pi) \in \mathcal{P}$, after checking $\mathsf{RankVerify}(\mathsf{rank}, j, \pi, tag) = 1$ as in the original experiment, the challenger additionally checks the following *consistency condition*:

> Parse π as $(\mathsf{eph_vk}, \pi_{\mathsf{NE}}, \pi_{\mathsf{coin}}, \pi_{\mathsf{trans}}, \pi_{\mathsf{key}})$ and let $\mathsf{eph_vk}_{j,tag}$ denote the unique ephemeral verification key fixed by Phase 1 for party j in execution tag (uniqueness follows from non-equivocation). Check that $\mathsf{eph_vk} = \mathsf{eph_vk}_{j,tag}$ and that $\mathsf{rank} = \mathsf{rank}_{j,tag}$.

If any tuple fails this consistency check, the challenger aborts.

Justification. We claim that $\mathbf{G}_3$ and $\mathbf{G}_2$ are computationally indistinguishable. The only way they can differ is if $\mathcal{A}$ outputs a verifying tuple $(j, tag, \mathsf{rank}, \pi) \in \mathcal{P}$ that passes $\mathsf{RankVerify}$ but fails the above consistency.

Fix such a failing tuple and consider its parsed $\mathsf{eph_vk}$. By soundness of non-equivocation, $\mathsf{eph_vk}$ must equal the unique key $\mathsf{eph_vk}_{j,tag}$ corresponding to party j and tag tag, or else π_{NE} is a soundness violation of the non-equivocation proof system/protocol. Hence we may assume $\mathsf{eph_vk} = \mathsf{eph_vk}_{j,tag}$.

Now consider the statement verified inside $\mathsf{RankVerify}$: $\mathsf{PVerify}_{\mathsf{coin}}((\mathsf{eph_vk}, \mathsf{rank}, j, tag), \pi_{\mathsf{coin}}) = 1$. Because the threshold signature component is unique per public key and message, there exists at most one value $\sigma'_{j,tag}$ such that $\mathsf{VerifyCoin}(\mathsf{eph_vk}_{j,tag}, (j, tag), (\sigma'_{j,tag}, \cdot)) = 1$. Moreover, in $\mathbf{G}_2$ the challenger already programmed $\mathsf{Hash}(\sigma'_{j,tag}) = \mathsf{rank}_{j,tag}$. Therefore, for $\mathsf{rank} \neq \mathsf{rank}_{j,tag}$ the statement

$$\exists\, \sigma', \pi^* \;:\; \mathsf{rank} = \mathsf{Hash}(\sigma') \wedge \mathsf{VerifyCoin}(\mathsf{eph_vk}_{j,tag}, (j, tag), (\sigma', \pi^*)) = 1$$

is *false*. If nevertheless π_{coin} verifies for $(\mathsf{eph_vk}_{j,tag}, \mathsf{rank}, j, tag)$, this yields a proof of a false statement, contradicting the soundness of $\mathsf{PS}_{\mathsf{coin}}$.

Reduction. Formally, we build a PPT adversary $\mathcal{B}$ against the soundness of $\mathsf{PS}_{\mathsf{coin}}$ as follows. $\mathcal{B}$ runs the simulation of $\mathbf{G}_2$ for $\mathcal{A}$, including the precomputation of all $\sigma'_{j,tag}$ and the programming of Hash after Phase 1 of each execution, thereby determining each $\mathsf{rank}_{j,tag}$. When $\mathcal{A}$ outputs $\mathcal{P}$, $\mathcal{B}$ searches for a tuple $(j, tag, \mathsf{rank}, \pi)$ such that $\mathsf{RankVerify}(\mathsf{rank}, j, \pi, tag) = 1$ but $\mathsf{rank} \neq \mathsf{rank}_{j,tag}$ (with $\mathsf{eph_vk} = \mathsf{eph_vk}_{j,tag}$ as argued above). On finding the first such tuple, $\mathcal{B}$ outputs the corresponding public statement

$$x := (\mathsf{eph_vk}_{j,tag}, \mathsf{rank}, j, tag)$$

together with the purported proof π_{coin} extracted from π. By the argument above, x is false yet the proof verifies $\mathsf{PVerify}_{\mathsf{coin}}(x, \pi_{\mathsf{coin}}) = 1$, violating soundness. Thus the probability that $\mathcal{A}$ causes $\mathbf{G}_3$ to abort is at most negligible, and hence

$$\left|\Pr[\mathbf{G}_3 \Rightarrow 1] - \Pr[\mathbf{G}_2 \Rightarrow 1]\right| \leq \mathsf{negl}(\lambda).$$

Game $\mathbf{G}_4$ (replace all ranks by independent uniform). In game $\mathbf{G}_3$, by construction, whenever the challenger assigns $\mathsf{Hash}(\sigma'_\ell)$ for the first time, it samples this value uniformly at random from $\mathcal{D}$. Thus, each individual value

$$\mathsf{rank}_\ell = \mathsf{Hash}(\sigma'_\ell)$$

is marginally distributed uniformly over $\mathcal{D}$.

We now argue joint independence. Fix any two distinct indices $\ell \neq \ell'$. If $(j_\ell, tag_\ell) = (j_{\ell'}, tag_{\ell'})$, then by the uniqueness property of the threshold signature scheme, there exists at most one signature component σ' that verifies under the public key $\mathsf{eph_vk}_\ell$ for message (j_ℓ, tag_ℓ). Therefore, $\sigma'_\ell = \sigma'_{\ell'}$, and the two tuples correspond to the same element of $\mathcal{S}$. Otherwise, if $(j_\ell, tag_\ell) \neq (j_{\ell'}, tag_{\ell'})$, then the uniqueness property implies that the corresponding verifying signature components σ'_ℓ and $\sigma'_{\ell'}$ must be distinct. Consequently, the challenger programs the random oracle at two distinct inputs $\sigma'_\ell \neq \sigma'_{\ell'}$. Since the random oracle is programmed independently at each previously unassigned input, the values $\mathsf{Hash}(\sigma'_\ell)$ and $\mathsf{Hash}(\sigma'_{\ell'})$ are independent uniform samples from $\mathcal{D}$. This argument extends to any finite subset of indices, and thus the random variables $\mathsf{Hash}(\sigma'_1), \ldots, \mathsf{Hash}(\sigma'_k)$ are jointly independent and uniformly distributed over $\mathcal{D}^k$.

Therefore, the joint distribution $(\mathsf{rank}_1, \ldots, \mathsf{rank}_k)$ in game $\mathbf{G}_3$ is identical to $(U_1, \ldots, U_k) \leftarrow \mathcal{D}^k$, and hence $\mathbf{G}_3$ and $\mathbf{G}_4$ are identical. That is,

$$\Pr[\mathbf{G}_4 \Rightarrow 1] = \Pr[\mathbf{G}_3 \Rightarrow 1].$$

Combining the hybrid games, we obtain

$$(\mathsf{rank}_1, \ldots, \mathsf{rank}_k) \approx_c (U_1, \ldots, U_k).$$

Therefore, $\mathcal{D}$ distinguishes the cases $b = 0$ and $b = 1$ in Definition 20 with at most negligible advantage, and the experiment returns 1 with probability negligibly close to 1/2. □

Lemma 5 (Termination)**.** *Assume that: (i)* PVSSExchange *is verifiable and has termination, (ii)* Disperse *has provability, termination, totality, binding, and external validity with respect to the predicate* ExchangeVerify*, (iii) the non-equivocation protocol* NE *has validity and liveness, (iv) the underlying threshold signature scheme is correct, and (v) the proof systems* $\mathsf{PS}_{\mathsf{trans}}$*,* $\mathsf{PS}_{\mathsf{key}}$*, and* $\mathsf{PS}_{\mathsf{share}}$ *are complete.*

Then the protocol pair satisfies the termination property.

Proof. All honest parties start the protocol by invoking $\mathsf{PVSSExchange}(tag)$. By the Termination and Verifiability properties of PVSSExchange, there exists at least one forever-honest party i that eventually outputs a pair $(\mathsf{trans}, \pi_{\mathsf{trans}})$ satisfying $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi_{\mathsf{trans}}, tag) = 1$. Party i then invokes $\mathsf{Disperse}(\mathsf{trans}, \pi_{\mathsf{trans}})$, and all honest parties participate in every invocation of Disperse using external validity predicate ExchangeVerify. By the Termination and Provability properties of Disperse, party i eventually outputs a proof π_{disp} such that $\mathsf{DisperseVerify}(i, \pi_{\mathsf{disp}}) = 1$ and sends it to all parties in a $\langle$"dispersed", $\pi_{\mathsf{disp}}\rangle$ message.

Every honest party eventually receives this message, verifies $\mathsf{DisperseVerify}(i, \pi_{\mathsf{disp}}) = 1$, and (if it has not already done so) sends a $\langle$"retrieve", $i\rangle$ message. Moreover, every honest party eventually receives the $\langle$"retrieve", $k\rangle$ messages sent by every other honest party k and participates in the corresponding PrivateRetrieve instances. Since each "retrieve" message is sent only after verifying a valid dispersal, the Totality property of the protocol implies that every honest party j eventually outputs a pair $(\mathsf{trans}, \pi_{\mathsf{trans}})$ from PrivateRetrieve and stores it in $(\mathsf{trans}_j, \pi_{\mathsf{trans},j})$ if these variables were not already set earlier. By the Binding, External Validity, and Agreement properties, the stored pair satisfies $\mathsf{ExchangeVerify}(\mathsf{trans}_j, \pi_{\mathsf{trans},j}, tag) = 1$. If $(\mathsf{trans}_j, \pi_{\mathsf{trans},j})$ was instead stored earlier directly from PVSSExchange, the same property follows from its Verifiability.

Each honest party i then invokes the non-equivocation protocol NE with input $\mathsf{eph_vk}_i = \mathsf{trans}_i[0]$ and tag ("flip", tag). By the Validity and Liveness of NE, party i eventually outputs a proof $\pi_{\mathsf{NE},i}$ such that $\mathsf{NEVerify}(i, \mathsf{eph_vk}_i, (\text{"flip"}, tag), \pi_{\mathsf{NE},i}) = 1$. Party i then sends every party j a $\langle$"flip", $\cdot\rangle$ message containing the key $\mathsf{eph_vk}_i$, the proofs $\pi_{\mathsf{NE},i}$, $\pi_{\mathsf{trans},i}$, and $\pi_{\mathsf{key},i}$, its transcript share $\mathsf{share}_j = \mathsf{trans}_i[j]$, and a proof $\pi_{\mathsf{share},j}$ that share_j is the j-th share of trans_i.

Every honest party j eventually receives this message, verifies all included proofs, extracts S_j from share_j, and replies with $\langle$"flip", $\mathsf{PartialCoin}(S_j, (i, tag))\rangle$. Party i eventually receives such messages from every honest party, verifies each partial coin $\sigma_j = \mathsf{PartialCoin}(S_j, (i, tag))$ using $\mathsf{VerifyPartialCoin}(\mathsf{trans}_i[j], (i, tag), \sigma_j) = 1$, and records the corresponding tuples. After collecting $f+1$ valid partial coins, party i performs the prescribed local computation and outputs a pair $(\mathsf{rank}_i, \pi_{\mathsf{rank},i})$.

□

□

C.2 Key Escrow (Section 6)

Henceforth, we denote by $|\pi|$ the (maximum) size of our underlying zkSNARKs. While our zkSNARKs are instantiated with proof size $O(\lambda)$ [Fu25, AGL+23], our protocols admit alternative instantiations provided that the proof size remains $O(\lambda^2)$. Our complexity analysis is stated in terms of the proof length $|\pi|$ to keep the presentation generic and to accommodate such alternative instantiations.

Theorem 6 (Complexity)**.** *The* KeyEscrow *and* KeyRetrieve *protocols have $O(\lambda^2 n^2)$ and $O(|\pi| n^2)$ communication complexity, respectively. In particular, when $|\pi| \in O(\lambda^2)$, this is $O(\lambda^2 n^2)$. In addition, the protocols have $O(1)$ round complexity.*

Proof. During the KeyEscrow and KeyRetrieve protocols, parties execute actions very similar to those performed during the ProvableRank protocol, resulting in a similar analysis. More specifically, parties do the following in the KeyEscrow protocol:

- call PVSSExchange, costing $O(\lambda^2 n^2)$;
- send each other proofs of size $O(|\pi|)$ for a total of $O(|\pi| n^2)$; and,
- each disperses a transcript and privately retrieves a single transcript for a total cost of $O(\lambda n^2 + n \cdot (\lambda n))$.

In the KeyRetrieve protocol, parties do the following:

- send each other a public key along with a share, each of size $O(\lambda)$, as well as a constant number of proofs of size $O(|\pi|)$, for a total of $O(|\pi| n^2)$; and,
- send each other decrypted shares of size $O(\lambda)$ for a total of $O(\lambda n^2)$.

In total, KeyEscrow costs $O((\lambda^2 + |\pi|) n^2)$ and KeyRetrieve costs $O(|\pi| n^2)$. In particular, when $|\pi| \in O(\lambda^2)$, this is $O(\lambda^2 n^2)$.

In addition, the protocols consist of a constant number of all-to-all rounds, as well as constant-round protocols. This results in both having $O(1)$ round complexity. □

Theorem 7 (Security)**.** *The tuple* (KeyEscrow, KeyVerify, KeyRetrieve) *(Algorithms 3 to 5) is a Key Escrow protocol resilient to $f < n/3$ Byzantine parties, except with negligible probability in λ.*

Proof. We prove each property in a separate lemma and explicitly state the required properties of the underlying primitives. Although all primitives in our construction satisfy these requirements, this modular presentation highlights which assumptions are needed for each guarantee and facilitates replacing components with alternative implementations satisfying the same properties.

Lemma 6 (Correctness)**.** *Assume that: (i)* PVSSExchange *is verifiable, and (ii)* Disperse *satisfies external validity with respect to the predicate* ExchangeVerify*.*

Then for any tag and any honest party i, if i completes KeyEscrow(tag) *and outputs* $(\mathsf{eph_pk}_i, \pi_i)$*, then there exists a unique secret key* $\mathsf{eph_sk}_i$ *such that* $(\mathsf{eph_pk}_i, \mathsf{eph_sk}_i) \in \mathcal{K}$*. Moreover, if i subsequently completes* KeyRetrieve(tag)*, then it outputs exactly* $\mathsf{eph_sk}_i$*. In particular, the protocol tuple satisfies the correctness property.*

Proof. Fix a tag and an honest party i that completes KeyEscrow(tag) and outputs $(\mathsf{eph_pk}_i, \pi_i)$. We proceed in three steps.

Step 1: Correctness of the transcript. Party i outputs $(\mathsf{eph_pk}_i, \pi_i)$ only after setting $\mathsf{eph_pk}_i := \mathsf{trans}_i[0]$, where trans_i is obtained either directly as the output $(\mathsf{trans}_i, \pi_{\mathsf{trans},i})$ of $\mathsf{PVSSExchange}(tag)$, or via PrivateRetrieve following a successful execution of Disperse by some party. In both cases, by the verifiability guarantee of PVSSExchange and the external validity property of Disperse with respect to the predicate ExchangeVerify, it holds that

$$\mathsf{ExchangeVerify}(\mathsf{trans}_i, \pi_{\mathsf{trans},i}, tag) = 1.$$

Thus, except with negligible probability, trans_i is a well-formed aggregated PVSS transcript that contains at least one honestly generated contribution.

Step 2: Existence and uniqueness of the escrowed secret key. By correctness of the underlying PVSS scheme, any well-formed transcript trans_i uniquely determines a secret value $\mathsf{eph_sk}_i$ whose corresponding public value is $\mathsf{eph_pk}_i = \mathsf{trans}_i[0]$. Since the key space $\mathcal{K}$ satisfies uniqueness of secret keys for public keys, it follows that $(\mathsf{eph_pk}_i, \mathsf{eph_sk}_i) \in \mathcal{K}$ and that $\mathsf{eph_sk}_i$ is uniquely defined.

Step 3: Correctness of key reconstruction. Suppose party i subsequently completes $\mathsf{KeyRetrieve}(tag)$. During this protocol, party i distributes the transcript shares $\mathsf{trans}_i[j]$ together with proofs of correctness. For each party j, any response S_j accepted by i satisfies $\mathsf{ShareVer}(S_j, \mathsf{trans}_i[j]) = 1$ and therefore constitutes a valid secret share corresponding to trans_i. Party i waits until it has collected $f + 1$ such valid shares and then computes

$$\mathsf{eph_sk}'_i := \mathsf{Reconstruct}(\{(j, S_j)\}).$$

By correctness of the PVSS reconstruction algorithm, reconstruction from any $f + 1$ valid shares yields exactly the secret encoded in trans_i. Hence $\mathsf{eph_sk}'_i = \mathsf{eph_sk}_i$, and party i outputs the correct secret key.

This concludes the proof. □

Lemma 7 (Provability)**.** *Assume that: (i)* PVSSExchange *is verifiable, (ii)* Disperse *satisfies external validity with respect to the predicate* ExchangeVerify*, and (iii) the proof system* $\mathsf{PS}_{\mathsf{key}}$ *is complete.*

Then for any tag and any honest party i, if i completes an execution of $\mathsf{KeyEscrow}(tag)$ *and outputs* $(\mathsf{eph_pk}_i, \pi_i)$*, it holds that* $\mathsf{KeyVerify}(\mathsf{eph_pk}_i, \pi_i, tag) = 1$*. In particular, the protocol tuple satisfies the provability property.*

Proof. Fix a tag and an honest party i that completes $\mathsf{KeyEscrow}(tag)$ with output $(\mathsf{eph_pk}_i, \pi_i)$. By construction, i outputs

$$\pi_i = (\pi_{\mathsf{trans},i}, \pi_{\mathsf{key},i}),$$

where $\mathsf{eph_pk}_i = \mathsf{trans}_i[0]$ and $\pi_{\mathsf{key},i}$ is generated as

$$\pi_{\mathsf{key},i} \leftarrow \mathsf{PProve}_{\mathsf{key}}\big((\mathsf{eph_pk}_i, \pi_{\mathsf{trans},i}, tag), \mathsf{trans}_i\big).$$

The values $(\mathsf{trans}_i, \pi_{\mathsf{trans},i})$ are assigned either directly as the output of $\mathsf{PVSSExchange}(tag)$ or via the PrivateRetrieve following a successful execution of Disperse with external validity predicate ExchangeVerify. In both cases, by the verifiability property of PVSSExchange and the external validity property of provable AVID, it holds that

$$\mathsf{ExchangeVerify}(\mathsf{trans}_i, \pi_{\mathsf{trans},i}, tag) = 1.$$

Therefore, the pair $\big((\mathsf{eph_pk}_i, \pi_{\mathsf{trans},i}, tag), \mathsf{trans}_i\big)$ satisfies the relation underlying the proof system $\mathsf{PProve}_{\mathsf{key}}$. By completeness of the proof system, the proof $\pi_{\mathsf{key},i}$ verifies, and hence

$$\mathsf{KeyVerify}(\mathsf{eph_pk}_i, \pi_i, tag) = \mathsf{PVerify}_{\mathsf{key}}\big((\mathsf{eph_pk}_i, \pi_{\mathsf{trans},i}, tag), \pi_{\mathsf{key},i}\big) = 1.$$

□

Lemma 8 (Validity)**.** *Assume that: (i)* PVSSExchange *is verifiable, (ii) the proof system* $\mathsf{PS}_{\mathsf{share}}$ *is complete, (iii) the proof system* $\mathsf{PS}_{\mathsf{key}}$ *is knowledge-sound with extractor* $\mathsf{Ext}_{\mathsf{key}}$*, and (iv) the underlying aggregatable PVSS scheme satisfies aggregated unpredictability (see Definition 14).*

Then for any tuple $(\mathsf{eph_pk}, \pi, tag)$ *such that* $\mathsf{KeyVerify}(\mathsf{eph_pk}, \pi, tag) = 1$*, there exists a unique* $\mathsf{eph_sk}$ *with* $(\mathsf{eph_pk}, \mathsf{eph_sk}) \in \mathcal{K}$*, and moreover* $\mathsf{eph_sk}$ *is unpredictable to any PPT adversary before any execution of* $\mathsf{KeyRetrieve}(tag)$ *(cf. Definition 6). In particular, the protocol tuple satisfies the validity property.*

Proof. We prove the two parts of validity: (1) existence and uniqueness of $\mathsf{eph_sk}$, and (2) unpredictability.

Part 1: Existence and uniqueness of the secret key. Fix any tuple $(\mathsf{eph_pk}, \pi, tag)$ such that $\mathsf{KeyVerify}(\mathsf{eph_pk}, \pi, tag) = 1$, and write $\pi = (\pi_{\mathsf{trans}}, \pi_{\mathsf{key}})$. By definition of $\mathsf{KeyVerify}$ and the fact that it accepts the tuple, we have

$$\mathsf{PVerify}_{\mathsf{key}}\big((\mathsf{eph_pk}, \pi_{\mathsf{trans}}, tag), \pi_{\mathsf{key}}\big) = 1.$$

By soundness of the proof system $\mathsf{PS}_{\mathsf{key}}$, we know that there exists a witness transcript trans such that

$$\mathsf{ExchangeVerify}(\mathsf{trans}, \pi_{\mathsf{trans}}, tag) = 1 \quad \text{and} \quad \mathsf{eph_pk} = \mathsf{trans}[0],$$

except with negligible probability.

By verifiability of $\mathsf{PVSSExchange}$, if $\mathsf{ExchangeVerify}(\mathsf{trans}, \pi_{\mathsf{trans}}, tag) = 1$, then trans is a valid/well-formed aggregated PVSS transcript that contains at least one honest contribution. By correctness of the underlying aggregatable PVSS scheme, the transcript trans uniquely determines a secret key $\mathsf{eph_sk}$ whose corresponding public key is $\mathsf{trans}[0] = \mathsf{eph_pk}$. Hence $(\mathsf{eph_pk}, \mathsf{eph_sk}) \in \mathcal{K}$.

Finally, uniqueness follows because (i) the secret encoded by a valid PVSS transcript is uniquely defined by the transcript, and (ii) the key space $\mathcal{K}$ satisfies the stated uniqueness property (for any public key, there exists at most one corresponding secret key). This concludes Part 1.

Part 2: Unpredictability of secret keys. We now prove that the secret key $\mathsf{eph_sk}$ corresponding to any valid tuple remains unpredictable until $\mathsf{KeyRetrieve}(tag)$ is executed. Formally, we show that every PPT adversary wins the experiment in Definition 6 only with negligible probability.

Let $\mathcal{A}$ be any PPT adversary in Definition 6. We define a sequence of games and reduce $\mathcal{A}$'s success probability in the final game to the Aggregated Unpredictability of the APVSS scheme (cf. Definition 14), which is computationally hard as shown in [BL23] (under the one-more discrete logarithm assumption, see Definition 17).

Game $\mathbf{G}_0$ (real unpredictability game). This is exactly the unpredictability experiment in Definition 6.

Game $\mathbf{G}_1$ (extract a witness transcript). This game is identical to $\mathbf{G}_0$ except that in the winning check, when $\mathcal{A}$ outputs $(\mathsf{eph_pk}^*, \pi^* = (\pi^*_{\mathsf{trans}}, \pi^*_{\mathsf{key}}), tag^*)$ and $\mathsf{eph_sk}^*$ with $\mathsf{KeyVerify}(\mathsf{eph_pk}^*, \pi^*, tag^*) = 1$, the challenger additionally runs the extractor $\mathsf{Ext}_{\mathsf{key}}$ guaranteed by knowledge-soundness of $\mathsf{PS}_{\mathsf{key}}$ to obtain a witness transcript trans^*. If extraction fails, the game aborts.

By knowledge-soundness of $\mathsf{PS}_{\mathsf{key}}$, the probability that $\mathsf{PVerify}_{\mathsf{key}}$ accepts but extraction fails is negligible, and therefore

$$\big|\Pr[\mathbf{G}_1 \Rightarrow 1] - \Pr[\mathbf{G}_0 \Rightarrow 1]\big| \leq \mathsf{negl}(\lambda).$$

Moreover, conditioned on not aborting, we have

$$\mathsf{ExchangeVerify}(\mathsf{trans}^*, \pi^*_{\mathsf{trans}}, tag^*) = 1 \quad \text{and} \quad \mathsf{eph_pk}^* = \mathsf{trans}^*[0].$$

Game $\mathbf{G}_2$ (fixing an honest contributing index). This game is identical to $\mathbf{G}_1$, and we additionally define a (conceptual) index $i^* \in [n]$ as follows. Conditioned on $\mathsf{ExchangeVerify}(\mathsf{trans}^*, \pi^*_{\mathsf{trans}}, tag^*) = 1$, by verifiability of $\mathsf{PVSSExchange}$ we know that trans^* is a valid/well-formed aggregated PVSS transcript and

$$\mathsf{Contr}(\mathsf{trans}^*) \cap \mathcal{H} \neq \emptyset,$$

where $\mathcal{H}$ denotes the set of honest parties (at the end of the experiment), i.e., the transcript has at least one honest contribution. Let i^* be any fixed choice of an index in $\mathsf{Contr}(\mathsf{trans}^*) \cap \mathcal{H}$ (e.g., the lexicographically smallest such index). No additional checks are performed in this game; this step only introduces notation. In particular, we have

$$\Pr[\mathbf{G}_2 \Rightarrow 1] = \Pr[\mathbf{G}_1 \Rightarrow 1].$$

Game $\mathbf{G}_3$ (guess the forged $\mathsf{KeyEscrow}$ execution). Let q_{esc} be an upper bound on the number of $\mathsf{KeyEscrow}(\cdot)$ queries made by $\mathcal{A}$ (as specified by the experiment, or by a syntactic bound on $\mathcal{A}$). This game

is identical to $\mathbf{G}_2$ except that the challenger guesses which $\mathsf{KeyEscrow}$ execution will be the one for which $\mathcal{A}$ eventually forges a valid output.

Concretely, at the beginning of the game, the challenger samples an index $\tilde{j} \leftarrow [q_{\mathsf{esc}}]$ uniformly at random. During the interaction, the challenger records the $\tilde{j}$-th $\mathsf{KeyEscrow}(\cdot)$ query made by $\mathcal{A}$; denote its tag by $\tilde{tag}$ (if fewer than $\tilde{j}$ such queries are made, set $\tilde{tag} := \bot$). At the end of the game, after $\mathcal{A}$ outputs its forgery with tag tag^* and trans^* is extracted, the game aborts unless $tag^* = \tilde{tag}$.

Since $\tilde{j}$ is sampled independently of $\mathcal{A}$'s view, we have

$$\Pr[\mathbf{G}_3 \Rightarrow 1] \;\geq\; \frac{1}{q_{\mathsf{esc}}} \cdot \Pr[\mathbf{G}_2 \Rightarrow 1].$$

Game $\mathbf{G}_4$ (guess the special honest contributor). This game is identical to $\mathbf{G}_3$ except that the challenger guesses the special honest contributing index i^*. Concretely, at the beginning of the game, the challenger samples $\tilde{i} \leftarrow [n]$ uniformly at random and executes the game as in $\mathbf{G}_3$. At the end of the game, after $\mathcal{A}$ outputs its forgery and trans^* is extracted, the game aborts if $\tilde{i} \neq i^*$.

Since $\tilde{i}$ is sampled independently of $\mathcal{A}$'s view, we can lower bound the success probability as

$$\Pr[\mathbf{G}_4 \Rightarrow 1] \;\geq\; \frac{1}{n} \cdot \Pr[\mathbf{G}_3 \Rightarrow 1].$$

Building a reduction. We build an efficient reduction $\mathcal{R}$ to the aggregated unpredictability game of APVSS (Definition 14) such that

$$\Pr[\mathbf{G}_4 \Rightarrow 1] \;\leq\; \Pr\big[\mathcal{R} \text{ wins } \textsc{AggUnpred}_{\mathsf{APVSS}}\big] + \mathsf{negl}(\lambda).$$

Combining the above inequalities and the game transitions will imply

$$\Pr[\mathbf{G}_0 \Rightarrow 1] \;\leq\; q_{\mathsf{esc}} \cdot n \cdot \varepsilon_{\mathsf{APVSS}} + \mathsf{negl}(\lambda),$$

as desired. We now explain the reduction.

Recall of aggregated unpredictability. In the aggregated unpredictability game of APVSS, the challenger first runs the key-generation algorithm for all parties: for every $i \in [n]$, it generates plain PKI keys $(\mathsf{pk}_i, \mathsf{sk}_i)$ honestly. It then gives parameters and the public-key list $\mathcal{L} = \{\mathsf{pk}_i\}_{i\in[n]}$ to the adversary $\mathcal{A}$. In the setup phase, $\mathcal{A}$ outputs an index set $\mathcal{C} \subseteq [n]$ of initially corrupted parties, together with replacement public keys $\{\hat{\mathsf{pk}}_j\}_{j\in\mathcal{C}}$; the challenger sets $\mathsf{pk}_j := \hat{\mathsf{pk}}_j$ for all $j \in \mathcal{C}$ (in particular, the updated keys can be generated maliciously). Let $\mathcal{H} := [n] \setminus \mathcal{C}$ denote the initially honest parties. Next, $\mathcal{A}$ is given oracle access to a *corruption oracle* $\mathcal{O}_{\mathsf{corr}}$: on input $i \in [n] \setminus \mathcal{C}$, it returns party i's internal state containing sk_i and updates the corruption set $\mathcal{C} := \mathcal{C} \cup \{i\}$. The adversary may make at most f total corruptions (counting the initial set $\mathcal{C}$ and any subsequent corruption queries). In addition, $\mathcal{A}$ has access to a *PVSS transcript oracle* $\mathcal{O}_{\mathsf{PVSS}}$, which on input $(\mathsf{givePVSS}, i)$ for some $i \in [n] \setminus \mathcal{C}$ returns a fresh PVSS transcript generated on behalf of party i using Dist with the current key list. Finally, $\mathcal{A}$ outputs an aggregated PVSS transcript T^* together with a candidate secret S^*, and wins if the transcript verifies, it contains at least one honest contribution (i.e., $\mathsf{Contr}(T^*) \cap \mathcal{H} \neq \emptyset$), and the candidate secret S^* equals the secret encoded by the transcript. Intuitively, this notion captures the property that the secret encoded in a transcript containing at least one honest contribution remains unpredictable to the adversary prior to the reconstruction phase.

One may observe that our notion of unpredictability for Key Escrow is closely aligned with the notion of aggregated unpredictability for APVSS.

Reduction $\mathcal{R}$. Reduction $\mathcal{R}$ interacts with its APVSS challenger as follows. It receives the public parameters and an initial public-key list $\mathcal{L} = \{\mathsf{pk}_i\}_{i\in[n]}$ from the APVSS challenger and forwards them to $\mathcal{A}$ as the public keys and parameters of the Key Escrow experiment.

Setup phase. When $\mathcal{A}$ outputs an initial corruption set $\mathcal{C} \subseteq [n]$ together with replacement public keys $\{\hat{\mathsf{pk}}_j\}_{j\in\mathcal{C}}$, reduction $\mathcal{R}$ forwards the same set $\mathcal{C}$ and replacement keys to its APVSS challenger (or, equivalently, updates the public-key list by setting $\mathsf{pk}_j := \hat{\mathsf{pk}}_j$ for all $j \in \mathcal{C}$). The parties in $\mathcal{C}$ are henceforth treated as

corrupted, and $\mathcal{R}$ gives $\mathcal{A}$ full control over their internal state. Let $\mathcal{H} := [n] \setminus \mathcal{C}$ denote the set of initially honest parties.

Guessing. At the beginning, $\mathcal{R}$ samples $\tilde{j} \leftarrow [q_{\mathsf{esc}}]$ and $\tilde{i} \leftarrow [n]$ uniformly at random as in game $\mathbf{G}_4$. It maintains a counter $j \leftarrow 0$ for $\mathsf{KeyEscrow}(\cdot)$ queries, and will record $\tilde{tag}$ as the tag appearing in the $\tilde{j}$-th such query (if it occurs).

Corruption queries. Whenever $\mathcal{A}$ corrupts a party i, $\mathcal{R}$ forwards this query to its own corruption oracle $\mathcal{O}_{\mathsf{corr}}$ and returns the obtained internal state (including the secret key) to $\mathcal{A}$. Since $\mathcal{A}$ corrupts at most f parties, $\mathcal{R}$ makes at most f such queries, as required by the aggregated unpredictability experiment.

KeyEscrow(tag) queries. On a query KeyEscrow(tag), $\mathcal{R}$ increments $j \leftarrow j + 1$. If $j = \tilde{j}$, it sets $\tilde{tag} \leftarrow tag$ (otherwise it leaves $\tilde{tag}$ unchanged). Then $\mathcal{R}$ simulates an execution of $\mathsf{KeyEscrow}(tag)$ among all parties exactly as in the real game. That is, it faithfully simulates the prescribed network schedule, executes the protocol code of all honest parties using fresh randomness, and allows the adversary $\mathcal{A}$ to fully control the behavior of corrupted parties. In particular, all PVSS transcripts generated in this execution, as well as all accompanying proofs (including the proofs in π_{trans} and π_{key}), are generated honestly from the corresponding witnesses in the simulated execution.

The only difference is for the *special* execution with $tag = \tilde{tag}$ (i.e., when $j = \tilde{j}$): $\mathcal{R}$ programs the simulation so that the contribution of party $\tilde{i}$ to the aggregated PVSS transcript of that execution is taken from the APVSS transcript oracle. Concretely, $\mathcal{R}$ issues a query $(\mathsf{givePVSS}, \tilde{i})$ to $\mathcal{O}_{\mathsf{PVSS}}$ to obtain a transcript $\mathsf{trans}_{\tilde{i}}$ generated on behalf of $\tilde{i}$, and ensures that $\mathsf{trans}_{\tilde{i}}$ is included as one of the aggregated components in the aggregated transcript trans^* for that execution (note that this property is already guaranteed by the changes introduced in games $\mathbf{G}_1$ through $\mathbf{G}_4$.). All other transcripts in that execution (and all transcripts in any non-special execution $tag \neq \tilde{tag}$) are generated honestly in the simulation. All proofs in π_{trans} and π_{key} are generated honestly from the corresponding witnesses in the simulated world.

KeyRetrieve(tag) queries. On query KeyRetrieve(tag), $\mathcal{R}$ simulates $\mathsf{KeyRetrieve}(tag)$ honestly using the simulated state of the corresponding key escrow $\mathsf{KeyEscrow}(tag)$ execution. This is always possible because the reduction knows the honest parties' internal randomness for every simulated transcript *except* for the embedded component obtained from $\mathcal{O}_{\mathsf{PVSS}}$ in the special execution $tag = \tilde{tag}$.

However, in $\mathbf{G}_4$ the reduction aborts unless $\tilde{i} = i^* \in \mathcal{H}$, and the winning condition of Definition 6 enforces that the forgery tag tag^* has not been retrieved, i.e., $tag^* \notin \mathcal{Q}_2$. Moreover, $\mathbf{G}_3$ enforces $tag^* = \tilde{tag}$. Therefore, conditioned on $\mathbf{G}_4$ not aborting, $\mathcal{R}$ never needs to simulate $\mathsf{KeyRetrieve}(\tilde{tag}) = \mathsf{KeyRetrieve}(tag^*)$, and hence never needs to reconstruct. Thus the simulation is well-defined.

Final output. At the end of the interaction, $\mathcal{A}$ outputs $(\mathsf{eph_pk}^*, \pi^* = (\pi^*_{\mathsf{trans}}, \pi^*_{\mathsf{key}}), tag^*)$ and $\mathsf{eph_sk}^*$. The reduction checks that $tag^* \notin \mathcal{Q}_2$ and that $\mathsf{KeyVerify}(\mathsf{eph_pk}^*, \pi^*, tag^*) = 1$; otherwise it outputs $\perp$. It then runs the extractor $\mathsf{Ext}_{\mathsf{key}}$ to obtain trans^* such that $\mathsf{ExchangeVerify}(\mathsf{trans}^*, \pi^*_{\mathsf{trans}}, tag^*) = 1$ and $\mathsf{eph_pk}^* = \mathsf{trans}^*[0]$.

If $tag^* \neq \tilde{tag}$ (equivalently, the forgery is not for the $\tilde{j}$-th KeyEscrow execution), then $\mathcal{R}$ aborts as in game $\mathbf{G}_4$. Otherwise, $\mathcal{R}$ defines i^* as in $\mathbf{G}_2$ from the extracted transcript trans^* and aborts if $\tilde{i} \neq i^*$. Finally, $\mathcal{R}$ outputs $(\mathsf{trans}^*, \mathsf{eph_sk}^*)$ to its APVSS challenger.

Correctness of the reduction. Conditioned on $\mathcal{R}$ not aborting, we have (i) $tag^* = \tilde{tag}$, so the oracle-generated component $\mathsf{trans}_{\tilde{i}}$ is embedded in the same simulated execution to which the forgery refers; and (ii) $\tilde{i} = i^* \in \mathsf{Contr}(\mathsf{trans}^*) \cap \mathcal{H}$, and thus $\mathsf{Contr}(\mathsf{trans}^*) \cap \mathcal{H} \neq \emptyset$. Moreover, $\mathsf{ExchangeVerify}(\mathsf{trans}^*, \pi^*_{\mathsf{trans}}, tag^*) = 1$ implies that trans^* is a valid aggregated PVSS transcript. Therefore, whenever $\mathcal{A}$ wins $\mathbf{G}_4$, the pair $(\mathsf{trans}^*, \mathsf{eph_sk}^*)$ satisfies all winning conditions of Definition 14. It follows that

$$\Pr[\mathbf{G}_4 \Rightarrow 1] \;\leq\; \Pr\big[\mathcal{R} \text{ wins } \textsc{AggUnpred}_{\mathsf{APVSS}}\big] + \mathsf{negl}(\lambda).$$

Since our APVSS [BL23] is $\varepsilon_{\mathsf{APVSS}}$-aggregated unpredictable [BL23], $\Pr\big[\mathcal{R} \text{ wins } \textsc{AggUnpred}_{\mathsf{APVSS}}\big]$ is neglgible, and thus

$$\Pr[\mathbf{G}_4 \Rightarrow 1] \leq \mathsf{negl}(\lambda).$$

Putting everything together. Combining the game transitions, we get

$$\Pr[\mathbf{G}_0 \Rightarrow 1] \;\leq\; q_{\mathsf{esc}} \cdot n \cdot \varepsilon_{\mathsf{APVSS}} + \mathsf{negl}(\lambda) \leq \mathsf{negl}(\lambda),$$

which concludes the proof.

□

Lemma 9 (Termination)**.** *Assume that: (i)* PVSSExchange *satisfies termination, (ii)* Disperse *satisfies termination, totality, and external validity with respect to the predicate* ExchangeVerify*, and (iii) the proof systems* $\mathsf{PS}_{\mathsf{key}}$ *and* $\mathsf{PS}_{\mathsf{share}}$ *are complete.*

Then, if all honest parties participate in an execution of KeyEscrow(tag) *for any tag, each honest party eventually outputs some pair* $(\mathsf{eph_pk}_i, \pi_i)$*. Moreover, if all honest parties subsequently participate in an execution of* KeyRetrieve(tag)*, then each honest party eventually outputs a value. In particular, the protocol tuple satisfies the termination property.*

Proof. Fix a tag.

Termination of KeyEscrow(tag)**.** Assume all honest parties invoke KeyEscrow(tag). By termination of PVSSExchange, at least one forever-honest party eventually completes the call to PVSSExchange(tag) and outputs $(\mathsf{trans}, \pi_{\mathsf{trans}})$. This party then invokes Disperse$(\mathsf{trans}, \pi_{\mathsf{trans}})$ with external validity ExchangeVerify$(\cdot, \cdot, tag)$. By termination of Disperse, it eventually outputs a dispersal proof π_{disp} and sends it to all.

Every honest party eventually receives a valid dispersal proof and, by totality of provable AVID, its ensuing participation in the corresponding private retrieval procedure PrivateRetrieve eventually yields an output $(\mathsf{trans}, \pi_{\mathsf{trans}})$. Hence, every honest party i eventually sets $\mathsf{trans}_i \neq \bot$. Once $\mathsf{trans}_i \neq \bot$, party i performs only local computations: it sets $\mathsf{eph_pk}_i := \mathsf{trans}_i[0]$, computes $\pi_{\mathsf{key},i} \leftarrow \mathsf{PProve}_{\mathsf{key}}((\mathsf{eph_pk}_i, \pi_{\mathsf{trans},i}, tag), \mathsf{trans}_i)$, and outputs

$$(\mathsf{eph_pk}_i, \pi_i) \quad \text{where} \quad \pi_i = (\pi_{\mathsf{trans},i}, \pi_{\mathsf{key},i}).$$

Therefore, every honest party eventually outputs in KeyEscrow(tag).

Termination of KeyRetrieve(tag)**.** Now assume all honest parties have completed the call to KeyEscrow(tag) and subsequently invoke KeyRetrieve(tag). Fix an honest party i. Since i completed KeyEscrow(tag), it holds that i obtained $(\mathsf{trans}_i, \pi_{\mathsf{trans},i})$ either directly from PVSSExchange(tag) or via PrivateRetrieve following a dispersal. In either case, by verifiability of PVSSExchange and external validity of Disperse with respect to ExchangeVerify$(\cdot, \cdot, tag)$, we have

$$\mathsf{ExchangeVerify}(\mathsf{trans}_i, \pi_{\mathsf{trans},i}, tag) = 1.$$

Party i then sets $\mathsf{eph_pk}_i := \mathsf{trans}_i[0]$ and computes

$$\pi_{\mathsf{key},i} \leftarrow \mathsf{PProve}_{\mathsf{key}}\big((\mathsf{eph_pk}_i, \pi_{\mathsf{trans},i}, tag), \mathsf{trans}_i\big),$$

which certifies (and, by completeness, is accepted) that $\mathsf{eph_pk}_i$ is the public key derived from a transcript trans_i whose PVSSExchange-verification succeeds.

During KeyRetrieve(tag), party i sends to each party j the share $\mathsf{share}_j := \mathsf{trans}_i[j]$ together with a proof

$$\pi_{\mathsf{share},i,j} \leftarrow \mathsf{PProve}_{\mathsf{share}}\big((\mathsf{eph_pk}_i, \pi_{\mathsf{key},i}, \mathsf{share}_j, j, tag), \mathsf{trans}_i\big),$$

which certifies that share_j is indeed the j-th transcript share of a transcript consistent with $\mathsf{eph_pk}_i$ and $\pi_{\mathsf{key},i}$. By completeness of $(\mathsf{PProve}_{\mathsf{share}}, \mathsf{PVerify}_{\mathsf{share}})$, every honest party j eventually receives this message and accepts it. Consequently, j responds with

$$\langle \text{“dec”}, S_j \rangle \quad \text{where} \quad S_j := \mathsf{ExtShare}(\mathsf{sk}_j, \mathsf{share}_j).$$

Party i eventually receives such responses from all honest parties and, for each honest j, verifies it as $\mathsf{ShareVer}(S_j, \mathsf{share}_j) = 1$ and stores (j, S_j) in shares_i. Since there are at least $n - f$ honest parties and $n - f \geq f + 1$, party i eventually accumulates $f + 1$ valid shares, after which it performs only local computation, reconstructs, outputs a value, and completes KeyRetrieve(tag).

Since i was arbitrary, every honest party eventually outputs a value from KeyRetrieve(tag). □

□

C.3 Weak Leader Election (Section 7)

Henceforth, we denote by $|\pi|$ the (maximum) size of our underlying zkSNARKs. While our zkSNARKs are instantiated with proof size $O(\lambda)$ [Fu25, AGL+23], our protocols admit alternative instantiations provided that the proof size remains $O(\lambda^2)$. Our complexity analysis is stated in terms of the proof length $|\pi|$ to keep the presentation generic and to accommodate such alternative instantiations.

Theorem 8 (Complexity)**.** *The* Election *protocol has $O((\lambda^2+|\pi|)n^2)$ communication complexity. In particular, when $|\pi| \in O(\lambda^2)$, this is $O(\lambda^2 n^2)$. In addition, the protocol has $O(1)$ round complexity.*

Proof. During the protocol, parties do the following:

- call the ProvableRank and KeyEscrow protocols, for a cost of $O((\lambda^2+|\pi|n^2)$ and $O(\lambda^2 n^2)$;
- send each other a constant number of messages that include public keys and encryptions of size $O(\lambda)$, proofs of size $O(|\pi|)$, and indices of size $O(\log n) = O(\lambda)$, for a total of $O(|\pi|n^2)$; and,
- call KeyRetrieve for a cost of $O(|\pi|n^2)$.

In total, the communication costs add up to $O((\lambda^2+|\pi|)n^2)$. In particular, when $|\pi| \in O(\lambda^2)$, this is $O(\lambda^2 n^2)$.

In addition, the protocol consists of a constant number of all-to-all rounds and constant-round protocols, yielding $O(1)$ round complexity. □

Theorem 9 (Security)**.** *The protocol* Election *(cf. Algorithm 6) is a Weak Leader Election protocol with $\alpha = \frac{1}{3}$ resilient to $f < n/3$ Byzantine parties, except with negligible probability in λ.*

Proof. We prove each property in a separate lemma and explicitly state the required properties of the underlying primitives. Although all primitives in our construction satisfy these requirements, this modular presentation highlights which assumptions are needed for each guarantee and facilitates replacing components with alternative implementations satisfying the same properties.

Lemma 10 (Termination)**.** *Assume that: (i)* (Encrypt, Decrypt) *is a correct encryption scheme (ii)* (KeyEscrow, KeyRetrieve) *is a Key Escrow protocol, and (iii)* ProvableRank *is a Provable Rank protocol.*

Then, if all honest parties participate in an execution of Election(tag)*, then each of them eventually completes the protocol and outputs an index. In particular, the protocol satisfies the termination property.*

Proof. Fix an arbitrary honest party i. Since all honest parties call Election(tag) with the same tag, by the code of Election they all invoke the sub-protocols

$$\mathsf{ProvableRank}((tag, \text{“rank”})) \quad \text{and} \quad \mathsf{KeyEscrow}((tag, \text{“key”})).$$

Step 1: Outputs of ProvableRank **and** KeyEscrow**.** By Termination and Correctness of ProvableRank and KeyEscrow, every honest party i eventually outputs

$$(r_i, \pi_{\mathsf{rank},i}) \leftarrow \mathsf{ProvableRank}((tag, \text{“rank”})) \quad \text{and}$$
$$(\mathsf{eph_pk}_i, \pi_{\mathsf{eph_pk},i}) \leftarrow \mathsf{KeyEscrow}((tag, \text{“key”})).$$

Moreover, by Correctness of ProvableRank and Provability of KeyEscrow, these outputs satisfy

$$\mathsf{RankVerify}(r_i, i, \pi_{\mathsf{rank},i}, (tag, \text{“rank”})) = 1 \quad \text{and}$$
$$\mathsf{KeyVerify}(\mathsf{eph_pk}_i, \pi_{\mathsf{eph_pk},i}, (tag, \text{“key”})) = 1.$$

Step 2: Dissemination of escrowed keys and rank encryptions. After outputting $(\mathsf{eph_pk}_i, \pi_{\mathsf{eph_pk},i})$, each honest party i sends a “key”-labeled message containing $(\mathsf{eph_pk}_i, \pi_{\mathsf{eph_pk},i})$ to all parties. By reliable delivery among honest parties, every honest party j eventually receives $(\mathsf{eph_pk}_i, \pi_{\mathsf{eph_pk},i})$.

Upon receiving such a “key” message, party j checks

$$\mathsf{KeyVerify}(\mathsf{eph_pk}_i, \pi_{\mathsf{eph_pk},i}, (tag, \text{“key”})) = 1.$$

If the check passes, party j responds to i with a "rank"-labeled message

$$\langle \text{“rank”},\ c_j,\ \pi_{\mathsf{rank},j} \rangle \quad \text{where} \quad c_j := \mathsf{Encrypt}(\mathsf{eph_pk}_i, r_j).$$

Because j is honest, its $(r_j, \pi_{\mathsf{rank},j})$ is the output of $\mathsf{ProvableRank}((tag, \text{“rank”}))$, hence as shown above, it satisfies

$$\mathsf{RankVerify}(r_j, j, \pi_{\mathsf{rank},j}, (tag, \text{“rank”})) = 1. \tag{1}$$

Since i is honest, it eventually receives such "rank" messages from every honest j. For each received $\langle \text{“rank”}, c_j, \pi_{\mathsf{rank},j} \rangle$, party i appends the tuple

$$(j, c_j, \pi_{\mathsf{rank},j})$$

to its local set ranks_i and replies with a "stored" message as specified by $\mathsf{Election}$.

Step 3: Progress to "ready" and invocation of $\mathsf{KeyRetrieve}$. There are at least $n - f$ honest parties. Hence every honest party eventually receives "stored" messages from at least $n - f$ honest parties, and therefore eventually sends its "ready" message to all. Similarly, every honest party eventually receives at least $n - f$ "ready" messages and thus invokes

$$\mathsf{KeyRetrieve}((tag, \text{“key”})).$$

By Termination and Correctness of $\mathsf{KeyRetrieve}$, every honest party i eventually outputs a secret key $\mathsf{eph_sk}_i$ that corresponds to its previously output public key $\mathsf{eph_pk}_i$ (recall that for each public key, there exists at most one such secret key); i.e., $(\mathsf{eph_pk}_i, \mathsf{eph_sk}_i)$ is a valid keypair for the encryption scheme used by $(\mathsf{Encrypt}, \mathsf{Decrypt})$.

Step 4: Collecting enough encrypted ranks. Because i appends one tuple to ranks_i for each "rank" response it receives, and because it receives such a response from every honest party, eventually

$$|\mathsf{ranks}_i| \geq n - f. \tag{2}$$

Step 5: Decryption, selection of a maximal verified rank, and well-definedness. After both events have occurred, namely: (i) i has output $\mathsf{eph_sk}_i$ from $\mathsf{KeyRetrieve}((tag, \text{“key”}))$, and (ii) (2) holds, party i performs the following local computation.

For every $(j, c_j, \pi_{\mathsf{rank},j}) \in \mathsf{ranks}_i$, it computes the candidate plaintext

$$\hat{r}_j := \mathsf{Decrypt}(\mathsf{eph_sk}_i, c_j)$$

and candidate ranks

$$\widehat{\mathsf{rank}}_j := \mathsf{H}(j, \hat{r}_j).$$

It then considers the subset of indices

$$S_i := \Big\{ j \ :\ (j, c_j, \pi_{\mathsf{rank},j}) \in \mathsf{ranks}_i \ \wedge\ \mathsf{RankVerify}(\hat{r}_j, j, \pi_{\mathsf{rank},j}, (tag, \text{“rank”})) = 1 \Big\}.$$

Finally, it chooses an index $\ell \in S_i$ such that $\widehat{\mathsf{rank}}_\ell$ is maximal among $\{\widehat{\mathsf{rank}}_j\}_{j \in S_i}$ (breaking ties in any fixed deterministic manner), and sets $\mathsf{rank}_\ell := \widehat{\mathsf{rank}}_\ell$.

We argue that S_i is nonempty, so ℓ is well-defined. Since $|\mathsf{ranks}_i| \geq n - f$ and there are at most f dishonest parties, ranks_i contains at least one tuple originating from some honest party j. For that honest j, we have $c_j = \mathsf{Encrypt}(\mathsf{eph_pk}_i, r_j)$ and $\pi_{\mathsf{rank},j}$ with (1). By correctness of encryption/decryption with the corresponding keypair $(\mathsf{eph_pk}_i, \mathsf{eph_sk}_i)$, we have

$$\mathsf{Decrypt}(\mathsf{eph_sk}_i, c_j) = r_j,$$

and therefore $\mathsf{RankVerify}(\mathsf{Decrypt}(\mathsf{eph_sk}_i, c_j), j, \pi_{\mathsf{rank},j}, (tag, \text{“rank”})) = 1$. Hence $j \in S_i$, implying $S_i \neq \emptyset$.

Step 6: Send to all parties and finalize. After selecting such an ℓ, party i sends to all parties an "elected"-labeled message

$$\langle \text{“elected”},\ \ell,\ \mathsf{rank}_\ell,\ r_\ell,\ \pi_{\mathsf{rank},\ell} \rangle.$$

Every honest party eventually receives "elected" messages from all honest parties. Upon receiving $\langle$"elected", $\ell, \mathsf{rank}_\ell, r_\ell, \pi_{\mathsf{rank},\ell}\rangle$, an honest party checks

$$\mathsf{RankVerify}(r_\ell, \ell, \pi_{\mathsf{rank},\ell}, (tag, \text{“rank”})) = 1 \quad \text{and} \quad \mathsf{rank}_\ell = \mathsf{H}(\ell, r_\ell),$$

and, if so, appends the corresponding tuple to its local set $\mathsf{candidates}$.

Since there are at least $n-f$ honest parties, every honest party eventually appends at least $n-f$ tuples to $\mathsf{candidates}$. At that point, by the specification of $\mathsf{Election}$, each honest party performs only local computation, outputs an index, and terminates.

Because the choice of i was arbitrary, every honest party terminates. □

Lemma 11 (α-Correctness)**.** *Assume that: (i)* $\mathsf{ProvableRank}$ *is a Provable Rank protocol, (ii)* ($\mathsf{KeyEscrow}$, $\mathsf{KeyRetrieve}$) *is a Key Escrow protocol, and (iii)* ($\mathsf{Encrypt}, \mathsf{Decrypt}$) *is a CPA-secure encryption scheme.*

Then with probability at least $\frac{1}{3}$, all honest parties that output an index output the same index $\ell \in [n]$, and the party indexed by ℓ was honest at the beginning of the protocol execution. In particular, the protocol satisfies the α-correctness property with $\alpha = \frac{1}{3}$.

Proof. We will use the following lemma. Recall that after the key-retrieval phase, each party j decrypts the set of stored ciphertexts and derives the (effective) ranks of other parties i as $\mathsf{rank}_i := \mathsf{H}(i, r_i)$, where r_i is the value output by $\mathsf{ProvableRank}$ and $\mathsf{H} : \{0,1\}^* \to \mathcal{R}$ is a random oracle. We want to argue that the ranks of honest parties remain hidden from the adversary. Since, by the security of the $\mathsf{ProvableRank}$ protocol, the values r_i generated by honest parties are computationally indistinguishable from uniform random to the adversary, the only information the adversary can obtain about an honest r_i during the execution of the protocol, prior to the key-retrieval phase, is through ciphertexts of the form $c_j = \mathsf{Encrypt}(\mathsf{ek}_j, r_i)$.

In this lemma, we show that the information the adversary obtains from these ciphertexts about any honest r_i is negligible, in the sense that it does not enable the adversary to compute or distinguish the corresponding rank $\mathsf{rank}_i := \mathsf{H}(i, r_i)$.

Lemma 12 (Rank-hiding before the first honest $\mathsf{KeyRetrieve}$)**.** *Let $\mathcal{H} \subseteq [n]$ denote the set of honest parties, and let T be the time at which the first honest party calls* $\mathsf{KeyRetrieve}((tag, \text{“key”}))$ *(equivalently, the first time an honest party has received $n-f$ "ready" messages and thus enters the key-retrieval phase). Then the adversary's view up to time T is computationally independent of the honest parties' effective ranks. More precisely, for $k := |\mathcal{H}|$, let $(U_1, \ldots, U_k)$ be i.i.d. uniform samples from the hash range $\mathcal{R}$. Then*

$$(\mathsf{rank}_i)_{i \in \mathcal{H}} \ \approx_c \ (U_1, \ldots, U_k) \quad \textit{with respect to } \mathsf{View}_{\mathcal{A},T},$$

i.e., no PPT adversary can distinguish the distribution of the honest parties' ranks from that of independent uniformly random values, given its entire view before time T, except with negligible probability in λ.

Proof. We prove the lemma via a sequence of hybrid games. Throughout, let H denote the random oracle used to derive the rank from the raw rank, i.e., in the real protocol party i's rank is

$$\mathsf{rank}_i := \mathsf{H}(i, r_i) \in \mathcal{R},$$

where r_i is the output of $\mathsf{ProvableRank}((tag, \text{“rank”}))$ (or equivalently, the value that is later recovered by decryption). Let $\mathsf{View}_{\mathcal{A},T}$ denote the adversary's entire view (internal state, randomness, and all delivered messages) restricted to the execution prefix strictly before time T.

Distinguishing experiment. For $b \in \{0,1\}$, define an experiment $\mathsf{Exp}(b)$ in which the adversary $\mathcal{A}$ interacts with the protocol execution up to time T and then receives a challenge vector $(X_1, \ldots, X_k)$ where:

- if $b = 0$, then $(X_1, \ldots, X_k) = (\mathsf{rank}_i)_{i \in \mathcal{H}}$ (the honest parties' real ranks);

- if $b = 1$, then $(X_1, \ldots, X_k) = (U_1, \ldots, U_k)$ where the U_j are sampled independently and uniformly at random $\mathcal{R}$.

Finally, $\mathcal{A}$ outputs a bit b', and we define its advantage as

$$\mathsf{Adv}_{\mathcal{A}} := \Big|\Pr[\mathsf{Exp}(0) \Rightarrow 1] - \Pr[\mathsf{Exp}(1) \Rightarrow 1]\Big|.$$

It suffices to show $\mathsf{Adv}_{\mathcal{A}} \leq \mathsf{negl}(\lambda)$.

Game $\mathbf{G}_0$ (real game). This is exactly $\mathsf{Exp}(0)$: run the real protocol up to time T with a true random oracle H, and at the end hand $\mathcal{A}$ the vector $(\mathsf{rank}_i)_{i \in \mathcal{H}}$.

Game $\mathbf{G}_1$ (program the rank hash at honest points). This game is identical to $\mathbf{G}_0$ except for how the random oracle H is answered on inputs of the form (i, r_i) for honest parties $i \in \mathcal{H}$, where r_i denotes the raw rank value output by $\mathsf{ProvableRank}((tag, \text{“rank”}))$ in the execution.

Concretely, for each honest party $i \in \mathcal{H}$, the challenger samples an independent uniform value $U_{i,tag} \leftarrow \mathcal{R}$ and *programs* the random oracle at the point (i, r_i) by setting

$$\mathsf{H}(i, r_i) := U_{i,tag}, \qquad \text{and defining } \mathsf{rank}_i := \mathsf{H}(i, r_i) = U_{i,tag}.$$

All other oracle queries are answered by lazy sampling as in a true random oracle, subject to consistency with the above programmed values.

Bad event. Let Bad be the event that the adversary $\mathcal{A}$ (or any corrupted party) queries the random oracle at some honest point (i, r_i) for the current execution tag *before* the challenger programs it, i.e.,

$$\mathsf{Bad} := \big[\exists\, i \in \mathcal{H} \text{ s.t. } \mathcal{A} \text{ queries } \mathsf{H}(i, r_i) \text{ at some time } < T\big].$$

Indistinguishability of $\mathbf{G}_0$ and $\mathbf{G}_1$ up to Bad. Conditioned on $\neg\mathsf{Bad}$, the programmed points (i, r_i) were never queried before time T, so $\mathbf{G}_1$ answers all oracle queries up to time T exactly as a true random oracle would. Therefore,

$$\Pr[\mathbf{G}_1 \Rightarrow 1 \mid \neg\mathsf{Bad}] = \Pr[\mathbf{G}_0 \Rightarrow 1 \mid \neg\mathsf{Bad}],$$

and hence, by a standard conditioning argument,

$$\big|\Pr[\mathbf{G}_1 \Rightarrow 1] - \Pr[\mathbf{G}_0 \Rightarrow 1]\big| \leq \Pr[\mathsf{Bad}].$$

It thus suffices to upper bound $\Pr[\mathsf{Bad}]$.

Game $\mathbf{G}_2$ (guess the first honest rank-hash query). Let Q_{H} be a polynomial upper bound on the number of random-oracle queries made by $\mathcal{A}$ before time T. The challenger samples an index $i^* \leftarrow \mathcal{H}$ uniformly at random and aborts unless the *first* query of the form $\mathsf{H}(i, r_i)$ made before time T is exactly $\mathsf{H}(i^*, r_{i^*})$. Let Good_1 denote the event that this guess is correct. Then $\Pr[\mathsf{Good}_1] \geq 1/|\mathcal{H}|$, and conditioned on Good_1, the adversary's view is distributed exactly as in Game 1. Therefore,

$$\Pr[\mathsf{Bad}] \leq |\mathcal{H}| \cdot \Pr[\mathsf{Bad}_2],$$

where Bad_2 is the event that $\mathcal{A}$ queries $\mathsf{H}(i^*, r_{i^*})$ before key retrieval and the game does not abort.

Fix the ciphertexts encrypting r_{i^*} that are sent by party i^* to the other parties:

$$c_{i^*,j} = (U_j,\, r_{i^*} \oplus \mathsf{H}(K_j)) \qquad \text{for } j \in [n],$$

where $U_j = g^{k_j}$ and $K_j = e(\mathsf{eph_pk}_j, h)^{k_j}$.

Game $\mathbf{G}_3$ (randomize i^*'s ciphertexts and program H on demand). Game $\mathbf{G}_3$ is identical to game $\mathbf{G}_2$ except for the way ciphertexts sent by the honest party i^* are generated and for how the random oracle H is answered on inputs of the form K_j.

Ciphertexts from parties $i \neq i^*$. For every honest sender $i \neq i^*$, the challenger generates all ciphertexts $c_{i,j}$ exactly as in the previous game.

Ciphertexts from i^*. For the ciphertexts sent by i^*, the challenger proceeds as follows. For each recipient $j \in [n]$, it samples fresh randomness k_j and sets $U_j := g^{k_j}$ as usual, but replaces the second component by a uniform pad:

$$c_{i^*,j} := (U_j, R_j) \qquad \text{where } R_j \leftarrow \mathcal{R}$$

(and $\mathcal{R}$ is the plaintext domain of r_{i^*}, with $\oplus$ denoting its group operation).

Programming H at K_j for consistency. The challenger maintains a table $\mathcal{T}_K$ initially empty. Upon a query $\mathsf{H}(x)$:

- if $x \notin \{K_j\}_{j\in[n]}$, answer by standard lazy sampling;
- if $x = K_j$ for some j and $\mathcal{T}_K$ does not yet contain an entry for K_j, then set

$$\mathsf{H}(K_j) := R_j \oplus r_{i^*}$$

  and record this value in $\mathcal{T}_K$; thereafter answer consistently using $\mathcal{T}_K[K_j]$.

All other protocol messages are simulated as in Game $\mathbf{G}_2$.

Justification. For each j, in the real protocol the second ciphertext component equals $r_{i^*} \oplus \mathsf{H}(K_j)$. In Game $\mathbf{G}_3$, the value R_j is uniform, and the first time $\mathsf{H}(K_j)$ is queried the challenger defines it so that $R_j = r_{i^*} \oplus \mathsf{H}(K_j)$. Since H is a random oracle, programming $\mathsf{H}(K_j)$ at its first query is perfectly consistent with the random-oracle distribution.

Defining the auxiliary bad event. Let Bad' be the event that the adversary queries $\mathsf{H}(K_j)$ for *some* recipient $j \in [n]$ before time T, i.e.,

$$\mathsf{Bad}' := \big[\exists\, j \in [n] \text{ s.t. } \mathcal{A} \text{ queries } \mathsf{H}(K_j) \text{ at some time } < T\big].$$

We will show $\Pr[\mathsf{Bad}'] = \mathsf{negl}(\lambda)$ soon. Assuming this bound for the moment, we obtain

$$\Pr[\mathsf{Bad}_2] \leq \Pr[\mathsf{Bad}_2 \wedge \neg\mathsf{Bad}'] + \Pr[\mathsf{Bad}'] = \Pr[\mathsf{Bad}_2 \wedge \neg\mathsf{Bad}'] + \mathsf{negl}(\lambda).$$

Bounding $\Pr[\mathsf{Bad}']$ via threshold signature unforgeability. We now bound $\Pr[\mathsf{Bad}']$ by a reduction to the unforgeability of the underlying threshold signature scheme.

Key observation: K_j is a threshold-signature component. Observe that for each ciphertext $c_{i^*,j} = (U_j,\, r_{i^*} \oplus \mathsf{H}(K_j))$, we have $U_j = g^{k_j}$ and

$$K_j \;=\; e(\mathsf{eph_pk}_j, h)^{k_j} \;=\; e(g,h)^{\mathsf{eph_sk}_j \cdot k_j} \;=\; e(g^{k_j}, h^{\mathsf{eph_sk}_j}) \;=\; e(U_j, \mathsf{eph_sk}_j).$$

By definition of the threshold signature scheme used in our construction, the value $e(U_j, \mathsf{eph_sk}_j)$ is precisely the *unique signature component* σ' associated with verification key $\mathsf{eph_vk}_j$ and message U_j (in the threshold signature, a message $m \in \{0,1\}^*$ is signed by its representative $\mathsf{H}_1(m) \in \mathbb{G}_1$, see Section 4.5). We recall that the authors of [BLL+24] show that computing this unique signature component σ' is computationally hard under the one-more discrete logarithm assumption. In our secrecy proof, we reduced occurrence of a bad event "Bad" to the unforgeability of this unique component, and we can apply a very similar reduction here.

Recall unforgeability game. We briefly recall the respective game. The game is played between a challenger $\mathcal{U}$ and an adversary $\mathcal{A}'$. The challenger generates plain public keys for all n parties and gives them to $\mathcal{A}'$; the adversary specifies an initial set of corrupted parties and may replace their public keys, which the challenger adopts. Throughout the game, $\mathcal{A}'$ has access to a corruption oracle and may corrupt at most f parties in total. For threshold key setup, $\mathcal{U}$ samples an honest PVSS contribution transcript trans_{j^*} for an honest party j^* that $\mathcal{A}'$ specifies, and subsequently gives trans_{j^*} to $\mathcal{A}'$. The adversary then specifies an aggregated transcript trans^* that includes trans_{j^*} as a contribution, ensuring that trans^* contains at least one honest contribution; this transcript defines the threshold secret key $\mathsf{eph_sk}$ and the corresponding threshold public

key $\mathsf{eph_pk}$. The adversary may query a partial-signature oracle on any message m and any honest party, and has access to a random oracle H_1 mapping messages to group elements. At the end of the game, $\mathcal{A}'$ outputs a pair (m^*, σ'^*) and wins if $\sigma'^* = e(\mathsf{H}_1(m^*), \mathsf{eph_sk})$ is the unique signature component for m^*, and m^* was at most called $f - |\mathcal{C}|$ times to the partial signature oracle.

Forger $\mathcal{F}$. We construct a PPT forger $\mathcal{F}$ that interacts with $\mathcal{U}$ and simulates game $\mathbf{G}_3$ for our adversary $\mathcal{A}$, while using Bad' to output a forgery to $\mathcal{U}$.

Guesses. $\mathcal{F}$ makes the following uniform guesses:

- an index $j^* \leftarrow [n]$, intended to be the recipient for which $\mathcal{A}$ makes an early query $\mathsf{H}(K_{j^*})$ witnessing Bad';
- a query index $q^* \leftarrow [Q_{\mathsf{H}}]$, intended to be the index of the *first* random-oracle query whose input equals K_{j^*}; and
- an index $k^* \leftarrow [n]$, intended to be an honest contributor whose PVSS Exchange contribution appears in the aggregated transcript trans^* that defines j^*'s escrow key.

Let Good denote the event that all guesses are correct.

Embedding an honest contribution into the escrow transcript. Let $\mathsf{eph_pk}_{j^*}$ be the escrow public key used by party j^*, and let trans^* denote the aggregated PVSS transcript output by the execution of $\mathsf{KeyEscrow}((tag, \text{“key”}))$ that fixes $\mathsf{eph_pk}_{j^*}$. By the guarantees of $\mathsf{PVSSExchange}$ together with dispersal and external validity, the transcript trans^* contains at least one honest contribution. Conditioned on the event Good, this honest contribution is exactly the one of party k^*.

$\mathcal{F}$ interacts with the threshold-signature unforgeability challenger $\mathcal{U}$ as follows. First, as prescribed by the unforgeability game, $\mathcal{U}$ provides $\mathcal{F}$ with an honestly generated contribution transcript trans_{k^*} for party k^*. $\mathcal{F}$ embeds this transcript into the simulation of $\mathsf{KeyEscrow}((tag, \text{“key”}))$ by using trans_{k^*} as the honest contribution of party k^* in that execution. All other protocol messages and contributions are simulated consistently with the real execution and with the adversary's behavior.

The adversary $\mathcal{A}$ eventually determines an aggregated transcript trans^* as the output of the $\mathsf{KeyEscrow}$ protocol. Conditioned on Good, this transcript indeed includes trans_{k^*} as one of its components. At this point, $\mathcal{F}$ forwards the resulting aggregated transcript trans^* to the challenger $\mathcal{U}$, which adopts it as the threshold key setup transcript and thereby fixes the corresponding threshold secret key and the escrow public key $\mathsf{eph_pk}_{j^*}$. From this point on, $\mathcal{F}$ uses this $\mathsf{eph_pk}_{j^*}$ as party j^*'s escrow public key in the simulated execution.

Simulating the ciphertexts. $\mathcal{F}$ simulates the Election execution up to time T for $\mathcal{A}$.

Honest senders $i \neq i^*$. For every honest sender $i \neq i^*$, $\mathcal{F}$ generates all ciphertexts $c_{i,j}$ exactly as in the real protocol. The corresponding raw ranks r_i produced by $\mathsf{ProvableRank}((tag, \text{“rank”}))$ are uniform and independent from r_{i^*}, guaranteed by the Provable Rank properties.

Special honest sender $i = i^*$. For the ciphertexts sent by i^*, $\mathcal{F}$ distinguishes two cases.

- **Recipients $j \neq j^*$.** For each such recipient, $\mathcal{F}$ samples $k_j \leftarrow \mathbb{Z}_q$, sets $U_j := g^{k_j}$, samples R_j uniformly from the plaintext space of r_{i^*} (denoted $\mathcal{R}$), and sends
$$\widetilde{c}_{i^*,j} := (U_j, R_j).$$
- **Recipient $j = j^*$.** $\mathcal{F}$ samples a fresh message m^* uniformly from the message space of the threshold signature scheme and queries the random oracle H_1 provided by the unforgeability challenger $\mathcal{U}$ to obtain $\mathsf{H}_1(m^*) \in \mathbb{G}_1$. It then sets
$$U_{j^*} := \mathsf{H}_1(m^*)$$
and samples $R_{j^*} \leftarrow \mathcal{R}$ uniformly at random. It then sends the ciphertext
$$\widetilde{c}_{i^*,j^*} := (U_{j^*}, R_{j^*}).$$

Conditioned on Good, this perfectly simulates game $\mathbf{G}_3$.

Bad event and extraction. If Bad' occurs, then the adversary has produced the group element

$$K_{j^*} = e(U_{j^*}, \mathsf{eph_sk}_{j^*}) = e(\mathsf{H}_1(m^*), \mathsf{eph_sk}_{j^*})$$

(as the q^*-th random oracle input) before time T. In this case, $\mathcal{F}$ aborts the simulation and outputs the pair

$$(m^*, K_{j^*})$$

to the unforgeability challenger $\mathcal{U}$.

Why this is a valid forgery. By construction, $K_{j^*} = e(\mathsf{H}_1(m^*), \mathsf{eph_sk}_{j^*})$ is exactly the unique signature component for message m^* under the challenge threshold secret key defined by trans^* in the unforgeability game. Moreover, $\mathcal{F}$ never queried the partial-signature oracle on m^* (and does not know $\mathsf{eph_sk}_{j^*}$), hence (m^*, K_{j^*}) constitutes a successful forgery whenever Bad' occurs.

Success probability. Whenever $\mathsf{Bad}' \wedge \mathsf{Good}$ occurs, $\mathcal{F}$ outputs a successful forgery. Therefore,

$$\Pr[\mathsf{Bad}' \wedge \mathsf{Good}] \leq \mathsf{Adv}^{\mathrm{uf}}_{\mathrm{TS}}(\mathcal{F}).$$

Since $\Pr[\mathsf{Good}] \geq 1/(n^2 Q_{\mathsf{H}})$, we obtain

$$\Pr[\mathsf{Bad}'] \cdot \frac{1}{n^2 Q_{\mathsf{H}}} \;\leq\; \mathsf{Adv}^{\mathrm{uf}}_{\mathrm{TS}}(\mathcal{F}) \;=\; \mathsf{negl}(\lambda),$$

and hence $\Pr[\mathsf{Bad}'] \leq \mathsf{negl}(\lambda)$, as required.

Conclusion (bounding $\Pr[\mathsf{Bad}]$ and finishing the indistinguishability). We now complete the proof by bounding $\Pr[\mathsf{Bad}]$ from Game $\mathbf{G}_1$.

Fix the hybrid sequence above and recall from Game $\mathbf{G}_2$ that

$$\Pr[\mathsf{Bad}] \leq |\mathcal{H}| \cdot \Pr[\mathsf{Bad}_2],$$

where Bad_2 is the event that $\mathcal{A}$ queries $\mathsf{H}(i^*, r_{i^*})$ at some time $< T$ in the non-aborting execution.

Consider Game $\mathbf{G}_3$, where all ciphertexts sent by i^* have uniformly random second components $R_j \leftarrow \mathcal{R}$, and $\mathsf{H}(K_j)$ is programmed on demand to maintain consistency. Define Bad' as above. Conditioned on $\neg\mathsf{Bad}'$, none of the points K_j is queried before time T. In particular, in Game $\mathbf{G}_3$ the values $\{R_j\}_{j\in[n]}$ are independent uniform pads from $\mathcal{A}$'s viewpoint before time T. Therefore, conditioned on $\neg\mathsf{Bad}'$, the adversary's view up to time T is independent of r_{i^*}. Consequently, before time T the only way for $\mathcal{A}$ to query $\mathsf{H}(i^*, r_{i^*})$ is to guess r_{i^*}.

Conditioned on $\neg\mathsf{Bad}'$, the adversary's view up to time T is independent of r_{i^*}, and r_{i^*} is uniformly distributed over $\mathcal{R}$. Therefore, for any fixed random-oracle query made by $\mathcal{A}$ before time T, the probability that this query equals $\mathsf{H}(i^*, r_{i^*})$ is exactly $1/|\mathcal{R}|$.

By a union bound over the at most Q_{H} random-oracle queries made before time T, we obtain

$$\Pr[\mathsf{Bad}_2 \wedge \neg\mathsf{Bad}'] \;\leq\; \frac{Q_{\mathsf{H}}}{|\mathcal{R}|}.$$

Since $|\mathcal{R}|$ is exponential in λ, this probability is negligible. Combining with the bound $\Pr[\mathsf{Bad}'] = \mathsf{negl}(\lambda)$ established above yields

$$\Pr[\mathsf{Bad}_2] \;\leq\; \Pr[\mathsf{Bad}_2 \wedge \neg\mathsf{Bad}'] + \Pr[\mathsf{Bad}'] \;\leq\; \frac{Q_{\mathsf{H}}}{|\mathcal{R}|} + \mathsf{negl}(\lambda) \;=\; \mathsf{negl}(\lambda).$$

Hence, we get

$$\Pr[\mathsf{Bad}] \leq |\mathcal{H}| \cdot \Pr[\mathsf{Bad}_2] = \mathsf{negl}(\lambda).$$

It follows from the conditioning argument in Game $\mathbf{G}_1$ that

$$\left|\Pr[\mathbf{G}_1 \Rightarrow 1] - \Pr[\mathbf{G}_0 \Rightarrow 1]\right| \leq \Pr[\mathsf{Bad}] = \mathsf{negl}(\lambda).$$

I.i.d. uniformity in Game $\mathbf{G}_1$. In Game $\mathbf{G}_1$, for each honest $i \in \mathcal{H}$ the challenger samples an independent uniform $U_{i,tag} \leftarrow \mathcal{R}$ and sets $\mathsf{H}(i, r_i) := U_{i,tag}$, defining $\mathsf{rank}_i := U_{i,tag}$. Since the programmed inputs (i, r_i) are all distinct (because the first component i is distinct), the values $\{\mathsf{rank}_i\}_{i\in\mathcal{H}}$ are jointly independent and uniformly distributed over $\mathcal{R}^k$. Therefore, the challenge vector given to $\mathcal{A}$ in Game $\mathbf{G}_1$ is distributed identically to $(U_1, \ldots, U_k) \leftarrow \mathcal{R}^k$.

Final step. Since $\mathbf{G}_0$ and $\mathbf{G}_1$ are computationally indistinguishable and $\mathbf{G}_1$ is identical to $\mathsf{Exp}(1)$, we conclude that $\mathsf{Adv}_{\mathcal{A}} \leq \mathsf{negl}(\lambda)$, i.e.,

$$(\mathsf{rank}_i)_{i\in\mathcal{H}} \approx_c (U_1, \ldots, U_k) \quad \text{with respect to } \mathsf{View}_{\mathcal{A},T}.$$

This completes the proof of the lemma. □

We can now proceed with the proof of α-correctness. Before any honest party outputs a value from the protocol, it receives "elected" messages from $n - f$ parties, at least one of which was sent by a honest party. Honest parties send such a message after outputting a key from the KeyRetrieve protocol, which they call after receiving "ready" messages from $n - f$ parties. From the Secrecy of the ProvableRank protocol, the adversary learns nothing about r_i for any honest i from the execution of ProvableRank.

During the Election protocol ,the adversary also receives encryption $c = \mathsf{Encrypt}(\mathsf{eph_pk}, r_i)$ for honest parties i after providing a key eph_pk and a proof π such that $\mathsf{KeyVerify}(\mathsf{eph_pk}, \pi, (tag, \text{"key"})) = 1$. From Lemma 12, before the first honest party receives "ready" messages from $n - f$ parties, the adversary's view is independent of the honest parties' ranks $\mathsf{rank}_i = \mathsf{H}(i, r_i)$.

Let H be the set of honest parties[18] that sent "ready" messages before the first honest party called KeyRetrieve, and note that $|H| \geq n - 2f$. Let R be the set of all parties for which ranks with verifying proofs are ever sent to honest parties throughout the protocol. More precisely, for every index $j \in R$, some honest party receives $j, \mathsf{rank}_j, r_j, \pi_{\mathsf{rank},j}$ in an "elected" message, or an encryption of r_j from j in a "rank" message. Below, we will refer to the ranks of parties in R, these are the only ranks for which the adversary can provide proofs, as defined in the Uniqueness of the ProvableRank protocol.

Consider the event G in which the maximal rank among all parties in R belongs to a honest party ℓ^* in H. First, note that since the honest parties' ranks are independent of the adversary's view before the first honest party calls KeyRetrieve (and H is defined before that), H is defined independently of the ranks of the parties in H. From the Validity of the ProvableRank protocol, each one of the ranks of parties in R is uniform and independent of all other parties' ranks. Therefore, each party in H has a $\frac{1}{n}$ probability of having the maximal rank among all parties in R. Since $|H| \geq n - 2f$ and $|R| < n$, the event G takes place with probability at least $\frac{n-2f}{n} \geq \frac{1}{3} = \alpha$.

Finally, we would like to show that if the event G takes place, all honest parties will output the index ℓ^* of a party that was honest when it sent its "ready" message, and was thus also honest at the beginning of the protocol. Assume the event G takes place. In that case, ℓ^* sent a "ready" message before the first honest party called KeyRetrieve. Before doing so, it received a "stored" message from $n - f$ parties. At least $f + 1$ of those parties are honest, and they stored $(\ell^*, c_{\ell^*}, \pi_{\mathsf{rank},\ell^*})$ in their respective ranks sets. If these parties send "elected" messages, they do so after calling KeyRetrieve and outputting their eph_sk keys. They then decrypt all the encrypted ranks they received, check which ones have verifying proofs, and pick the index of the party with the maximal verifying rank. They all stored $(\ell^*, c_{\ell^*}, \pi_{\mathsf{rank},\ell^*})$ in their ranks set before calling KeyRetrieve, and by assumption this ℓ^* has the maximal verifying rank among all parties in R, and thus every one of the above parties find ℓ^* have the maximal rank. Therefore, if any of these $f + 1$ parties sends and "elected" message, it does so with $\ell^*, \mathsf{rank}_{\ell^*}, r_{\ell^*}, \pi_{\mathsf{rank},\ell^*}$.

Before outputting a value from the protocol, honest parties wait to receive "elected" messages with verifying ranks from $n - f$ parties. Any party that does so receives at least one message from the $f + 1$

[18] Technically, they might be corrupted later. Here we mean the parties that were honest when they sent such messages.

parties that sent $\langle$"elected", $\ell^*, \mathsf{rank}_{\ell^*}, r_{\ell^*}, \pi_{\mathsf{rank},\ell^*}\rangle$. Since the event G takes place, no party will receive a higher verifying rank. Thus, every honest party that completes the protocol will see that rank_{ℓ^*} is the maximal verifying rank, and output the index ℓ^* of a party that was honest when starting the protocol.

□

□