Chainlink DKG: A Versatile Distributed Key Generation Protocol

Lorenz Breidenbach* Christian Cachin† Yan Ji
Kostis Karantias Gregory Neven Philipp Schindler
Chrysoula Stathakopoulou

July 25, 2026

Abstract

We present a practical, versatile, and easy to implement distributed key generation (DKG) protocol for discrete-logarithm keys that offers strong, composable security and acceptable performance for real-world committee sizes and network conditions. The protocol cleanly separates the *dealers*, who contribute randomness or existing shares, from the *recipients*, who obtain the output shares, enabling deployment patterns where the protocol participants differ from the eventual key holders. It supports fresh key generation as well as resharing across epochs and committees, and accommodates a wide range of reconstruction and corruption thresholds, including high-threshold and dual-threshold regimes relevant to threshold-signature applications. We formalize security via a natural ideal DKG functionality that samples a uniform sharing polynomial, eliminating any adversarial bias in the resulting public key and shares. This simple notion enables secure composition with a wide variety of primitives and schemes, allowing our protocol to act as a secure drop-in replacement for a trusted key generation algorithm. Our DKG protocol is formulated over an arbitrary total-order broadcast (consensus) layer; we describe an implementation as a reporting plugin atop Chainlink's OCR 3.1 protocol.

1 Introduction

Distributed key generation (DKG) is the foundation that lets a set of mutually distrustful parties create a shared public key with private key material split across them. In this work we present a versatile DKG for a Shamir secret

*Authors in alphabetical order.

†The author is a faculty member at University of Bern. He co-authored this work in his separate capacity as an advisor to Chainlink Labs.

sharing [Sha79] of discrete-logarithm keys that is deliberately engineered for practical deployments: it favors cryptographic simplicity over use-case-specific optimizations, targets the network conditions and committee sizes that we actually see in production at Chainlink, and comes with strong, composable security guarantees that make it a safe default in a wide range of applications.

Our design philosophy is to keep the cryptography as simple to implement and auditable as possible, with minimal external dependencies; we prefer building from first principles over importing heavy stacks. We particularly want to avoid re-solving consensus: modern consensus protocols are a commodity, so our DKG is built as a reporting plugin for Chainlink's Offchain Reporting (OCR) 3.1 protocol [BCC+25, BCK+24], treating consensus as an external service rather than as part of the DKG itself.

Performance is not the primary goal; the protocol only needs to be "reasonably efficient" for occasional invocations, say, at most once every 15 minutes, on committees of size $n \in [16, 64]$.

With respect to network assumptions, we target partial synchrony and aim for robustness under asynchronous scheduling; a protocol that is only secure in a fully synchronous communication model is not acceptable, as it doesn't capture the reality of a wide-area distributed system.

On the security side, we insist on strong and composable security proofs. We aim for versatility, ensuring secure composition with virtually any target application. In particular, this means that neither the joint secret nor the individual shares can admit biases that would force a per-use-case analysis about its safe use in a particular setting.[1]

We also target versatility in terms of thresholds: we want the protocol to support (re-)sharing of secrets under any reconstruction thresholds $t < n$ tolerating any number of corrupt parties $f < t$. This includes what is often referred to as high-threshold ($t > 2n/3$, $f = t - 1$) and dual-threshold ($t > 2n/3$, $f < n/3$) settings, the latter finding wide application for threshold signatures in consensus protocols. For integration with consensus, we typically assume $t < n - f$.

From a functional perspective, the protocol must support a clean separation between *dealers*, who contribute randomness or inputs to the protocol, and *recipients*, who receive the shares. This enables deployment patterns where the set of parties that runs the protocol differs from the set that later holds the key.

We further build in life-cycle features needed in long-running systems. The protocol supports resharing to the same committee (e.g., to obtain proactive security), to a different committee (e.g., to support churn in committee membership), or even across networks to replicate or back up a key. We want to achieve proactive security [HJKY95], which is closely related to forward security, whereby a compromise of party $\mathcal{P}$ in epoch T does not retroactively reveal

[1] For example, the bias inherent in Pedersen's [Ped92] DKG protocol and many successors is benign for Schnorr threshold signatures [GJKR07] but renders ECDSA threshold signatures insecure [GS22a].

$\mathcal{P}$'s shares from epochs earlier than T.

Finally, the cryptographic embedding of the construction must be agnostic of the elliptic curve that is used and must be easy to instantiate over both pairing-friendly and non-pairing curves, so that deployments can switch elliptic curves (or groups) without redesign.

Our DKG protocol. Our protocol is not new; it is essentially the "secret bulletin board"-based construction from Boneh and Shoup [BS23, Section 22.4.2] instantiated with Shoup's recent simple verifiably encrypted secret sharing (VESS) scheme [Sho25]. The latter is itself a variation on known cut-and-choose verifiable encryption techniques [ASW00]. We mold the protocol to accommodate the separation between dealers and recipients and to run on top of any total-order broadcast channel among the dealers.

We then prove that this protocol realizes what is probably the most natural ideal functionality for DKG protocols: it simply hands each recipient an evaluation of a uniformly random polynomial. [2]

Finally, we describe how the protocol can be implemented as a reporting plugin on top of the OCR 3.1 consensus protocol.

2 Preliminaries

For an integer $n \in \mathbb{N}$, we write $[n] = \{1, \ldots, n\}$ to denote the range of integers from 1 to n.

Let $\kappa \in \mathbb{N}$ be the (symmetric-key equivalent) security parameter. All algorithms below take 1^κ as an implicit input. Algorithms are said to be "efficient" if they are probabilistic polynomial time in κ. A function is said to be "negligible" if it is asymptotically smaller than the inverse of any polynomial in κ.

2.1 Secret Sharing

Vectors and polynomials. Let $\mathbb{G}$ be a multiplicative cyclic group of prime order p with generator g. We adopt and extend the useful notation from [GS22b] to

[2] The same functionality (modulo the separation between dealers and recipients) was described by Katz [Kat24] as an "alternate" or "more natural" functionality to their main functionality that lets the adversary choose corrupt parties' shares. Katz argues that the alternate functionality does not seem to have any practical disadvantages, while potentially allowing for more efficient protocols. We would argue that it's hard to predict future uses of the protocol; in fact, it's easy to come up with example applications that are secure with the alternate functionality but insecure with their main functionality, e.g., a lottery where the recipient with the smallest share or public key wins. Moreover, our protocol is sufficiently efficient for the low-throughput settings that we envisage, so we prefer strong security and application flexibility over better efficiency.

denote vectors of scalars $\boldsymbol{s} = (s_0, \ldots, s_{n-1})$ and group elements $\boldsymbol{g} = (g_0, \ldots, g_{n-1})$ using boldface, letting

$$g^{\boldsymbol{s}} \;=\; (g^{s_0}, \ldots, g^{s_{n-1}}) \;,$$

and, for $\beta \in \mathbb{Z}_p$, by letting

$$\boldsymbol{g}^{\beta} \;=\; (g_0^{\beta}, \ldots, g_{n-1}^{\beta})$$

and

$$\boldsymbol{g}^{(\beta)} \;=\; \prod_{i=0}^{n-1} g_i^{\beta^i} \;.$$

For equal-size vectors of group elements $\boldsymbol{g}, \boldsymbol{h}$, we define the component-wise group operation

$$\boldsymbol{g} \cdot \boldsymbol{h} \;=\; (g_0 \cdot h_0, \ldots, g_{n-1} \cdot h_{n-1}) \;;$$

the component-wise inverse operator / is defined analogously.

We occasionally overload notation by letting $\boldsymbol{\omega}(X) \in \mathbb{Z}_p[X]$ denote the polynomial with coefficients taken from the vector $\boldsymbol{\omega}$, i.e.,

$$\boldsymbol{\omega}(X) \;=\; \omega_0 + \omega_1 \cdot X + \ldots + \omega_{n-1} \cdot X^{n-1} \in \mathbb{Z}_p[X] \;.$$

If $\boldsymbol{x} = (x_0, \ldots, x_{m-1}) \in \mathbb{Z}_p^m$, we define $\boldsymbol{\omega}(\boldsymbol{x})$ to be the vector of values of the polynomial $\boldsymbol{\omega}(X)$ evaluated at the values in $\boldsymbol{x}$, i.e.,

$$\boldsymbol{\omega}(\boldsymbol{x}) = (\boldsymbol{\omega}(x_0), \ldots, \boldsymbol{\omega}(x_{m-1})) \;.$$

Lagrange interpolation. For any set of n distinct evaluation points $\boldsymbol{e} = (e_0, \ldots, e_{n-1}) \in \mathbb{Z}_p^n$, the Lagrange basis with respect to $\boldsymbol{e}$ consists of the unique polynomials $\boldsymbol{\Lambda}_{\boldsymbol{e},e_i}(X) \in \mathbb{Z}_p[X]$, $i = 0, \ldots, n-1$, so that $\boldsymbol{\Lambda}_{\boldsymbol{e},e_i}(e_i) = 1$ and $\boldsymbol{\Lambda}_{\boldsymbol{e},e_i}(e_j) = 0$ for all $j \neq i$, i.e.,

$$\boldsymbol{\Lambda}_{\boldsymbol{e},e_i}(X) \;=\; \prod_{j=0, j\neq i}^{n-1} \frac{X - e_j}{e_i - e_j} \;.$$

If $y_i = \boldsymbol{\omega}(e_i)$ for all $i = 0, \ldots, n-1$, one can rewrite $\boldsymbol{\omega}(X)$ as a linear combination of the Lagrange basis as

$$\boldsymbol{\omega}(X) \;=\; \sum_{i=0}^{n-1} y_i \cdot \boldsymbol{\Lambda}_{\boldsymbol{e},e_i}(X) \;,$$

which is easily seen from the fact that the right-hand side is the unique polynomial of degree at most $n-1$ that evaluates to y_i in e_i for all $i = 0, \ldots, n-1$.

Shamir secret sharing. Shamir's celebrated secret sharing scheme [Sha79] shares a secret $s \in \mathbb{Z}_p$ over n parties $\boldsymbol{\mathcal{P}}$ so that any coalition of at least t parties

in $\boldsymbol{\mathcal{P}}$ can reconstruct the secret, but any smaller coalition learns nothing about the secret.

We assume that each party $\mathcal{P} \in \boldsymbol{\mathcal{P}}$ is uniquely assigned to an element of $\mathbb{Z}_p \setminus \{0\}$; this element could be defined by its index in the list $\boldsymbol{\mathcal{P}}$, by the party's public key, or by any other means. We will interchangeably use $\mathcal{P}$ to refer to both the party and its assigned element of $\mathbb{Z}_p \setminus \{0\}$.

Shamir's secret sharing scheme lets the secret owner choose a random polynomial of degree $t-1$

$$\boldsymbol{\omega}(X) = \omega_0 + \omega_1 \cdot X + \cdot + \omega_{t-1} \cdot X^{t-1} \in \mathbb{Z}_p[X]$$

with $\boldsymbol{\omega}(0) = \omega_0 \leftarrow s$. It then hands to party $\mathcal{P}$ the secret share $s_{\mathcal{P}} \leftarrow \boldsymbol{\omega}(\mathcal{P})$.

Let $\boldsymbol{\mathcal{P}}' \subseteq \boldsymbol{\mathcal{P}}$ be a subset of t parties from $\boldsymbol{\mathcal{P}}$. Then the parties in $\boldsymbol{\mathcal{P}}'$ can jointly reconstruct the secret as

$$s \leftarrow \sum_{\mathcal{P} \in \boldsymbol{\mathcal{P}}'} s_{\mathcal{P}} \cdot \boldsymbol{\Lambda}_{\boldsymbol{\mathcal{P}}', \mathcal{P}}(0) \ .$$

2.2 Hardness Assumptions

Definition 1 (Discrete logarithms) *Let $\mathbb{G}$ be a cyclic multiplicative group of prime order p with generator $g \in \mathbb{G}$.* [3] *Consider the following game with an adversary $\mathcal{A}$:*

- *The experiment chooses $Y \leftarrow_{\$} \mathbb{G}$ and runs $\mathcal{A}$ on input Y.*
- *The adversary outputs $x \in \mathbb{Z}_p$ and wins the game if $Y = g^x$.*

The advantage $\mathbf{Adv}^{\mathsf{dlog}}_{\mathbb{G},\mathcal{A}}(\kappa)$ is the probability that $\mathcal{A}$ wins the game. The discrete logarithm assumption is said to hold if $\mathbf{Adv}^{\mathsf{dlog}}_{\mathbb{G},\mathcal{A}}(\kappa)$ is negligible for all efficient adversaries $\mathcal{A}$.

Boneh and Shoup [BS23] introduced the interactive computational Diffie-Hellman (ICDH) assumption below as a refutable replacement for the (non-refutable) assumptions that computational Diffie-Hellman (CDH) remains hard when the adversary is given access to a decisional Diffie-Hellman (DDH) oracle.

Definition 2 (Interactive computational Diffie-Hellman [BS23]) *The one- or two-oracle interactive computational Diffie-Hellman (ICDH1 or ICDH2) problem in a cyclic group $\mathbb{G}$ of prime order p with generator $g \in \mathbb{G}$ is defined through the following game with an adversary $\mathcal{A}$:*

[3] For asymptotic security, we actually consider the group $\mathbb{G}$ as being selected from a family of groups $\mathbb{G}(\kappa)$; we simply write $\mathbb{G}$ for ease of notation.

- *The experiment chooses* $a, b \leftarrow_\$ \mathbb{Z}_p$*, computes* $A \leftarrow g^a$ *and* $B \leftarrow g^b$*, and runs* $\mathcal{A}$ *on input* (A, B)*.*
- *In the ICDH1 problem, the adversary can make an arbitrary number of queries to a Diffie-Hellman oracle* $\mathsf{DDH1}$ *that, on input* (X, Z)*, outputs whether* $X^a = Z$*.*
- *In the ICDH2 problem, the adversary additionally has access to an oracle* $\mathsf{DDH2}$ *that, on input* (X, Z)*, outputs whether* $X^b = Z$*.*
- *The adversary outputs* $C \in \mathbb{G}$ *and wins if* $C = g^{ab}$*.*

The advantage functions $\mathbf{Adv}^{\mathsf{icdh1}}_{\mathbb{G},\mathcal{A}}(\kappa)$ *and* $\mathbf{Adv}^{\mathsf{icdh2}}_{\mathbb{G},\mathcal{A}}(\kappa)$ *are given by the probability that* $\mathcal{A}$ *wins the ICDH1 and ICDH2 game above, respectively. We say that the ICDH1 and ICDH2 assumptions hold if their respective advantages are negligible for all efficient adversaries* $\mathcal{A}$*.*

2.3 Universal Composability

In the universal composability (UC) framework [Can01], the security of a protocol Π is defined by comparing its execution in the *real world* with the execution of an *ideal functionality* $\mathcal{F}$ in the *ideal world*.

The key idea is that an external environment $\mathcal{E}$ interacts with the parties executing the protocol and with the adversarial entity (either the real-world adversary $\mathcal{A}$ or the ideal-world simulator $\mathcal{S}$). The environment provides inputs to the parties, receives their outputs, and can exchange arbitrary messages with the adversary. In this way, $\mathcal{E}$ models any possible external context in which the protocol might be executed, and the adversary models any possible attack strategy.

In the real world, the adversary $\mathcal{A}$ controls the scheduling and delivery of messages between protocol participants, can corrupt parties, and can interact freely with the environment. We write $\mathbf{Real}^{\Pi}_{\mathcal{E},\mathcal{A}}(\kappa)$ to denote the probability that the environment $\mathcal{E}$ outputs 1 when interacting with the protocol Π and a real-world adversary $\mathcal{A}$.

In the ideal world, the protocol is replaced by an incorruptible trusted functionality $\mathcal{F}$. The environment still interacts with the adversary, but now the adversary's role is taken by a *simulator* $\mathcal{S}$, which can only influence the interaction with $\mathcal{F}$ in limited ways (e.g., corrupting parties at the interface defined by $\mathcal{F}$). Let $\mathbf{Ideal}^{\mathcal{F}}_{\mathcal{E},\mathcal{S}}(\kappa)$ denote the probability that $\mathcal{E}$ outputs 1 when interacting with the ideal functionality $\mathcal{F}$ and an ideal-world adversary or simulator $\mathcal{S}$. The distinguishing advantage of $(\mathcal{E}, \mathcal{A})$ against a simulator $\mathcal{S}$ is defined as

$$\left| \mathbf{Ideal}^{\mathcal{F}}_{\mathcal{E},\mathcal{S}}(\kappa) - \mathbf{Real}^{\Pi}_{\mathcal{E},\mathcal{A}}(\kappa) \right| .$$

In the *hybrid world*, all parties additionally have access to auxiliary ideal functionalities $\boldsymbol{\mathcal{G}} = (\mathcal{G}_1, \ldots, \mathcal{G}_m)$ as subroutines. The environment interacts

with a hybrid-world adversary $\mathcal{A}$ that interacts with the protocol $\mathcal{A}$ and the ideal subfunctionalities $\boldsymbol{\mathcal{G}}$. We denote the probability that the environment outputs 1 in this setting as $\mathbf{Hybrid}^{\Pi/\boldsymbol{\mathcal{G}}}_{\mathcal{E},\mathcal{A}}(\kappa)$. The advantage of $(\mathcal{E},\mathcal{A})$ against a simulator $\mathcal{S}$ is defined as

$$|\,\mathbf{Ideal}^{\mathcal{F}}_{\mathcal{E},\mathcal{S}}(\kappa) - \mathbf{Hybrid}^{\Pi/\boldsymbol{\mathcal{G}}}_{\mathcal{E},\mathcal{A}}(\kappa)\,|\ .$$

Completeness. As our protocols are in the asynchronous network setting, an interactive protocol cannot securely realize an ideal functionality that guarantees liveness, i.e., that guarantees that all honest parties obtain their outputs. Without a bound on when messages get delivered, or even a guarantee that they get delivered at all, progress is entirely in the hands of the adversary. At the same time, an ideal functionality that does not guarantee liveness is trivially realized by a "mute" protocol that simply doesn't do anything.

Rather than introducing the complex machinery to add a concept of time to the UC framework [KMTZ13], we follow [SS24] in seeing *completeness* as a separate property of the protocol that must hold *eventually*, i.e., when all messages between honest parties have been delivered.

For each ideal functionality, we will therefore define completeness through a game where the adversary arbitrarily schedules message to be delivered in an arbitrary order. The adversary wins if it has delivered all messages sent between honest parties and the protocol state violates the stated completeness condition; the protocol is complete if all efficient adversaries have negligible probability of winning this game.

3 System Model

Parties and communication channels. Remember our goal to create a simple and versatile distributed key generation and re-sharing system, that can be used securely in a wide variety of settings.

Each instance of the DKG protocol is either a *generation* instance or a *re-sharing* instance. In a generation instance, a group of n_{D} *dealers* $\boldsymbol{\mathcal{D}}$, up to f_{D} of which may be corrupt, jointly generate a freshly generated secret s. They deal the secret so that each *recipient* $\mathcal{R}$ of a group of n_{R} recipients $\boldsymbol{\mathcal{R}}$, up to f_{R} of which may be corrupt, obtains a share $s_{\mathcal{R}}$ of the secret, so that a threshold t_{R} shares are needed to reconstruct the secret.

Even though we treat the sets of dealers and recipients as separate sets here, they do not have to be disjoint: a party can act as a dealer and as a recipient in a single protocol instance. Our model covers the setting with a single set of parties $\boldsymbol{\mathcal{D}} = \boldsymbol{\mathcal{R}}$ as a special case.

At the start of a re-sharing instance, each dealer $\mathcal{D} \in \boldsymbol{\mathcal{D}}$ already owns a share $s_{\mathcal{D}}$ of a secret s with reconstruction threshold t_{D}. That share may be

the outcome of a previous DKG instance where the current dealers acted as recipients, or may have been obtained out-of-band. At the end of the protocol, each recipient $\mathcal{R} \in \boldsymbol{\mathcal{R}}$ obtains a share $s'_{\mathcal{R}}$ of the same secret s with reconstruction threshold t_{R}. We stress that t_{R} does not have to be equal to t_{D}, i.e., the secret may be re-shared under a different threshold among the recipients than it was originally shared among the dealers.

The dealers $\boldsymbol{\mathcal{D}}$ communicate exclusively through a total-order broadcast (i.e., consensus) channel, e.g., a blockchain. We make no assumptions on the communication channels available to the recipients $\boldsymbol{\mathcal{R}}$; our protocol doesn't require them to send any messages at all. In particular, the recipients don't have to have insight into the dealers' broadcast channel. This enables use cases where the recipients are not yet online, or are air-gapped backup devices.

Each party $\mathcal{P}$ locally generates a verification and signing key pair $(vk_{\mathcal{P}}, sk_{\mathcal{P}})$ for a digital signature scheme and an encryption and decryption key $(ek_{\mathcal{P}}, dk_{\mathcal{P}})$ for a public-key encryption scheme. We assume that all parties are securely provisioned with the public keys $(vk_{\mathcal{P}}, ek_{\mathcal{P}})$ of all parties $\mathcal{P} \in \boldsymbol{\mathcal{D}} \cup \boldsymbol{\mathcal{R}}$.

We consider static corruptions, meaning that the adversary announces at the very beginning of the game which parties it wants to corrupt.

DKG protocol suite. A distributed key generation (DKG) protocol suite consists of two interactive protocols Deal and Reshare and two algorithms VerifyDealing and Receive:

- To start a generation instance of the DKG protocol, the dealers $\boldsymbol{\mathcal{D}}$ run the *dealing protocol* Deal. Each dealer $\mathcal{D} \in \boldsymbol{\mathcal{D}}$ starts the protocol on input the dealers $\boldsymbol{\mathcal{D}}$, the recipients $\boldsymbol{\mathcal{R}}$, the recipients' recovery threshold t_{R}, all dealers' and recipients' public keys $(vk_{\mathcal{P}}, ek_{\mathcal{P}})$ for $\mathcal{P} \in \boldsymbol{\mathcal{D}} \cup \boldsymbol{\mathcal{R}}$, and its own private keys $(sk_{\mathcal{D}}, dk_{\mathcal{D}})$.

  At the end of the protocol, all honest dealers output the same *dealings package* pkg.

- To engage in a re-sharing protocol instance of a secret s that is t_{D}-out-of-n_{D} shared among the dealers $\boldsymbol{\mathcal{D}}$, the dealers run the *re-sharing protocol* Reshare. Each dealer $\mathcal{D} \in \boldsymbol{\mathcal{D}}$ starts the protocol on input the groups of dealers $\boldsymbol{\mathcal{D}}$ and recipients $\boldsymbol{\mathcal{R}}$, the dealers' and recipients' recovery thresholds t_{D} and t_{R}, all dealers' and recipients' public keys $(vk_{\mathcal{P}}, ek_{\mathcal{P}})$ for $\mathcal{P} \in \boldsymbol{\mathcal{D}} \cup \boldsymbol{\mathcal{R}}$, its own private keys $(sk_{\mathcal{D}}, dk_{\mathcal{D}})$, and its own secret share $s_{\mathcal{D}}$.

  Just like in the Deal protocol, the honest dealers finish the protocol by outputting the same dealings package pkg.

- The VerifyDealing algorithm verifies a dealings package pkg based on the public keys $(vk_{\mathcal{P}}, ek_{\mathcal{P}})$ of all involved parties $\mathcal{P} \in \boldsymbol{\mathcal{D}} \cup \boldsymbol{\mathcal{R}}$.

  If the dealing package is valid, the VerifyDealing algorithm outputs the overall public key pk and the partial public key $pk_{\mathcal{R}}$ of all recipients $\mathcal{R} \in \boldsymbol{\mathcal{R}}$.

- Upon obtaining and verifying the dealing package pkg, each recipient $\mathcal{R}$ locally runs the $\mathsf{Receive}$ algorithm on input pkg and its decryption key $dk_{\mathcal{R}}$ to recover its secret share $s_{\mathcal{R}}$.

4 Ingredients

4.1 Verifiable Secret Sharing

Verifiable secret sharing (VSS) allows parties in a secret sharing scheme to verify that they received a valid share.

In Feldman's VSS scheme [Fel87], the dealer of a secret $s \in \mathbb{Z}_p$ first proceeds as in Shamir secret sharing, choosing a random polynomial $\boldsymbol{\omega}(X) \in \mathbb{Z}_p[X]$ of degree $t-1$ with $\boldsymbol{\omega}(0) = s$. He then computes the Feldman commitment $\boldsymbol{C}$ to $\boldsymbol{\omega}$ given by

$$\boldsymbol{C} \;\leftarrow\; g^{\boldsymbol{\omega}}$$

and securely broadcasts $\boldsymbol{C}$ to all parties. To each recipient $\mathcal{R} \in \boldsymbol{\mathcal{R}}$, he sends the secret share $s_{\mathcal{R}} = \boldsymbol{\omega}(\mathcal{R})$ over a secure point-to-point channel. Recipient $\mathcal{R}$ can verify the correctness of its share by checking that

$$\boldsymbol{C}^{(\mathcal{R})} \;=\; g^{s_{\mathcal{R}}} \;.$$

4.2 Digital Signatures

A digital signature scheme $\mathcal{DS}$ consists of the following algorithms:

$(pk, sk) \leftarrow_{\$} \mathsf{Keygen}()$**:** The key generation algorithm outputs a public key pk and a corresponding secret key sk.

$\sigma \leftarrow_{\$} \mathsf{Sign}(sk, m)$**:** The signing algorithm uses the secret key sk to sign a message m and produces a signature σ.

$0/1 \leftarrow \mathsf{Verify}(pk, m, \sigma)$**:** The verification algorithm verifies a signature σ with respect to a public key pk and a message m and outputs 1 if it is valid, 0 if it isn't.

Correctness requires that for all messages m and all honestly generated key pairs (pk, sk), $\mathsf{Verify}(pk, m, \mathsf{Sign}(sk, m)) = 1$ with probability 1.

Definition 3 (Existential unforgeability [GMR88].) *Existential unforgeability against chosen-message attack (*$\mathsf{uf\text{-}cma}$*) of a digital signature scheme* $\mathcal{DS} = (\mathsf{Keygen}, \mathsf{Sign}, \mathsf{Verify})$ *is defined through the following game with an adversary* $\mathcal{A}$*:*

- *The experiment generates* $(pk, sk) \leftarrow_{\$} \mathsf{Keygen}()$ *and runs* $\mathcal{A}$ *on input* pk*.*

Functionality $\mathcal{F}_{\text{tob}}$

Initialize $pool \leftarrow \emptyset$, let $ledger$ be an empty array, set $head \leftarrow 0$, and set $delivered[\mathcal{P}] \leftarrow 0$ for all $\mathcal{P} \in \boldsymbol{\mathcal{P}}$.

- Upon input $(sid, \texttt{broadcast}, m)$ from party $\mathcal{P}$, add $(\mathcal{P}, m)$ to $pool$ and send $(sid, \texttt{broadcast}, \mathcal{P}, m)$ to the adversary.
- Upon input $(sid, \texttt{schedule}, \mathcal{P}, m)$ from the adversary, check whether $(\mathcal{P}, m) \in pool$ and that $(\mathcal{P}, m) \notin ledger$. If so, remove $(\mathcal{P}, m)$ from $pool$, increment $head$ by 1, and set $ledger[head] \leftarrow (\mathcal{P}, m)$.
- Upon input $(sid, \texttt{deliver}, \mathcal{P})$ from the adversary, check that $delivered[\mathcal{P}] < head$. If so, increment $delivered[\mathcal{P}]$ by 1 and output $(sid, \texttt{deliver}, ledger[delivered[\mathcal{P}]])$ to $\mathcal{P}$.

Figure 1: Total-order broadcast functionality.

- *The adversary $\mathcal{A}$ can make an arbitrary number of queries to a signing oracle* $\mathsf{Sign}(sk, \cdot)$.
- *The adversary outputs m, σ and wins if* $\mathsf{Verify}(pk, m, \sigma) = 1$ *and $\mathcal{A}$ never submitted m to its* Sign *oracle.*

The advantage $\mathbf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathcal{DS},\mathcal{A}}(\kappa)$ *is given by the probability that $\mathcal{A}$ wins the game. The scheme $\mathcal{DS}$ is said to be* uf-cma *secure if this advantage is negligible for all efficient adversaries $\mathcal{A}$.*

4.3 Total-Order Broadcast

A total-order broadcast protocol implements a broadcast channel that guarantees that all parties receive the same messages in the exact same order. All parties can submit messages to the broadcast channel; the order in which they appear on the broadcast channel is determined by the adversary.

We present an ideal functionality for total-order broadcast in Figure 1. It maintains the messages that appeared on the channel in an array *ledger*. All parties can submit messages for broadcast using the `broadcast` interface, but the scheduling of messages on the channel is completely in the hands of the adversary through the `schedule` interface. The adversary also schedules the (in-order) delivery of messages to individual recipients through the `deliver` interface.

The completeness condition for $\mathcal{F}_{\text{tob}}$ requires that, if an honest party $\mathcal{P}$ broadcast a message m, or an honest party outputs a message $(\mathcal{P}, m)$, then all honest parties *eventually* output $(\mathcal{P}, m)$.

Functionality $\mathcal{F}_{\text{crs}}^{\boldsymbol{Y}}$

- On initialization, choose $crs \leftarrow_{\$} \boldsymbol{Y}$.
- Upon input $(sid, \texttt{query})$ from a party $\mathcal{P}$, output (sid, crs) to $\mathcal{P}$.

Figure 2: Functionality for a common reference string with range $\boldsymbol{Y}$.

Functionality $\mathcal{F}_{\text{ro}}^{\boldsymbol{X} \to \boldsymbol{Y}}$

- Upon input (sid, m) with $m \in \boldsymbol{X}$ from a party $\mathcal{P}$, check whether $T[m]$ is already defined; if not, set $T[m] \leftarrow_{\$} \boldsymbol{Y}$. Output $(sid, T[m])$ to $\mathcal{P}$.

Figure 3: Functionality for a random oracle with domain $\boldsymbol{X}$ and range $\boldsymbol{Y}$. If the domain is $\boldsymbol{X} = \{0,1\}^*$, we also refer to the functionality as $\mathcal{F}_{\text{ro}}^{\boldsymbol{Y}}$.

In practice, the $\mathcal{F}_{\text{tob}}$ can be implemented using a consensus protocol that is run by the parties themselves (in which case less than a third of the parties can be corrupt), or by an external set of parties (e.g., as a blockchain) that the current parties interact with. Section 8 explains how to realize $\mathcal{F}_{\text{tob}}$ with an oracle network that runs OCR 3.1 [BCK+24].

4.4 Other Subfunctionalities

Our protocol also relies on the standard ideal functionalities for a common reference string $\mathcal{F}_{\text{crs}}$, random oracles $\mathcal{F}_{\text{ro}}$, and a public-key infrastructure $\mathcal{F}_{\text{pki}}$ depicted in Figures 2, 3, and 4, respectively. The latter two keep an initially empty table T to ensure consistency in their responses.

The $\mathcal{F}_{\text{ro}}$ and $\mathcal{F}_{\text{crs}}$ functionalities only have interfaces that produce immediate outputs, so they do not have a completeness condition. For $\mathcal{F}_{\text{pki}}$, the completeness condition requires that if any party $\mathcal{P}$ registers its public key pk and an honest party $\mathcal{P}'$ looks up the key for $\mathcal{P}$, then $\mathcal{P}'$ eventually outputs $(\mathcal{P}, pk)$.

Functionality $\mathcal{F}_{\text{pki}}$

- Upon input $(sid, \texttt{register}, pk)$ from party $\mathcal{P}$, check whether $T[\mathcal{P}]$ is already defined; if not, set $T[\mathcal{P}] \leftarrow pk$.
- Upon input $(sid, \texttt{lookup}, \mathcal{P})$ from party $\mathcal{P}'$, send a public delayed output $(sid, \mathcal{P}, T[\mathcal{P}])$ to $\mathcal{P}'$.

Figure 4: Functionality for a public-key infrastructure.

4.5 Multi-Recipient Encryption

A multi-recipient encryption (MRE) scheme enables a sender to encrypt n different messages to n different recipients in a single ciphertext. The hope is that the ciphertext can be shorter or cheaper to encrypt or decrypt than the concatenation of n separately encrypted ciphertexts.

We recall the MRE scheme suggested by Shoup [Sho25]; we simply refer to it as the $\mathcal{MRE}$ scheme here. Just like the DKG protocol, it uses a cyclic prime-order group, but it doesn't have to be the same group as the DKG protocol uses. To distinguish both groups, we describe the MRE scheme in a group $\bar{\mathbb{G}}$ of prime order $\bar{p}$ and with generator $\bar{g}$. Of course, one can always use $(\bar{\mathbb{G}}, \bar{p}, \bar{g}) = (\mathbb{G}, p, g)$.

The scheme also uses a hash function $\mathsf{H}_{\text{enc}} : \{0,1\}^* \to \{0,1\}^\ell$, where ℓ is the length of the encrypted messages.

Key generation. $(ek, dk) \leftarrow_\$ \mathsf{Keygen}()$: Each recipient generates a private decryption key $dk \leftarrow_\$ \mathbb{Z}_{\bar{p}}$ and computes the corresponding public encryption key $ek \leftarrow \bar{g}^{dk}$.

Encryption. $\boldsymbol{E} \leftarrow_\$ \mathsf{Encrypt}(\boldsymbol{ek}, \boldsymbol{m}, ad; r)$: To encrypt a vector of messages $\boldsymbol{m} = (m_1, \ldots, m_n)$ to public keys $\boldsymbol{ek} = (ek_1, \ldots, ek_n)$ with associated data ad and randomness $r \leftarrow \mathbb{Z}_{\bar{p}}$, the sender computes

$$\begin{aligned} E_0 &\leftarrow \bar{g}^r \\ E_i &\leftarrow m_i \oplus \mathsf{H}_{\text{enc}}(ad, i, ek_i, E_0, ek_i^r) \quad \text{for } i = 1, \ldots, n \\ \boldsymbol{E} &\leftarrow (E_0, E_1, \ldots, E_n) \ . \end{aligned}$$

Decryption. $m_i \leftarrow \mathsf{Decrypt}(dk_i, i, \boldsymbol{E}, ad)$: To decrypt its message component from a ciphertext $\boldsymbol{E}$, recipient i computes

$$m_i \leftarrow E_i \oplus \mathsf{H}_{\text{enc}}(ad, i, ek_i, E_0, E_0^{dk_i}) \ .$$

Completeness. One can easily check that if an honest recipient generates a key pair (ek^*, dk^*) and an honest sender encrypts a message $\boldsymbol{m} = (m_1, \ldots, m_n)$ with associated data ad to $\boldsymbol{ek} = (ek_1, \ldots, ek_n)$ with $ek_i = ek^*$ to produce a ciphertext $\boldsymbol{E}$, then $\mathsf{Decrypt}(dk_i, i, \boldsymbol{E}, ad)$ returns m_i.

4.6 Cut-and-Choose Verifiably Encrypted Secret Sharing

The following cut-and-choose verifiably encrypted secret sharing (VESS) scheme by Shoup [Sho25], that we simply refer to as the $\mathcal{VESS}$ scheme here, works for any cyclic group g of prime order p with generator g and any multi-recipient encryption scheme $\mathcal{MRE}$ with message length $\ell = \lceil \log_2(p-1) \rceil$.

It is parameterized by the (symmetric-key equivalent) security parameter $\kappa \in \mathbb{N}$, the number of recipients n, the recovery threshold t with $0 < t \leq n$, a

repetition parameter $N \in \mathbb{N}$ and a subset size parameter M with $0 < M \leq N$ so that $1/\binom{N}{M}$ is negligible. It uses three hash functions

$$\begin{aligned} \mathsf{H}^t_{\text{exp}} &: \{0,1\}^* \to \mathbb{Z}_p \times \mathbb{Z}_p^t \\ \mathsf{H}_{\text{comp}} &: \{0,1\}^* \to \{0,1\}^{2\kappa} \\ \mathsf{H}_{\text{ch}} &: \{0,1\}^* \to \binom{[N]}{M}, \end{aligned}$$

where $\binom{[N]}{M}$ denotes the set of all subsets of $[N] = \{1, \ldots, N\}$ of size M. A practical implementation for $\mathsf{H}^t_{\text{exp}}$ could use an extendable-output hash function XOF such as SHAKE-256 and let $\mathsf{H}^t_{\text{exp}}(\cdot)$ be defined by drawing as many bits from $\mathsf{XOF}(t, \cdot)$ as needed.

The scheme also uses a common reference string *crs* sampled uniformly from $\mathbb{G}$ that is an implicit input to all algorithms. In practice, *crs* can be set to the output of a hash function with range $\mathbb{G}$ modeled as a random oracle. We will use the *crs* as a public key "in the sky" for the MRE scheme, meaning that no party knows the corresponding decryption key in the real protocol, but that the simulator can program in the security proof to extract any message that was "encrypted to the sky".

All hash functions above are modeled as independent random oracles; domain separation must be used if they are implemented through a common, real-world hash function.

Key generation. $(ek, dk) \leftarrow_{\$} \mathsf{Keygen}()$: Each recipient $\mathcal{R}_i \in \boldsymbol{\mathcal{R}}$ generates a key pair for the MRE scheme through $\mathcal{MRE}.\mathsf{Keygen}()$ to generate an encryption and decryption key pair (ek_i, dk_i).

Dealing creation. $D \leftarrow_{\$} \mathsf{Deal}(s, \boldsymbol{\mathcal{R}}, t, \boldsymbol{ek}_{\boldsymbol{\mathcal{R}}}, ad)$: To create a publicly verifiable t-out-of-n sharing of a secret $s \in \mathbb{Z}_p$ to a list of n recipients $\boldsymbol{\mathcal{R}}$ with public encryption keys $\boldsymbol{ek}_{\boldsymbol{\mathcal{R}}}$ and associated data ad, the dealer

- chooses a random polynomial $\boldsymbol{\omega}(X) \in \mathbb{Z}_p[X]$ of degree $t-1$ with $\boldsymbol{\omega}(0) = \omega_0 = s$,
- computes $\boldsymbol{C} \leftarrow g^{\boldsymbol{\omega}}$,
- for $k = 1, \ldots, N$, chooses an expansion seed $seed_k \leftarrow_{\$} \{0,1\}^{\kappa}$ and computes

$$\begin{aligned} (r_k, \boldsymbol{\omega}'_k) &\leftarrow \mathsf{H}^t_{\text{exp}}(k, seed_k, ad) \\ \boldsymbol{C}'_k &\leftarrow g^{\boldsymbol{\omega}'_k} \\ \boldsymbol{ek} &\leftarrow \boldsymbol{ek}_{\boldsymbol{\mathcal{R}}} \,\|\, crs \\ \boldsymbol{m} &\leftarrow \boldsymbol{\omega}'_k(\boldsymbol{\mathcal{R}}) \,\|\, \langle \boldsymbol{\omega}'_k \rangle \\ \boldsymbol{E}_k &\leftarrow \mathcal{MRE}.\mathsf{Encrypt}(\boldsymbol{ek}, \boldsymbol{m}, ad; r_k) \end{aligned}$$

where we use $\|$ to denote appending an element to the end of a vector, and we use $\langle \boldsymbol{\omega}'_k \rangle$ to denote an encoding of $\boldsymbol{\omega}'_k$ interpreted as a single component, and where we use $\mathcal{MRE}.\mathsf{Encrypt}(\ldots; r_k)$ to denote MRE encryption using randomness r_k.

- computes
$$h \leftarrow \mathsf{H}_{\mathrm{comp}}(\boldsymbol{C}'_1, \boldsymbol{E}_1, \ldots, \boldsymbol{C}'_N, \boldsymbol{E}_N, ad)$$
and computes the challenge set S as
$$S \leftarrow \mathsf{H}_{\mathrm{ch}}(ad, \boldsymbol{C}, h) \ ,$$
- for $k = 1, \ldots, N$,
  - if $k \in S$, computes $\boldsymbol{\omega}''_k \leftarrow \boldsymbol{\omega} + \boldsymbol{\omega}'_k$ and sets the response
  $$\rho_k \leftarrow (\boldsymbol{\omega}''_k, \boldsymbol{E}_k) \ ,$$
  - if $k \notin S$, sets the response $\rho_k \leftarrow \mathit{seed}_k$,
- lets $\boldsymbol{\rho} \leftarrow (\rho_1, \ldots, \rho_N)$ and outputs the dealing
$$D \leftarrow (\boldsymbol{C}, h, \boldsymbol{\rho}) \ .$$

Dealing verification. $b \leftarrow \mathsf{VerifyDealing}(D, \boldsymbol{\mathcal{R}}, t_{\mathrm{R}}, \boldsymbol{ek}_{\boldsymbol{\mathcal{R}}}, ad)$: To verify a dealing $D = (\boldsymbol{C}, h, \boldsymbol{\rho})$, the verifier

- checks that $\boldsymbol{C} \in \mathbb{G}^{t_{\mathrm{R}}}$,
- recomputes the challenge set $S \leftarrow \mathsf{H}_{\mathrm{ch}}(ad, \boldsymbol{C}, h)$,
- for $k = 1, \ldots, N$, recomputes $(\boldsymbol{C}'_k, \boldsymbol{E}_k)$ by
  - if $k \in S$, parsing $\rho_k = (\boldsymbol{\omega}''_k, \boldsymbol{E}_k)$ and recomputing
  $$\boldsymbol{C}'_k \leftarrow g^{\boldsymbol{\omega}''_k} / \boldsymbol{C} \ ,$$
  - if $k \notin S$, parsing $\rho_k = \mathit{seed}_k$ and recomputing
  $$\begin{aligned} (r_k, \boldsymbol{\omega}'_k) &\leftarrow \mathsf{H}^t_{\mathrm{exp}}(k, \mathit{seed}_k, ad) \\ \boldsymbol{C}'_k &\leftarrow g^{\boldsymbol{\omega}'_k} \\ \boldsymbol{ek} &\leftarrow \boldsymbol{ek}_{\boldsymbol{\mathcal{R}}} \,\|\, \mathit{crs} \\ \boldsymbol{m} &\leftarrow \boldsymbol{\omega}'_k(\boldsymbol{\mathcal{R}}) \,\|\, \langle \boldsymbol{\omega}'_k \rangle \\ \boldsymbol{E}_k &\leftarrow \mathcal{MRE}.\mathsf{Encrypt}(\boldsymbol{ek}, \boldsymbol{m}, ad; r_k) \end{aligned}$$
- checks that
$$h = \mathsf{H}_{\mathrm{comp}}(\boldsymbol{C}'_1, \boldsymbol{E}_1, \ldots, \boldsymbol{C}'_N, \boldsymbol{E}_N, ad)$$
and returns 1 if so, otherwise returns 0.

Decryption. $s_{\mathcal{R}} \leftarrow \mathsf{Decrypt}(dk_{\mathcal{R}}, D, \mathcal{R}, ad)$: To decrypt its share $s_{\mathcal{R}}$ from a dealing $D = (\boldsymbol{C}, h, \boldsymbol{\rho})$, recipient $\mathcal{R} \in \boldsymbol{\mathcal{R}}$ recomputes the challenge set $S \leftarrow \mathsf{H}_{\mathrm{ch}}(ad, \boldsymbol{C}, h)$, and for $k \in S$

- parses $\rho_k = (\boldsymbol{\omega}''_k, \boldsymbol{E}_k)$,
- decrypts
$$s'_{\mathcal{R}} \leftarrow \mathcal{MRE}.\mathsf{Decrypt}(dk_{\mathcal{R}}, i, \boldsymbol{E}_k, ad)$$
where i is the index of $\mathcal{R}$ in $\boldsymbol{\mathcal{R}}$,
- computes
$$s_{\mathcal{R}} \leftarrow \boldsymbol{\omega}''_k(\mathcal{R}) - s'_{\mathcal{R}} \ ,$$
- verifies the validity of the share by checking that
$$\mathcal{VESS}.\mathsf{VerifyShare}(s_{\mathcal{R}}, D, \mathcal{R}) = 1 \ .$$
If so, it returns $s_{\mathcal{R}}$ and halts, otherwise it proceeds to the next $k \in S$. If this fails for all $k \in S$, decryption fails; this happens with negligible probability because $1/\binom{N}{M}$ is negligible.

Share verification. $0/1 \leftarrow \mathsf{VerifyShare}(s_{\mathcal{R}}, D, \mathcal{R})$: To check whether a share $s_{\mathcal{R}}$ is recipient $\mathcal{R}$'s correct share for the dealing $D = (\boldsymbol{C}, h, \boldsymbol{\rho})$, one checks whether
$$g^{s_{\mathcal{R}}} = \boldsymbol{C}^{(\mathcal{R})}$$
and outputs 1 if so, or outputs 0 if not.

Soundness. An adversary $\mathcal{A}$ breaks the soundness of the VESS scheme if it can create a valid dealing $D = (\boldsymbol{C}, h, \boldsymbol{\rho})$ so that decryption fails for some honest recipient $\mathcal{R}$ or so that the encryption to the CRS does not yield a polynomial $\boldsymbol{\omega}$ such that $\boldsymbol{C} = g^{\boldsymbol{\omega}}$. (Note that if a recipient successfully decrypts its share, it always obtains a share $s_{\mathcal{R}}$ such that $g^{s_{\mathcal{R}}} = \boldsymbol{C}^{(\mathcal{R})}$.)

To do so, the adversary must have used MRE ciphertexts with an invalid message encrypted to at least one honest recipient $\mathcal{R}$ or the CRS for all $k \in S = \mathsf{H}_{\mathrm{ch}}(ad, \boldsymbol{C}, h)$, but must have used a valid encryption of $s'_{\mathcal{R}}$ such that $g^{s'_{\mathcal{R}}} = g^{\boldsymbol{\omega}''_k(\mathcal{R})}/\boldsymbol{C}^{(\mathcal{R})}$ or of an incorrect polynomial $\langle \boldsymbol{\omega}'_k \rangle$ to the CRS for all $k \notin S$.

Let's assume that no collisions (i.e., different inputs mapping to the same output) occur in $\mathsf{H}_{\mathrm{comp}}$; the probability of a collision appearing in $\mathsf{H}_{\mathrm{comp}}$ if $\mathcal{A}$ makes at most q_{H} random-oracle queries is at most $q_{\mathrm{H}}^2/2^{2\kappa}$. Every query $S = \mathsf{H}_{\mathrm{ch}}(ad, \boldsymbol{C}, h)$ therefore corresponds to at most one query $h = \mathsf{H}_{\mathrm{comp}}(\boldsymbol{C}'_1, \boldsymbol{E}_1, \ldots, \boldsymbol{C}'_N, \boldsymbol{E}_N, ad)$. The ciphertexts $\boldsymbol{E}_1, \ldots, \boldsymbol{E}_N$ can be divided in "good" ciphertexts $\boldsymbol{E}_k$ that for all honest recipients $\mathcal{R} \in \boldsymbol{\mathcal{HR}}$ decrypt to a correct share $s'_{\mathcal{R}}$ such that $\boldsymbol{C}'^{(\mathcal{R})}_k = g^{s'_{\mathcal{R}}}$ and that the CRS decrypts to a correct polynomial

$\boldsymbol{\omega}'_k$ such that $\boldsymbol{C}'_k = g^{\boldsymbol{\omega}'_k}$, and "bad" ciphertexts $\boldsymbol{E}_k$ that do not decrypt correctly for at least one honest recipient or the CRS. For the zero-knowledge proof to be valid, $\boldsymbol{E}_k$ has to be a good ciphertext for all $k \notin S$. For the ciphertext to not be decryptable by at least one honest recipient, $\boldsymbol{E}_k$ must be a bad ciphertext for all $k \in S$.

For each value h, there is therefore only a single outcome for S that enables the adversary to create a valid but not decryptable VESS dealing. The probability that $\mathsf{H}_{\mathrm{ch}}(ad, \boldsymbol{C}, h)$ hits exactly that value S is $1/\binom{N}{M}$; the probability that the adversary succeeds at least once in at most q_{H} random-oracle queries is at most $q_{\mathrm{H}}/\binom{N}{M}$. Overall, we therefore have that the soundness error is at most

$$\frac{q_{\mathrm{H}}^2}{2^{2\kappa}} + \frac{q_{\mathrm{H}}}{\binom{N}{M}} \ .$$

Completeness. One can easily check that if

- a set of honest recipients $\mathcal{R} \in \boldsymbol{\mathcal{HR}}$ generate key pairs and
- an honest dealer creates a dealing $D = (\boldsymbol{C}, h, \boldsymbol{\rho})$ of a secret s to recipients $\boldsymbol{\mathcal{R}} \supseteq \boldsymbol{\mathcal{HR}}$,

then

- the dealing D is valid according to $\mathsf{VerifyDealing}$, and
- all honest recipients $\mathcal{R} \in \boldsymbol{\mathcal{HR}}$ decrypt it to a valid share $s_{\mathcal{R}}$ such that $\boldsymbol{C}^{(\mathcal{R})} = g^{s_{\mathcal{R}}}$.

5 The DKG Protocol Suite

We present a full DKG protocol suite based on the $\mathcal{VESS}$ scheme from the previous section and the "secret bulletin board" DKG protocol from [BS23], Section 22.4.2; we'll simply refer to it as the $\mathcal{DKG}$ protocol suite here.

We assume that all participants in a dealing or resharing protocol agree on a unique identifier *iid* for the protocol instance. The protocol uses a variable-output-length hash function $\mathsf{H}^{\ell}_{\mathrm{deal}} : \{0,1\}^* \to \{0,1\}^{\ell}$, modeled as a random oracle.

Note that all random oracles are considered local to the protocol instance, including those in the VESS and MRE scheme. In a real-world implementation using a hash function, this is most easily realized by prepending the instance identifier *iid* to the hash argument for $\mathsf{H}_{\mathrm{comp}}$ and H_{ch}, and by prepending *iid* and the iteration counter k to the hash arguments in $\mathsf{H}_{\mathrm{exp}}$ and $\mathsf{H}_{\mathrm{enc}}$.

Dealing protocol.

$$pkg \leftarrow_{\$} \mathsf{Deal}(iid, \boldsymbol{\mathcal{D}}, f_{\mathrm{D}}, \boldsymbol{\mathcal{R}}, f_{\mathrm{R}}, t_{\mathrm{R}}, \boldsymbol{vk}_{\boldsymbol{\mathcal{D}}\cup\boldsymbol{\mathcal{R}}}, \boldsymbol{ek}_{\boldsymbol{\mathcal{D}}\cup\boldsymbol{\mathcal{R}}}, \mathcal{D}, sk_{\mathcal{D}}, dk_{\mathcal{D}})$$

Let $n_{\mathrm{D}} = |\boldsymbol{\mathcal{D}}|$ and $n_{\mathrm{R}} = |\boldsymbol{\mathcal{R}}|$. Each dealer $\mathcal{D} \in \boldsymbol{\mathcal{D}}$

- chooses a random secret $s_{\mathcal{D}} \leftarrow_{\$} \mathbb{Z}_p$ and creates an "inner" dealing that t_{R}-out-of-n_{R} secret-shares $s_{\mathcal{D}}$ to the recipients $\boldsymbol{\mathcal{R}}$

  $$ID_{\mathcal{D}} \leftarrow_{\$} \mathcal{VESS}.\mathsf{Deal}(s_{\mathcal{D}}, \boldsymbol{\mathcal{R}}, t_{\mathrm{R}}, \boldsymbol{ek}_{\boldsymbol{\mathcal{R}}}, ad)$$

  as described in Section 4.6 with $ad = (iid, \texttt{inner}, \mathcal{D}, attempt)$, where `inner` is some fixed string and *attempt* is a counter of re-sharing attempts that is set to zero in a fresh generation,

- chooses a second random secret $z_{\mathcal{D}} \leftarrow_{\$} \mathbb{Z}_p$ and creates an "outer" dealing that $(f_{\mathrm{D}}+1)$-out-of-n_{D} secret-shares $z_{\mathcal{D}}$ to the dealers $\boldsymbol{\mathcal{D}}$

  $$OD_{\mathcal{D}} \leftarrow_{\$} \mathcal{VESS}.\mathsf{Deal}(z_{\mathcal{D}}, \boldsymbol{\mathcal{D}}, f_{\mathrm{D}}+1, \boldsymbol{ek}_{\boldsymbol{\mathcal{D}}}, ad)$$

  with $ad = (iid, \texttt{outer}, \mathcal{D}, attempt)$, where `outer` is a fixed string different from `inner` and $attempt = 0$,

- encrypts the inner dealing as

  $$EID_{\mathcal{D}} \leftarrow \mathsf{H}^{\ell}_{\mathrm{deal}}(iid, \mathcal{D}, z_{\mathcal{D}}) \oplus ID_{\mathcal{D}} \tag{1}$$

  with $\ell = |ID_{\mathcal{D}}|$,

- signs and broadcasts $(OD_{\mathcal{D}}, EID_{\mathcal{D}})$ to all other dealers.

Each dealer $\mathcal{D}$ compiles the list $\boldsymbol{L}$ of the first $f_{\mathrm{D}}+1$ dealings $(OD_{\mathcal{D}'}, EID_{\mathcal{D}'})$ that appear on the broadcast channel that

- are validly signed by different dealers $\mathcal{D}' \in \boldsymbol{\mathcal{D}}$, and
- have valid outer dealings, i.e.,

  $$\mathcal{VESS}.\mathsf{VerifyDealing}(OD_{\mathcal{D}'}, \boldsymbol{\mathcal{D}}, f_{\mathrm{D}}+1, \boldsymbol{ek}_{\boldsymbol{\mathcal{D}}}, ad) = 1$$

  for $ad = (iid, \texttt{outer}, \mathcal{D}', attempt)$.

We will ambiguously use $\boldsymbol{L}$ to denote the list of dealings as well as the list of dealers $\mathcal{D}'$ whose dealings appear in $\boldsymbol{L}$; it should be clear from the context which is meant. Note that, because of the total-order broadcast and the public verifiability of the outer dealings, all dealers compile the same list $\boldsymbol{L}$.

Once the list $\boldsymbol{L}$ has been compiled, each dealer $\mathcal{D}$

- decrypts its decryption key shares $z_{\mathcal{D}',\mathcal{D}}$ for all $\mathcal{D}' \in \boldsymbol{L}$ as

$$z_{\mathcal{D}',\mathcal{D}} \leftarrow \mathcal{VESS}.\mathsf{Decrypt}(dk_{\mathcal{D}}, OD_{\mathcal{D}'}, \mathcal{D}, ad)$$

  for $ad = (iid, \texttt{outer}, \mathcal{D}', attempt)$, and

- signs and broadcasts $(z_{\mathcal{D}',\mathcal{D}})_{\mathcal{D}' \in \boldsymbol{L}}$ to all dealers.

When some dealer $\mathcal{D}$ receives a tuple of decryption key shares $(z_{\mathcal{D}',\mathcal{D}^*})_{\mathcal{D}' \in \boldsymbol{L}}$ from dealer $\mathcal{D}^*$ on the broadcast channel, it verifies the signature by $\mathcal{D}^*$ and verifies the shares by checking that

$$\mathcal{VESS}.\mathsf{VerifyShare}(z_{\mathcal{D}',\mathcal{D}^*}, OD_{\mathcal{D}'}, \mathcal{D}^*) = 1$$

for all $\mathcal{D}' \in \boldsymbol{L}$.

When it has received valid tuples of decryption key shares from any subset of $f_{\mathrm{D}} + 1$ different dealers $\boldsymbol{\mathcal{D}^*} \subseteq \boldsymbol{\mathcal{D}}$, dealer $\mathcal{D}$, for all $\mathcal{D}' \in \boldsymbol{L}$,

- reconstructs the decryption keys $z_{\mathcal{D}'}$ as

$$z_{\mathcal{D}'} \leftarrow \sum_{\mathcal{D}^* \in \boldsymbol{\mathcal{D}^*}} \boldsymbol{\Lambda}_{\boldsymbol{\mathcal{D}^*},\mathcal{D}^*}(0) \cdot z_{\mathcal{D}',\mathcal{D}^*} \tag{2}$$

- recovers the inner dealing as

$$ID_{\mathcal{D}'} \leftarrow \mathsf{H}^{\ell}_{\mathrm{deal}}(iid, \mathcal{D}', z_{\mathcal{D}'}) \oplus EID_{\mathcal{D}'} \tag{3}$$

  with $\ell = |EID_{\mathcal{D}'}|$, and

- verifies the inner dealing by checking that

$$\mathcal{VESS}.\mathsf{VerifyDealing}(ID_{\mathcal{D}'}, \boldsymbol{\mathcal{R}}, t_{\mathrm{R}}, \boldsymbol{ek}_{\boldsymbol{\mathcal{R}}}, ad) = 1$$

  with $ad = (iid, \texttt{inner}, \mathcal{D}', attempt)$, and if so, adds $ID_{\mathcal{D}'}$ to an initially empty set $\boldsymbol{L'}$.

Note that the set $\boldsymbol{L'}$ must have at least one dealing in it, as at least one dealing in $\boldsymbol{L}$ was created by an honest dealer. Also note that all dealers will construct the same set $\boldsymbol{L'}$ by the deterministic verification.

The dealer $\mathcal{D}$ then composes the list of inner dealings

$$\boldsymbol{ID} \leftarrow (ID_{\mathcal{D}'})_{\mathcal{D}' \in \boldsymbol{L'}}$$

and broadcasts a signature on $\boldsymbol{ID}$. When it has received valid signatures on $\boldsymbol{ID}$ from $f_{\mathrm{D}} + 1$ different dealers in $\boldsymbol{\mathcal{D}}$, it bundles the signatures into a vector $\boldsymbol{\sigma}$ and creates the dealing package $pkg \leftarrow (\boldsymbol{ID}, \boldsymbol{\sigma}, attempt)$, where the attempt counter *attempt* is included so that recipients can reconstruct the associated data ad used in the inner dealings.

Verification and public-key extraction.

$$(y, \boldsymbol{y}_{\boldsymbol{\mathcal{R}}}) \leftarrow_{\$} \mathsf{VerifyDealing}(pkg, \boldsymbol{\mathcal{D}}, f_{\mathrm{D}}, \boldsymbol{\mathcal{R}}, \boldsymbol{vk}_{\boldsymbol{\mathcal{D}}})$$

To verify a dealing package $pkg = (\boldsymbol{ID}, \boldsymbol{\sigma}, attempt)$, one verifies that $\boldsymbol{\sigma}$ contains valid signatures on $\boldsymbol{ID}$ by $f_{\mathrm{D}} + 1$ dealers in $\boldsymbol{\mathcal{D}}$.

To extract the generated master public key y and the recipients' partial public keys $y_{\mathcal{R}}$, $\mathcal{R} \in \boldsymbol{\mathcal{R}}$, one parses $\boldsymbol{ID} = (\boldsymbol{C}_{\mathcal{D}}, h_{\mathcal{D}}, \boldsymbol{\rho}_{\mathcal{D}})_{\mathcal{D} \in \boldsymbol{L'}}$.

For the dealing package resulting from a fresh dealing, one computes

$$\boldsymbol{C} \leftarrow \prod_{\mathcal{D} \in \boldsymbol{L'}} \boldsymbol{C}_{\mathcal{D}} \ ,$$

while for the dealing package resulting from a re-sharing, one computes

$$\boldsymbol{C} \leftarrow \prod_{\mathcal{D} \in \boldsymbol{L'}} \boldsymbol{C}_{\mathcal{D}}^{\boldsymbol{\Lambda}_{\boldsymbol{L'}, \mathcal{D}}(0)} \ .$$

The master and partial public keys are given by

$$y \leftarrow \boldsymbol{C}^{(0)}$$
$$\boldsymbol{y}_{\boldsymbol{\mathcal{R}}} \leftarrow \boldsymbol{C}^{(\boldsymbol{\mathcal{R}})} \ .$$

Resharing protocol.

$pkg \leftarrow_{\$} \mathsf{Reshare}(iid, \boldsymbol{\mathcal{D}}, f_{\mathrm{D}}, t_{\mathrm{D}}, \boldsymbol{\mathcal{R}}, f_{\mathrm{R}}, t_{\mathrm{R}}, \boldsymbol{vk}_{\boldsymbol{\mathcal{D}} \cup \boldsymbol{\mathcal{R}}}, \boldsymbol{ek}_{\boldsymbol{\mathcal{D}} \cup \boldsymbol{\mathcal{R}}}, \mathcal{D}, sk_{\mathcal{D}}, dk_{\mathcal{D}}, y, \boldsymbol{y}_{\boldsymbol{\mathcal{D}}}, s_{\mathcal{D}})$

The Reshare protocol takes a secret s that was previously t_{D}-out-of-n_{D} secret-shared to the dealers $\boldsymbol{\mathcal{D}}$, meaning, it was freshly generated or re-shared in a previous protocol instance where the dealers $\boldsymbol{\mathcal{D}}$ acted as the recipients. It then t_{R}-out-of-n_{R} re-shares this secret to the new recipients $\boldsymbol{\mathcal{R}}$.

In addition to the inputs of the Deal protocol, the Reshare protocol also takes:

- the reconstruction threshold t_{D} under which the secret was previously shared among $\boldsymbol{\mathcal{D}}$,
- the master public key $y = g^s$ and the partial public keys $\boldsymbol{y}_{\boldsymbol{\mathcal{D}}}$ of the previous (re-)sharing of the secret, and
- this dealer $\mathcal{D}$'s secret share $s_{\mathcal{D}}$.

The Reshare protocol is identical to the Deal protocol above, except that each dealer $\mathcal{D}$

- keeps an attempt counter *attempt* and list of banned dealers $\boldsymbol{\mathcal{B}}$ that are initially set to zero and empty, respectively;

- secret-shares its *existing* share $s_{\mathcal{D}}$, rather than a random element of $\mathbb{Z}_p$;
- includes *attempt* in the associated data for the inner and outer dealings, i.e., $ad = (iid, \texttt{inner}, \mathcal{D}, attempt)$ and $ad = (iid, \texttt{outer}, \mathcal{D}, attempt)$;
- keeps an initially empty set of "banned" dealers $\boldsymbol{\mathcal{B}}$ and compiles the list $\boldsymbol{L}$ by collecting dealings from t_{D} different dealers in $\boldsymbol{\mathcal{D}} \setminus \boldsymbol{\mathcal{B}}$, rather than from $f_{\mathrm{D}} + 1$ different dealers in $\boldsymbol{\mathcal{D}}$,
- when verifying the inner dealing $ID_{\mathcal{D}'} = (\boldsymbol{C}_{\mathcal{D}'}, h_{\mathcal{D}'}, \boldsymbol{\rho}_{\mathcal{D}'})$ received from dealer $\mathcal{D}'$, adds a check that

$$\boldsymbol{C}_{\mathcal{D}'}^{(0)} = y_{\mathcal{D}'} \ ,$$

- if an inner dealing $ID_{\mathcal{D}'}$ by dealer $\mathcal{D}'$ is found to be invalid, adds $\mathcal{D}'$ to the banned dealers $\boldsymbol{\mathcal{B}}$, discards all dealings, increments *attempt*, and restarts the dealing protocol with the updated set $\boldsymbol{\mathcal{B}}$.

Receiving.

$$s_{\mathcal{R}} \leftarrow \mathsf{Receive}(iid, \boldsymbol{\mathcal{D}}, f_{\mathrm{D}}, \boldsymbol{\mathcal{R}}, f_{\mathrm{R}}, t_{\mathrm{R}}, \boldsymbol{vk}_{\boldsymbol{\mathcal{D}}}, \mathcal{R}, dk_{\mathcal{R}}, pkg)$$

Recipient $\mathcal{R} \in \boldsymbol{\mathcal{R}}$ parses the dealing package $pkg = ((ID_{\mathcal{D}})_{\mathcal{D} \in \boldsymbol{L'}}, \boldsymbol{\sigma}, attempt)$ and

- verifies that the signatures $\boldsymbol{\sigma}$ are valid signatures by $f_{\mathrm{D}}+1$ different dealers,
- decrypts each dealing by $\mathcal{D} \in \boldsymbol{L'}$ separately to recover

$$s_{\mathcal{D},\mathcal{R}} \leftarrow \mathcal{VESS}.\mathsf{Decrypt}(dk_{\mathcal{R}}, ID_{\mathcal{D}}, \mathcal{R}, ad)$$

  with $ad = (iid, \texttt{inner}, \mathcal{D}, attempt)$,
- if *pkg* is the outcome of a fresh dealing, computes its share as

$$s_{\mathcal{R}} \leftarrow \sum_{\mathcal{D} \in \boldsymbol{L'}} s_{\mathcal{D},\mathcal{R}} \ ,$$

  and if *pkg* is the outcome of a resharing, computes its share as

$$s_{\mathcal{R}} \leftarrow \sum_{\mathcal{D} \in \boldsymbol{L'}} \boldsymbol{\Lambda}_{\boldsymbol{L'},\mathcal{D}}(0) \cdot s_{\mathcal{D},\mathcal{R}} \ .$$

Note that the $\mathcal{VESS}.\mathsf{Decrypt}$ algorithm in the second step above will verify each sub-share $s_{\mathcal{D},\mathcal{R}}$ with respect to the polynomial commitment $\boldsymbol{C}_{\mathcal{D}}$ in $ID_{\mathcal{D}}$. Alternatively, one could proceed optimistically and decrypt without performing that check, only to verify the final share $s_{\mathcal{R}}$ against the partial public key $y_{\mathcal{R}}$.

Functionality $\mathcal{F}_{\text{dkg}}$

Upon input $(sid, \texttt{generate})$ from an honest dealer $\mathcal{D} \in \boldsymbol{\mathcal{HD}}$:

- If an output has already been produced for this instance, ignore this input.
- Choose $\boldsymbol{\omega}(X) \leftarrow_{\$} \mathbb{Z}_p[X]_{<t_{\text{R}}}$ and compute $\boldsymbol{C} \leftarrow g^{\boldsymbol{\omega}}$.
- Send a delayed output $(sid, \boldsymbol{C}, \boldsymbol{\omega}(\mathcal{R}))$ to each recipient $\mathcal{R} \in \boldsymbol{\mathcal{R}}$ and send $(sid, \boldsymbol{C})$ to the adversary.

Upon input $(sid, \texttt{reshare}, s_{\mathcal{D}})$ from a dealer $\mathcal{D} \in \boldsymbol{\mathcal{D}}$:

- If an output has already been produced for this instance, ignore this input.
- If $g^{s_{\mathcal{D}}} \neq \boldsymbol{C}^{(\mathcal{D})}$, ignore this input.
- If there is now a set $\boldsymbol{L}$ of t_{D} different dealers that successfully called this interface,
  - interpolate the secret as $s \leftarrow \sum_{\mathcal{D}' \in \boldsymbol{L}} \boldsymbol{\Lambda}_{\boldsymbol{L},\mathcal{D}'}(0) \cdot s_{\mathcal{D}'}$,
  - choose $\boldsymbol{\omega}'(X) \leftarrow_{\$} \mathbb{Z}_p[X]_{<t_{\text{R}}}$ with $\boldsymbol{\omega}'(0) = s$ and compute $\boldsymbol{C}' \leftarrow g^{\boldsymbol{\omega}'}$,
  - send a delayed output $(sid, \boldsymbol{C}', \boldsymbol{\omega}'(\mathcal{R}))$ to each recipient $\mathcal{R} \in \boldsymbol{\mathcal{R}}$ and send $(sid, \boldsymbol{C}')$ to the adversary.

Figure 5: Distributed key generation (DKG) ideal functionality.

6 Security Analysis

6.1 Ideal Functionality

Our ideal functionality $\mathcal{F}_{\text{dkg}}$ is described in Figure 5. It is essentially a variant of Katz' "fully secure" functionality $\widehat{\mathcal{F}}_{\text{KeyGen}}$ [Kat24] that secret-shares a uniformly distributed secret through a uniform polynomial. Our $\mathcal{F}_{\text{dkg}}$ functionality adds the separation between dealers and recipients, as well as an interface to re-share existing shared secrets.

An instance of the $\mathcal{F}_{\text{dkg}}$ can be either a generation instance or a re-sharing instance. In the former case, the dealers can only invoke the `generate` interface that initiates the generation of a fresh secret and secret-shares it among the recipients. In the latter case, the dealers can only invoke the `reshare` interface. They each provide as input a share of an existing, pre-shared secret that is re-shared among the recipients. The pre-shared secret may (but doesn't have to) be the outcome of a previous instance of $\mathcal{F}_{\text{dkg}}$ where the current dealers acted as recipients.

A generation instance of $\mathcal{F}_{\text{dkg}}$ is identified by a session identifier

$$sid \;=\; (\texttt{generate}, \boldsymbol{\mathcal{D}}, f_{\text{D}}, \boldsymbol{\mathcal{R}}, f_{\text{R}}, t_{\text{R}}, sid')$$

that is agreed upon by all participants before the start of the protocol. It is composed of the following items:

- the instance type `generate`, indicating that this is a generation instance that only allows invocation of the `generate` interface,
- the set of dealers $\boldsymbol{\mathcal{D}}$ and the tolerated number of corrupt dealers f_{D}, with $f_{\text{D}} < n_{\text{D}} = |\boldsymbol{\mathcal{D}}|$,
- the set of recipients $\boldsymbol{\mathcal{R}}$, the tolerated number of corrupt recipients f_{R}, and the reconstruction threshold t_{R} under which the generated secret must be secret-shared among the recipients $\boldsymbol{\mathcal{R}}$, with $f_{\text{R}} < t_{\text{R}} \leq n_{\text{R}} = |\boldsymbol{\mathcal{R}}|$,
- an arbitrary string sid', to ensure uniqueness of the session identifier.

As soon as the first honest dealer invokes the `generate` interface, the functionality generates a random polynomial $\boldsymbol{\omega}$ of degree $t_{\text{R}} - 1$ and sends each recipient $\mathcal{R} \in \boldsymbol{\mathcal{R}}$ a delayed output containing its share $\boldsymbol{\omega}(\mathcal{R})$ and the Feldman commitment $\boldsymbol{C} = g^{\boldsymbol{\omega}}$ for the polynomial $\boldsymbol{\omega}$. The adversary obtains $\boldsymbol{C}$ and gets to determine the timing for the delivery of each delayed output.

We recall that a *delayed output* to a party $\mathcal{P}$ means that the functionality hands the adversary the identity of $\mathcal{P}$ and a unique handle for the output, but not its contents. The adversary can arbitrarily schedule the delivery of these outputs by calling a dedicated delivery interface with the output handle.

Similarly, a re-sharing instance of $\mathcal{F}_{\text{dkg}}$ is identified by a tuple

$$sid \;=\; (\texttt{reshare}, \boldsymbol{\mathcal{D}}, f_{\text{D}}, t_{\text{D}}, \boldsymbol{C}, \boldsymbol{\mathcal{R}}, f_{\text{R}}, t_{\text{R}}, sid')$$

with most of the same components as in the generation case, but

- the instance type `reshare` ensures that only the `reshare` interface of this instance can be invoked,
- the reconstruction threshold t_{D} and Feldman commitment $\boldsymbol{C} = g^{\boldsymbol{\omega}}$ using which the existing secret is secret-shared among the set of dealers $\boldsymbol{\mathcal{D}}$, with $f_{\text{D}} < t_{\text{D}} \leq n_{\text{D}}$.

To invoke the `reshare` interface, each dealer $\mathcal{D}$ has to provide a share $s_{\mathcal{D}}$ of a pre-existing secret that is t_{D}-out-of-n_{D} shared among the dealers with Feldman commitment $\boldsymbol{C}$, where t_{D} and $\boldsymbol{C}$ are specified in the session identifier.

When t_{D} dealers $\mathcal{D}$ invoked the `reshare` interface with a valid share $s_{\mathcal{D}}$ such that $g^{s_{\mathcal{D}}} = \boldsymbol{C}^{(\mathcal{D})}$, the functionality reconstructs the secret s through interpolation and chooses a random polynomial $\boldsymbol{\omega'}$ of degree $t_{\text{R}} - 1$ to re-share the secret among the recipients $\boldsymbol{\mathcal{R}}$. It sends a delayed output to each recipient $\mathcal{R} \in \boldsymbol{\mathcal{R}}$ containing its secret share $s'_{\mathcal{R}} = \boldsymbol{\omega'}(\mathcal{R})$ and the Feldman commitment $\boldsymbol{C'} = g^{\boldsymbol{\omega}'}$.

Corruption model. We assume a static corruption model whereby the adversary, at the beginning of the game, chooses f_{D} corrupt dealers $\boldsymbol{\mathcal{CD}} \subseteq \boldsymbol{\mathcal{D}}$ and f_{R} corrupt recipients $\boldsymbol{\mathcal{CR}} \subseteq \boldsymbol{\mathcal{R}}$. We define the set of honest dealers and recipients as $\boldsymbol{\mathcal{HD}} \leftarrow \boldsymbol{\mathcal{D}} \setminus \boldsymbol{\mathcal{CD}}$ and $\boldsymbol{\mathcal{HR}} \leftarrow \boldsymbol{\mathcal{R}} \setminus \boldsymbol{\mathcal{CR}}$, respectively. We also define the set of corrupt parties $\boldsymbol{\mathcal{C}} \leftarrow \boldsymbol{\mathcal{CD}} \cup \boldsymbol{\mathcal{CR}}$ and the set of honest parties $\boldsymbol{\mathcal{H}} \leftarrow \boldsymbol{\mathcal{HD}} \cup \boldsymbol{\mathcal{HR}}$.

In most practical applications, one would set the number of corrupt parties to their maximal values, namely $f_{\text{D}} = t_{\text{D}} - 1$ and $f_{\text{R}} = t_{\text{R}} - 1$. There are situations, however, where smaller values for f_{D} or f_{R} can be appropriate. For example, if the reconstruction threshold is externally fixed, it may be beneficial to accept a lower corruption threshold for a more efficient protocol. Also, consensus protocols are inherently restricted to $f < n/3$, but can benefit greatly from threshold signatures with a reconstruction threshold $t = n - f > 2n/3$ (the so-called *dual-threshold* setting).

The ideal functionality also supports so-called *high-threshold* settings with $t_{\text{D}} > n_{\text{D}} - f_{\text{D}}$ or $t_{\text{R}} > n_{\text{R}} - f_{\text{R}}$. In such cases, completeness cannot be guaranteed, since all honest parties together do not have sufficient information to reconstruct the secret. Nevertheless, high-threshold configurations can be meaningful in applications where completeness is only a secondary concern, for example, in key escrow when the policy intentionally requires a super-majority, or in settings where completeness can always be restored by re-sharing to a different committee of participants.

Completeness. The functionality from Figure 5 guarantees that if an honest party outputs a share, then it is a correct share, but it does not guarantee that

parties receive anything at all in the first place.

In any execution with at most f_{D} corrupt dealers and at most f_{R} corrupt recipients, suppose that eventually all messages sent by honest parties are delivered by the adversary. We say that the DKG protocol satisfies *completeness* if

1. if the protocol instance is a generation instance and at least $f_{\mathrm{D}}+1$ honest dealers invoke `generate`, then every honest recipient $\mathcal{R} \in \boldsymbol{\mathcal{HR}}$ eventually outputs a share $(sid, \boldsymbol{C}, s_{\mathcal{R}})$, and
2. if the protocol instance is a re-sharing instance and at least t_{D} honest dealers $\mathcal{D} \in \boldsymbol{\mathcal{D}}$ invoke `reshare` with inputs $s_{\mathcal{D}}$ such that $\boldsymbol{C}^{(\mathcal{D})} = g^{s_{\mathcal{D}}}$, then every honest recipient $\mathcal{R} \in \boldsymbol{\mathcal{HR}}$ eventually outputs a share $(sid, \boldsymbol{C}', s'_{\mathcal{R}})$.

6.2 Simulator

We present a simulator $\mathcal{S}$ for the DKG protocol suite in Section 5 in the $(\mathcal{F}_{\mathrm{tob}}, \mathcal{F}_{\mathrm{crs}}, \mathcal{F}_{\mathrm{ro}}, \mathcal{F}_{\mathrm{pki}})$-hybrid model. In particular, the common reference string *crs* is modeled through a subfunctionality $\mathcal{F}_{\mathrm{crs}}^{\bar{\mathbb{G}}}$, and the hash functions $\mathsf{H}_{\mathrm{enc}}^{\ell}$, $\mathsf{H}_{\mathrm{exp}}^{t}$, H_{ch}, and $\mathsf{H}_{\mathrm{deal}}^{\ell}$ are modeled as random-oracle subfunctionalities $\mathcal{F}_{\mathrm{ro}}^{\{0,1\}^{\ell}}$, $\mathcal{F}_{\mathrm{ro}}^{\mathbb{Z}_p \times \mathbb{Z}_p^t}$, $\mathcal{F}_{\mathrm{ro}}^{\binom{[N]}{M}}$, and $\mathcal{F}_{\mathrm{ro}}^{\{0,1\}^{\ell}}$ described by Figure 3, respectively. We note that all instances of $\mathcal{F}_{\mathrm{crs}}$ and $\mathcal{F}_{\mathrm{ro}}$ are independent, local functionalities.

The simulator $\mathcal{S}$ generates signature key pairs $(vk_{\mathcal{P}}, sk_{\mathcal{P}})$ and encryption key pairs $(ek_{\mathcal{P}}, dk_{\mathcal{P}})$ for all honest parties $\mathcal{P} \in \boldsymbol{\mathcal{H}}$ and registers $(vk_{\mathcal{P}}, ek_{\mathcal{P}})$ in $\mathcal{F}_{\mathrm{pki}}$. The adversary registers $(vk_{\mathcal{P}}, ek_{\mathcal{P}})$ for all corrupt parties $\mathcal{P} \in \boldsymbol{\mathcal{C}}$.

Let T_{deal} be the table kept by the instance of the $\mathcal{F}_{\mathrm{ro}}^{\{0,1\}^{\ell}}$ functionality that models the random oracle $\mathsf{H}_{\mathrm{deal}}^{\ell}$. The simulator simulates the common reference string *crs* generated by $\mathcal{F}_{\mathrm{crs}}^{\bar{\mathbb{G}}}$ so that it knows its discrete logarithm with respect to $\bar{g}$.

We first focus on a fresh key generation protocol instance. The simulator $\mathcal{S}$ simulates all honest dealers $\mathcal{D} \in \boldsymbol{\mathcal{HD}}$ by letting them broadcast a dealing $(OD_{\mathcal{D}}, EID_{\mathcal{D}})$, where $OD_{\mathcal{D}}$ is an honestly created outer dealing, but $EID_{\mathcal{D}}$ is just a random string of the appropriate length.

When $f_{\mathrm{D}}+1$ validly signed dealings $(OD_{\mathcal{D}}, EID_{\mathcal{D}})$ by different dealers $\mathcal{D} \in \boldsymbol{L}$ have appeared on the broadcast channel, the simulator performs the following two steps for each corrupt dealer $\mathcal{D} \in \boldsymbol{L} \cap \boldsymbol{\mathcal{CD}}$. First, it decrypts the inner dealing $ID_{\mathcal{D}} \leftarrow EID_{\mathcal{D}} \oplus T_{\mathrm{deal}}[\mathcal{D}, z_{\mathcal{D}}]$ by looking up (or creating, if it doesn't yet exist) an entry $T_{\mathrm{deal}}[\mathcal{D}, z_{\mathcal{D}}]$ such that $g^{z_{\mathcal{D}}} = \boldsymbol{D}_{\mathcal{D}}^{(0)}$, where $OD_{\mathcal{D}} = (\boldsymbol{D}_{\mathcal{D}}, h_{\mathcal{D}}, \boldsymbol{\rho}_{\mathcal{D}})$. Second, the simulator verifies the inner dealing $ID_{\mathcal{D}}$ and, if it is valid, recovers the polynomial $\boldsymbol{\omega}_{\mathcal{D}}$ by using the discrete logarithm of *crs* to decrypt the encryption "to the sky" of $\boldsymbol{\omega}'_{\mathcal{D},k}$, computing $\boldsymbol{\omega}_{\mathcal{D}} \leftarrow \boldsymbol{\omega}''_{\mathcal{D},k} - \boldsymbol{\omega}'_{\mathcal{D},k}$, and checking that $\boldsymbol{C}_{\mathcal{D}} = g^{\boldsymbol{\omega}_{\mathcal{D}}}$ for some $k \in S_{\mathcal{D}}$. (If no such k can be found, $\mathcal{S}$ aborts.)

When $\mathcal{S}$ receives from $\mathcal{F}_{\text{dkg}}$ a message $(sid, \boldsymbol{C})$ as well as messages $(sid, \boldsymbol{C}, s_{\mathcal{R}})$ for all corrupt recipients $\mathcal{R} \in \boldsymbol{\mathcal{CR}}$, $\mathcal{S}$ programs the random oracle for H_{deal} with inner dealings by honest dealers so that the final polynomial commitment turns out to $\boldsymbol{C}$. To do so, it first removes from the list $\boldsymbol{L}$ any corrupt dealers who included an invalid inner dealing in their dealings. For all honest dealers $\mathcal{D} \in \boldsymbol{L} \cap \boldsymbol{\mathcal{HR}}$, except one honest dealer $\hat{\mathcal{D}} \in \boldsymbol{L} \cap \boldsymbol{\mathcal{HR}}$, $\mathcal{S}$ generates an honest inner dealing $ID_{\mathcal{D}}$ for a random polynomial $\boldsymbol{\omega}_{\mathcal{D}}$ and programs $T_{\text{deal}}[\mathcal{D}, z_{\mathcal{D}}] \leftarrow ID_{\mathcal{D}} \oplus EID_{\mathcal{D}}$. (Remember that, since there are at most f_{D} corrupt dealers, there must be at least one honest dealer in $\boldsymbol{L}$.) For the final honest dealer $\hat{\mathcal{D}}$, $\mathcal{S}$ creates a simulated dealing with polynomial commitment

$$\boldsymbol{C}_{\hat{\mathcal{D}}} \leftarrow \boldsymbol{C} / \prod_{\mathcal{D} \in \boldsymbol{L} \setminus \{\hat{\mathcal{D}}\}} \boldsymbol{C}_{\mathcal{D}}$$

encrypting to all corrupt recipients $\mathcal{R} \in \boldsymbol{\mathcal{CR}}$

$$s_{\hat{\mathcal{D}},\mathcal{R}} \leftarrow s_{\mathcal{R}} - \sum_{\mathcal{D}' \in \boldsymbol{L} \setminus \{\hat{\mathcal{D}}\}} \omega_{\mathcal{D}'}(\mathcal{R})$$

and encrypting zero to all honest recipients. It does so by creating a simulated transcript for the cut-and-choose protocol by programming H_{ch} and programming $T_{\text{deal}}[\hat{\mathcal{D}}, z_{\hat{\mathcal{D}}}] \leftarrow ID_{\hat{\mathcal{D}}} \oplus EID_{\hat{\mathcal{D}}}$.

The simulator lets the honest dealers perform the rest of the protocol honestly, revealing decryption shares for outer dealings, broadcasting signatures on the dealings package, and sending the resulting certified dealings package to all recipients as prescribed by the protocol.

For a re-sharing protocol instance, the simulator proceeds similarly, letting honest dealers produce a dealing with a valid outer dealing $OD_{\mathcal{D}}$ but with a random encrypted inner dealing $EID_{\mathcal{D}}$. It now waits for validly signed dealings by t_{D} different dealers $\boldsymbol{L}$ for the same value $\boldsymbol{C}'$ to appear on the broadcast channel. It recovers the corrupt dealers' inner dealings as before. If any of the inner dealings are invalid, $\mathcal{S}$ produces honest random inner dealings for all honest dealers in $\boldsymbol{L}$ and programs them into H_{deal}. If all are valid, $\mathcal{S}$ does the same for all honest dealers except one $\hat{\mathcal{D}} \in \boldsymbol{L} \cap \boldsymbol{\mathcal{HD}}$, where it creates a simulated dealing with commitment

$$\boldsymbol{C}_{\hat{\mathcal{D}}} \leftarrow \left(\boldsymbol{C} / \prod_{\mathcal{D} \in \boldsymbol{L} \setminus \{\hat{\mathcal{D}}\}} \boldsymbol{C}_{\mathcal{D}}^{\boldsymbol{\Lambda}_{\boldsymbol{L},\mathcal{D}}(0)} \right)^{1/\boldsymbol{\Lambda}_{\boldsymbol{L},\hat{\mathcal{D}}}(0)}$$

encrypting to all corrupt recipients $\mathcal{R} \in \boldsymbol{\mathcal{CR}}$

$$s_{\hat{\mathcal{D}},\mathcal{R}} \leftarrow \left(s_{\mathcal{R}} - \sum_{\mathcal{D}' \in \boldsymbol{L} \setminus \{\hat{\mathcal{D}}\}} \boldsymbol{\Lambda}_{\boldsymbol{L},\mathcal{D}'}(0) \cdot \omega_{\mathcal{D}'}(\mathcal{R}) \right) / \boldsymbol{\Lambda}_{\boldsymbol{L},\hat{\mathcal{D}}}(0)$$

and encrypting zero to all honest recipients.

6.3 Reductions

In this subsection, we prove the following theorem:

Theorem 1 *The $\mathcal{DKG}$ protocol suite from Section 5 securely realizes $\mathcal{F}_{\text{dkg}}$ in the $\boldsymbol{\mathcal{F}} = (\mathcal{F}_{\text{tob}}, \mathcal{F}_{\text{crs}}, \mathcal{F}_{\text{ro}}, \mathcal{F}_{\text{pki}})$-hybrid model if the ICDH1 assumption holds in $\bar{\mathbb{G}}$, the discrete-logarithm assumption holds in $\mathbb{G}$, and $\mathcal{DS}$ is* uf-cma *secure.*

More specifically, for all efficient environments $\mathcal{E}$ and adversaries $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ and efficient algorithms $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$ such that

$$\begin{aligned}|\,\mathbf{Ideal}^{\mathcal{F}_{\text{dkg}}}_{\mathcal{E},\mathcal{S}}(\kappa) - \mathbf{Hybrid}^{\mathcal{DKG}/\boldsymbol{\mathcal{F}}}_{\mathcal{E},\mathcal{A}}(\kappa)\,| \;\le\; & n_{\text{D}} \cdot \mathbf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathcal{DS},\mathcal{B}_1}(\kappa) + 2 \cdot \mathbf{Adv}^{\mathsf{icdh1}}_{\bar{\mathbb{G}},\mathcal{B}_2}(\kappa) \\ & + \mathbf{Adv}^{\mathsf{dlog}}_{\mathbb{G},\mathcal{B}_3}(\kappa) + \frac{q_{\text{H}}^2 + n_{\text{D}}^2 + 2n_{\text{D}}^2 q_{\text{H}}}{2^{2\kappa}} + \frac{q_{\text{H}}}{\binom{N}{M}} + \frac{2n_{\text{D}}^2 N q_{\text{H}}}{2^{\kappa}} + \frac{2n_{\text{D}}^2 q_{\text{H}}}{p}\;,\end{aligned}$$

where $n_{\text{D}} = |\boldsymbol{\mathcal{D}}|$ is the number of dealers, N and M are parameters of H_{ch}, q_{H} is the number of random-oracle queries made by $\mathcal{A}$, p is the order or $\mathbb{G}$, and κ is the security parameter.

We describe a sequence of games with the real-world adversary $\mathcal{A}$ and environment $\mathcal{E}$, where the goal of the environment is to distinguish in which of two subsequent games it's executing. Let ϵ_i be the probability that $\mathcal{E}$ outputs 1 in Game i.

Game 0: Hybrid world. This is the hybrid-world experiment, where the hybrid-world adversary $\mathcal{A}$ controls all corrupt parties in a session *sid* of the DKG protocol suite. The experiment runs all real-world honest parties using the real $\mathcal{DKG}$ protocol suite as well as the subfunctionalities $\boldsymbol{\mathcal{G}} = (\mathcal{F}_{\text{tob}}, \mathcal{F}_{\text{crs}}, \mathcal{F}_{\text{ro}}, \mathcal{F}_{\text{pki}})$ based on inputs from the environment $\mathcal{E}$. We have that

$$\epsilon_0 \;=\; \mathbf{Hybrid}^{\mathcal{DKG}/\boldsymbol{\mathcal{G}}}_{\mathcal{E},\mathcal{A}}(\kappa)\;. \tag{4}$$

Game 1: Signature unforgeability. In this game, the experiment aborts when a message is submitted to the broadcast channel that contains a valid signature by an honest party, but that honest party never signed this message. Given such an environment $\mathcal{E}$ and adversary $\mathcal{A}$, one can easily construct an algorithm $\mathcal{B}_1$ that breaks the existential unforgeability under chosen-message attack (uf-cma) of the digital signature scheme $\mathcal{DS}$ by choosing one of the $n_{\text{D}} - f_{\text{D}}$ honest dealers at random as the "challenge" dealer $\hat{\mathcal{D}}$, using its signing oracle to simulate signatures by $\hat{\mathcal{D}}$, and outputting the forged signature as soon as one is submitted to the broadcast channel. We have that

$$\mathbf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathcal{DS},\mathcal{B}_1}(\kappa) \;\ge\; \frac{|\,\epsilon_1 - \epsilon_0\,|}{n_{\text{D}}}$$

or equivalently,

$$|\,\epsilon_1 - \epsilon_0\,| \;\le\; n_{\text{D}} \cdot \mathbf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathcal{DS},\mathcal{B}_1}(\kappa)\;. \tag{5}$$

Game 2: ZKP soundness. The experiment now sets the common reference string that is generated by $\mathcal{F}_{\text{crs}}$ and used by the MRE scheme to $crs \leftarrow \bar{g}^{dk_{crs}}$ for a random $dk_{crs} \leftarrow_{\$} \mathbb{Z}_{\bar{p}}$, where $\bar{\mathbb{G}}$, $\bar{p}$, and $\bar{g}$ are the group, group order, and generator used by the MRE scheme, respectively. This change is purely conceptual and doesn't affect the adversary's view at all.

Also, this experiment aborts whenever a valid inner or outer dealing $D = (\boldsymbol{C}, h, \boldsymbol{\rho})$ created by a corrupt dealer cannot be validly decrypted by some honest recipient $\mathcal{P}$ or by dk_{crs}. (Note that in the case of an outer dealing, the "recipients" of the dealing are actually the dealers of the DKG protocol instance; hence, we refer to them here as a generic party $\mathcal{P}$.) More particularly, it aborts whenever the dealing is valid according to $\mathcal{VESS}.\mathsf{VerifyDealing}$, but fails to decrypt to a share $s_{\mathcal{P}}$ such that $\boldsymbol{C}^{(\mathcal{P})} = g^{s_{\mathcal{P}}}$ for an honest recipient $\mathcal{P}$, or the encryption to the sky fails to decrypt by dk_{crs} to a polynomial $\boldsymbol{\omega}$ such that $\boldsymbol{C} = g^{\boldsymbol{\omega}}$.

The probability that the experiment aborts is bounded by the soundness of the cut-and-choose zero-knowledge protocol in the VESS scheme. Overall, we therefore have that

$$|\,\epsilon_2 - \epsilon_1\,| \;\leq\; \frac{q_{\text{H}}^2}{2^{2\kappa}} + \frac{q_{\text{H}}}{\binom{N}{M}} \;. \tag{6}$$

Game 3: Simulate ZKPs. In this game, when an honest dealer calls $\mathcal{VESS}.\mathsf{Deal}$, it simulates the cut-and-choose zero-knowledge proof by programming the H_{ch} random oracle. It does so by choosing a random set $S \leftarrow_{\$} \binom{[N]}{M}$ and

- for $k \in S$, choosing a random $\boldsymbol{\omega}''_k \leftarrow_{\$} \mathbb{Z}_p^t$, computing $\boldsymbol{C}'_k \leftarrow g^{\boldsymbol{\omega}''_k}/\boldsymbol{C}$, and computing $\boldsymbol{E}_k$ as an encryption of $\boldsymbol{\omega}''(\mathcal{P}) - \boldsymbol{\omega}(\mathcal{P})$ for all corrupt recipients $\mathcal{P}$ of the dealing, and as an encryption of zeroes for all honest recipients $\mathcal{P}$ and for the CRS,
- for $k \notin S$, by choosing random $seed_k \leftarrow_{\$} \{0,1\}^\kappa$ and by computing $\boldsymbol{C}'_k$ and $\boldsymbol{E}_k$ as in the honest protocol, i.e., computing $(r_k, \boldsymbol{\omega}'_k) \leftarrow \mathsf{H}_{\text{exp}}^t(k, seed_k, ad)$ and using the polynomial $\boldsymbol{\omega}'_k$ and randomness r_k to compute $\boldsymbol{C}'_k, \boldsymbol{E}_k$.

It then computes $h \leftarrow \mathsf{H}_{\text{comp}}(ad, \boldsymbol{C}'_1, \boldsymbol{E}_1, \ldots, \boldsymbol{C}'_N, \boldsymbol{E}_N)$ and programs H_{ch} so that $\mathsf{H}_{\text{ch}}(ad, \boldsymbol{C}, h) = S$.

This programming will fail if $\mathsf{H}_{\text{ch}}(ad, \boldsymbol{C}, h)$ was already defined. If h is "fresh", meaning the random-oracle query $\mathsf{H}_{\text{comp}}(ad, \boldsymbol{C}'_1, \boldsymbol{E}_1, \ldots, \boldsymbol{C}'_N, \boldsymbol{E}_N)$ was new when the experiment made it in the simulation, then the probability of any of the at most q_{H} entries in the table for H_{ch} containing h is at most $q_{\text{H}}/2^{2\kappa}$. If $(r_k, \boldsymbol{\omega}'_k)$ is fresh, meaning that $\mathsf{H}_{\text{exp}}^t(k, seed_k, ad)$ was undefined prior to the simulation, then the probability of h not being fresh is at most $q_{\text{H}}/p^{t+1} \leq q_{\text{H}}/p$. The probability of $(r_k, \boldsymbol{\omega}'_k)$ not being fresh is bounded by the probability of the event that one of the at most q_{H} entries in the table for H_{exp} already contains the random seed $seed_k$, i.e., by $q_{\text{H}}/2^\kappa$.

Each dealing requires H_{ch} and $\mathsf{H}_{\mathrm{comp}}$ to be programmed once and $\mathsf{H}_{\mathrm{exp}}$ to be programmed $N - M$ times, so that the programming for a single dealing fails with probability at most

$$\frac{q_{\mathrm{H}}}{2^{2\kappa}} + \frac{q_{\mathrm{H}}}{p} + (N - M) \cdot \frac{q_{\mathrm{H}}}{2^{\kappa}} \ .$$

The $n_{\mathrm{D}} - f_{\mathrm{D}}$ honest dealers each broadcast at most $2(f_{\mathrm{D}} + 1)$ VESS dealings (one inner and one outer at each iteration, with up to f_{D} retries in a re-sharing). The overall probability that the random-oracle programming goes wrong in any of the simulations is therefore bounded by

$$2(f_{\mathrm{D}}+1)(n_{\mathrm{D}} - f_{\mathrm{D}}) \left(\frac{q_{\mathrm{H}}}{2^{2\kappa}} + \frac{q_{\mathrm{H}}}{p} + (N - M) \cdot \frac{q_{\mathrm{H}}}{2^{\kappa}} \right) \ \leq \ 2n_{\mathrm{D}}^2 \left(\frac{q_{\mathrm{H}}}{2^{2\kappa}} + \frac{q_{\mathrm{H}}}{p} + \frac{N q_{\mathrm{H}}}{2^{\kappa}} \right)$$

where we use the fact that $0 \leq f_{\mathrm{D}} < n_{\mathrm{D}}$ and $M, q_{\mathrm{H}} \geq 0$, so that

$$| \, \epsilon_3 - \epsilon_2 \, | \ \leq \ \frac{2n_{\mathrm{D}}^2 q_{\mathrm{H}}}{2^{2\kappa}} + \frac{2n_{\mathrm{D}}^2 q_{\mathrm{H}}}{p} + \frac{2n_{\mathrm{D}}^2 N q_{\mathrm{H}}}{2^{\kappa}} \ . \tag{7}$$

Game 4: Encrypt bogus shares to honest parties. This game, whenever an honest dealer $\mathcal{D}$ calls $\mathcal{VESS}.\mathsf{Deal}$, the experiment replaces the MRE ciphertext components in unopened ciphertexts (i.e., where $k \in S$ in the simulated zero-knowledge proof) intended for honest parties with random strings. More particularly, for $k \in S$, it replaces the share components in the MRE $\boldsymbol{E}$ that correspond to an honest party's key with random strings of the appropriate length. This doesn't affect the creation of the rest of the zero-knowledge proof, as that is already simulated since Game 3.

Given an environment $\mathcal{E}$ and an adversary $\mathcal{A}$ that can distinguish the experiments in Game 3 and Game 4, we construct the following algorithm $\mathcal{B}$ that solves the ICDH1 problem in $\bar{\mathbb{G}}$.

Note that because we're reducing straight from ICDH1, rather than modeling MRE as a separate primitive and reducing from its CPA or CCA security, we can actually do without the typical hybrid argument and their associated loss in tightness. Instead, we exploit the random self-reducibility of the Diffie-Hellman problem to embed the challenge B in *every* honest party's public key $ek_{\mathcal{P}}$ and the challenge A in *every* ciphertext at the same time.

On input $(A = \bar{g}^a, B = \bar{g}^b)$, $\mathcal{B}$ sets $crs \leftarrow B$ and sets the encryption key of each honest dealer $\mathcal{D}$ to $ek_{\mathcal{D}} \leftarrow B^{dk_{\mathcal{D}}}$, where $dk_{\mathcal{D}} \leftarrow_{\$} \mathbb{Z}_{\bar{p}}$, where $\bar{\mathbb{G}}$, $\bar{p}$, and $\bar{g}$ are the group, order, and generator used in the MRE scheme.

For each time i that an honest dealer $\mathcal{D}$ invokes $\mathcal{MRE}.\mathsf{Encrypt}(\boldsymbol{ek}, \boldsymbol{m}, ad)$ in an iteration $k \in S$ of the zero-knowledge proof, it chooses $\rho_{\mathcal{D},i} \leftarrow_{\$} \mathbb{Z}_p$. It sets $E_0 \leftarrow A^{\rho_{\mathcal{D},i}}$ and uses random strings of the appropriate length for the components $E_1, \ldots, E_n$ and keeps a record $(ad, \boldsymbol{ek}, E_0, \rho_{\mathcal{D},i}, \boldsymbol{m})$.

Note that the honest parties don't know their own decryption keys, so cannot decrypt the ciphertexts in incoming dealings using the regular $\mathcal{MRE}.\mathsf{Decrypt}$

algorithm. Instead, for dealings created by honest parties, they decrypt using the plaintext messages $\boldsymbol{m}$ in the stored record. For dealings created by corrupt parties, $\mathcal{B}$ uses dk_{crs} to decrypt the polynomial $\boldsymbol{\omega}'_k$ encrypted to the CRS and computes the share of honest party $\mathcal{P}$ as $s'_{k,\mathcal{P}} \leftarrow \boldsymbol{\omega}'_k(\mathcal{P})$.

Whenever $\mathcal{A}$ makes a random-oracle query $\mathsf{H}_{\mathrm{enc}}(ad, j, ek_j, E_0, F)$, $\mathcal{B}_2$ checks whether it has a stored record $(ad, \boldsymbol{ek}, E_0, \rho, \boldsymbol{m})$. If not, then $\mathcal{B}_2$ simply returns a random string as usual. If so, then it makes a query $\mathsf{DDH}_1(ek_j^{\rho}, F)$ to its Diffie-Hellman oracle. If the oracle returns 0, $\mathcal{B}_2$ returns a random string as the random-oracle response. If the oracle returns 1 and ek_j is an honest party $\mathcal{D}$'s key in $\boldsymbol{ek}$, then $\mathcal{B}_2$ halts, outputting $C \leftarrow F^{1/(\rho \cdot dk_{\mathcal{D}})}$ as its solution to the ICDH1 problem. If the oracle returns 1 and $ek_j = crs$, then $\mathcal{B}_2$ also halts, outputting $C \leftarrow F^{1/(\rho \cdot dk_{crs})}$ as its solution. If the oracle returns 1 and ek_j is the public key of a corrupt party $\mathcal{P} \in \boldsymbol{\mathcal{C}}$ in $\boldsymbol{ek}$, then $\mathcal{B}_2$ programs the random-oracle to respond $E_j \oplus m_{\mathcal{P}}$, where $m_{\mathcal{P}}$ is the message that was supposed to be encrypted to $\mathcal{P}$.

It is clear that if $\mathcal{A}$ never makes a random-oracle query $\mathsf{H}_{\mathrm{enc}}(ad, j, ek, E_0, F)$ that causes $\mathcal{B}_2$ to halt, then its view is independent of the messages $m_j = E_j \oplus \mathsf{H}_{\mathrm{enc}}(ad, j, ek, E_0, F)$. If it does make such a query, then $\mathcal{B}_2$ always wins the ICDH1 game. We therefore have that

$$|\,\epsilon_4 - \epsilon_3\,| \;\leq\; \mathbf{Adv}^{\mathsf{icdh1}}_{\bar{\mathbb{G}},\mathcal{B}_2}(\kappa)\;. \tag{8}$$

Note that we could have avoided using the encryption to the sky in this game hop by proving security under the ICDH2 assumption instead of the ICDH1 assumption. We could have used the DDH_2 oracle to decrypt honest parties' shares in corrupt parties' dealings by finding a random-oracle query $\mathsf{H}_{\mathrm{enc}}(ad, j, ek, E_0, F)$ such that $\mathsf{DDH}_2(E_0^{dk_{\mathcal{P}}}, F) = 1$. That wouldn't have eliminated the need for encryption to the sky, however: in Game 7, we'll see that the simulator online-extracts corrupt parties' shares from corrupt parties' dealings in order to ensure that corrupt parties are dealt unbiased shares.

Game 5: Encrypt bogus polynomials to CRS. This game, whenever an honest dealer $\mathcal{D}$ calls $\mathcal{VESS}.\mathsf{Deal}$, the experiment replaces the component in unopened MRE ciphertexts that encrypts a polynomial $\langle \boldsymbol{\omega}'_k \rangle$ to the CRS with a random string. More particularly, for $k \in S$, it uses a random string E_{n+1} of the appropriate length for the last encryption-to-the-sky component of the ciphertext $\boldsymbol{E} = (E_0, \ldots, E_{n+1})$. Again, this change doesn't affect the zero-knowledge proof, as it is already simulated.

Given an environment $\mathcal{E}$ and an adversary $\mathcal{A}$ that distinguish between Game 4 and Game 5, consider the following algorithm $\mathcal{B}'_2$ solving the ICDH1 problem in $\bar{\mathbb{G}}$. On input $(A = \bar{g}^a, B = \bar{g}^b)$, $\mathcal{B}'_2$ sets $crs \leftarrow B$.

When an honest dealer calls

$$\mathcal{MRE}.\mathsf{Encrypt}(\boldsymbol{ek}, \boldsymbol{m}, ad)$$

in an iteration $k \in S$ of the zero-knowledge proof, it chooses $\rho \leftarrow_\$ \mathbb{Z}_p$, sets $E_0 \leftarrow A^\rho$, uses a random string of the appropriate length for the components $E_1, \ldots, E_{n+1}$, and keeps a record $(ad, \boldsymbol{ek}, E_0, \rho, \boldsymbol{m})$.

Whenever $\mathcal{A}$ makes a random-oracle query $\mathsf{H}_{\mathrm{enc}}(ad, j, ek_j, E_0, F)$, $\mathcal{B}_2'$ checks whether it has a stored record $(ad, \boldsymbol{ek}, E_0, \rho, \boldsymbol{m})$ for some $\boldsymbol{m}$ with $ek_j \in \boldsymbol{ek}$. If not, $\mathcal{B}_2'$ returns a random string. If so, it makes a query $\mathsf{DDH}_1(ek_j^\rho, F)$ to its Diffie-Hellman oracle. If the oracle returns 0, $\mathcal{B}_2'$ returns a random string. If the oracle returns 1 and $ek_j \neq crs$, then $\mathcal{B}_2'$ programs the random-oracle to respond $E_j \oplus m_j$. If the oracle returns 1, $j = n+1$ and $ek_j = crs$, then $\mathcal{B}_2'$ halts, outputting $C \leftarrow F^{1/\rho}$ as its solution to the ICDH1 problem.

It is clear that if $\mathcal{A}$ never makes a random-oracle query $\mathsf{H}_{\mathrm{enc}}(ad, n+1, crs, E_0, F)$ that causes $\mathcal{B}_2'$ to halt, then its view is independent of the messages encrypted to the CRS. As soon as it does make such a query, $\mathcal{B}_2'$ wins the ICDH1 game, so that

$$|\,\epsilon_5 - \epsilon_4\,| \;\leq\; \mathbf{Adv}^{\mathsf{icdh1}}_{\mathbb{G},\mathcal{B}_2'}(\kappa)\;. \tag{9}$$

Game 6: Lazily program inner dealings. In this game, all honest dealers replace the encrypted inner dealing *EID* in their dealings $D = (OD, EID)$ with a random string of the appropriate length. Only when the first honest dealer broadcasts its decryption shares for the outer dealing does the experiment program the random oracle $\mathsf{H}_{\mathrm{deal}}$ to match the actual inner dealings.

The adversary's view is identical to the previous game, unless it makes a random-oracle query $\mathsf{H}_{\mathrm{deal}}(sid, \mathcal{D}, z_{\mathcal{D}})$ *before* an honest dealer broadcasts its decryption shares. We show that any environment $\mathcal{E}$ and adversary $\mathcal{A}$ that do so can be turned into an algorithm $\mathcal{B}_3$ solving the discrete-logarithm problem in $\mathbb{G}$.

On input $Y \in \mathbb{G}$, $\mathcal{B}_3$ runs $\mathcal{E}$ and $\mathcal{A}$ as in Game 5, but lets each honest dealer $\mathcal{D} \in \boldsymbol{\mathcal{HD}}$ create its outer dealing $OD_{\mathcal{D}}$ as follows. Rather than following $\mathcal{VESS}.\mathsf{Deal}$ by encrypting a random polynomial $\boldsymbol{\omega}$ and computing $\boldsymbol{C} \leftarrow g^{\boldsymbol{\omega}}$, it chooses random shares $z_{\mathcal{D},\mathcal{D}'} \leftarrow_\$ \mathbb{Z}_p$ for all corrupt dealers $\mathcal{D}' \in \boldsymbol{\mathcal{CD}}$ and a randomizer $\rho_{\mathcal{D}} \leftarrow_\$ \mathbb{Z}_p$, computes $\boldsymbol{C}$ by interpolation in the exponent as

$$\boldsymbol{C} \leftarrow Y^{\rho_{\mathcal{D}} \cdot \boldsymbol{\Lambda}_{\boldsymbol{\mathcal{C}}',0}} \cdot \prod_{\mathcal{D}' \in \boldsymbol{\mathcal{C}} \cap \boldsymbol{\mathcal{D}}} g^{z_{\mathcal{D},\mathcal{D}'} \cdot \boldsymbol{\Lambda}_{\boldsymbol{\mathcal{C}}',\mathcal{D}'}}\;,$$

and encrypts the correct shares $z_{\mathcal{D},\mathcal{D}'}$ to all corrupt players. Then $\mathcal{B}_3$ uses bogus encryptions for all honest parties and the CRS as in Games 4 and 5. Note that it thereby implicitly defines $z_{\mathcal{D}} = \rho_{\mathcal{D}} \cdot \log_g Y$ and creates a VESS for a polynomial $\boldsymbol{\omega}$ defined by $\boldsymbol{\omega}(0) = z_{\mathcal{D}}$ and $\boldsymbol{\omega}(\mathcal{D}') = z_{\mathcal{D},\mathcal{D}'}$ for all corrupt dealers $\mathcal{D}'$.

When the adversary makes a random-oracle query $\mathsf{H}^{\ell}_{\mathrm{deal}}(iid, \mathcal{D}, z_{\mathcal{D}})$ with $g^{z_{\mathcal{D}}} = Y^{\rho_{\mathcal{D}}}$, algorithm $\mathcal{B}_3$ outputs $z_{\mathcal{D}}/\rho_{\mathcal{D}}$ as the discrete logarithm of Y.

Algorithm $\mathcal{B}_3$ succeeds in solving the discrete logarithm problem whenever $\mathcal{A}$ makes a random-oracle query that would allow $\mathcal{E}$ to distinguish this game

from the previous one. We therefore have that

$$|\,\epsilon_6 - \epsilon_5\,| \;\leq\; \mathbf{Adv}^{\mathsf{dlog}}_{\mathbb{G},\mathcal{B}_3}(\kappa)\;. \tag{10}$$

Game 7: Force the final outcome. At the beginning of this game, the experiment chooses a random polynomial $\boldsymbol{\omega}(X) \leftarrow_{\$} \mathbb{Z}_p[X]_{<t_{\mathrm{R}}}$ and ensures that the outcome of the protocol is a dealing with commitment $\boldsymbol{C} = g^{\boldsymbol{\omega}}$ where all corrupt recipients $\mathcal{R} \in \boldsymbol{\mathcal{CR}}$ receive a valid share $s_{\mathcal{R}} = \boldsymbol{\omega}(\mathcal{R})$.

It does so by using random strings for the honest dealers' encrypted inner dealings *EID* as in Game 6 and by extracting the corrupt dealers' inner dealings as $ID_{\mathcal{D}} \leftarrow EID_{\mathcal{D}} \oplus \mathsf{H}^{\ell}_{\mathrm{deal}}(iid, \mathcal{D}, z_{\mathcal{D}})$ for $\mathcal{D} \in \boldsymbol{\mathcal{CD}}$ as soon as their dealings appear on the broadcast channel, and before any decryption shares become known. Note that the adversary must have made the query $\mathsf{H}^{\ell}_{\mathrm{deal}}(iid, \mathcal{D}, z_{\mathcal{D}})$ when it created the dealing, otherwise the inner dealing will be a random string and will be invalid with overwhelming probability; merely the probability that the value h in the dealing matches the output of $\mathsf{H}_{\mathrm{comp}}$ is at most $1/2^{2\kappa}$, or at most $f_{\mathrm{D}}^2/2^{2\kappa} \leq n_{\mathrm{D}}^2/2^{2\kappa}$ over all at most $f_{\mathrm{D}}+1$ iterations of the protocol involving at most f_{D} corrupt dealers each. Also note that the experiment can easily find the correct entry by computing $g^{z_{\mathcal{D}}}$ and checking whether $\boldsymbol{C}^{(0)} = g^{z_{\mathcal{D}}}$ where $\boldsymbol{C}$ is taken from the outer dealing.

From the inner dealing $ID_{\mathcal{D}}$, it extracts the polynomial $\boldsymbol{\omega}_{\mathcal{D}}$ that was secret-shared in $ID_{\mathcal{D}}$ using dk_{crs}. It then programs the random oracle $\mathsf{H}_{\mathrm{deal}}$ to fix the honest dealers' dealings *after* seeing the corrupt dealers' dealings. Let $\boldsymbol{L}$ be the first $f_{\mathrm{D}}+1$ (in a fresh dealing) or t_{D} (in a re-sharing) dealings to appear on the broadcast channel. The experiment lets all honest dealers $\mathcal{D} \in \boldsymbol{L}$ share random polynomials $\boldsymbol{\omega}_{\mathcal{D}}$, except for one honest dealer $\hat{\mathcal{D}} \in \boldsymbol{L}$ where it sets

$$\boldsymbol{\omega}_{\hat{\mathcal{D}}} \leftarrow \boldsymbol{\omega} - \sum_{\mathcal{D} \in \boldsymbol{L} \setminus \{\hat{\mathcal{D}}\}} \boldsymbol{\omega}_{\mathcal{D}}$$

for a fresh dealing, or

$$\boldsymbol{\omega}_{\hat{\mathcal{D}}} \leftarrow \frac{\boldsymbol{\omega} - \sum_{\mathcal{D} \in L \setminus \{\hat{\mathcal{D}}\}} \boldsymbol{\Lambda}_{L,\mathcal{D}}(0) \cdot \boldsymbol{\omega}_{\mathcal{D}}}{\boldsymbol{\Lambda}_{L,\hat{\mathcal{D}}}(0)}$$

for a re-sharing.

This change does not impact the adversary's view beyond the probability of random strings turning out to be valid dealings mentioned above, so we have that

$$|\,\epsilon_7 - \epsilon_6\,| \;\leq\; \frac{n_{\mathrm{D}}^2}{2^{2\kappa}}\;. \tag{11}$$

Game 8: Simulate based on C and corrupt shares. Note that, due to Game 4, the experiment doesn't let honest dealers encrypt real shares to other honest users, or real polynomials to *crs*. We can therefore perform the full

simulation in Game 6 based on just the corrupt parties' shares $\{s_{\mathcal{R}}\}_{\mathcal{R}\in\boldsymbol{\mathcal{R}}\cap\boldsymbol{\mathcal{C}}}$ and the polynomial commitment $\boldsymbol{C}$.

More particularly, for a fresh dealing, the experiment can compute the polynomial commitment $\boldsymbol{C}_{\hat{\mathcal{D}}}$ for the honest dealer $\hat{\mathcal{D}}$ and the shares $s_{\hat{\mathcal{D}},\mathcal{R}}$ it sends to corrupt recipients $\mathcal{R}$ as

$$\boldsymbol{C}_{\hat{\mathcal{D}}} \leftarrow \boldsymbol{C} - \prod_{\mathcal{D}\in\boldsymbol{L}\setminus\{\hat{\mathcal{D}}\}} \boldsymbol{C}_{\mathcal{D}} \quad \text{and} \quad s_{\hat{\mathcal{D}},\mathcal{R}} \leftarrow s_{\mathcal{R}} - \sum_{\mathcal{D}\in\boldsymbol{L}\setminus\{\hat{\mathcal{D}}\}} \boldsymbol{\omega}_{\mathcal{D}}(\mathcal{R}) \ ,$$

respectively, and for a re-sharing, as

$$\boldsymbol{C}_{\hat{\mathcal{D}}} \leftarrow \left(\boldsymbol{C} / \prod_{\mathcal{D}\in L\setminus\{\hat{\mathcal{D}}\}} \boldsymbol{C}_{\mathcal{D}}^{\boldsymbol{\Lambda}_{L,\mathcal{D}}(0)} \right)^{1/\boldsymbol{\Lambda}_{L,\hat{\mathcal{D}}}(0)}$$

and

$$s_{\hat{\mathcal{D}},\mathcal{R}} \leftarrow \frac{s_{\mathcal{R}} - \sum_{\mathcal{D}\in L\setminus\{\hat{\mathcal{D}}\}} \boldsymbol{\Lambda}_{L,\mathcal{D}}(0)\cdot s_{\mathcal{D},\mathcal{R}}}{\boldsymbol{\Lambda}_{L,\hat{\mathcal{D}}}(0)} \ .$$

This game change is purely conceptual, so that

$$|\,\epsilon_8 - \epsilon_7\,| \ = \ 0 \ . \tag{12}$$

Game 9: Ideal world. From the previous game, it is easy to carve out the simulator $\mathcal{S}$ from Section 6.2 that simulates the full protocol based on a given polynomial commitment $\boldsymbol{C}$ and the corrupt recipients' shares $s_{\mathcal{R}}$. We therefore have that

$$\mathbf{Ideal}_{\mathcal{E},\mathcal{S}}^{\mathcal{F}_{\mathrm{dkg}}}(\kappa) \ = \ \epsilon_8 \ . \tag{13}$$

By adding up Equations (4–13), we get

$$\begin{aligned} |\,\mathbf{Ideal}_{\mathcal{E},\mathcal{S}}^{\mathcal{F}_{\mathrm{dkg}}} - \mathbf{Hybrid}_{\mathcal{E},\mathcal{A}}^{\mathcal{DKG}/\boldsymbol{\mathcal{G}}}\,| \ \leq \ & n_{\mathrm{D}}\cdot\mathbf{Adv}_{\mathcal{DS},\mathcal{B}_1}^{\mathsf{uf\text{-}cma}}(\kappa) + 2\cdot\mathbf{Adv}_{\mathbb{G},\mathcal{B}_2}^{\mathsf{icdh1}}(\kappa) \\ & + \mathbf{Adv}_{\mathbb{G},\mathcal{B}_3}^{\mathsf{dlog}}(\kappa) + \frac{q_{\mathrm{H}}^2 + n_{\mathrm{D}}^2 + 2n_{\mathrm{D}}^2 q_{\mathrm{H}}}{2^{2\kappa}} + \frac{q_{\mathrm{H}}}{\binom{N}{M}} + \frac{2n_{\mathrm{D}}^2 N q_{\mathrm{H}}}{2^{\kappa}} + \frac{2n_{\mathrm{D}}^2 N q_{\mathrm{H}}}{p} \end{aligned} \tag{14}$$

as stated in the theorem, where $\mathcal{B}_2$ is whichever of the algorithms $\mathcal{B}_2$ and $\mathcal{B}_2'$ above that has the highest advantage.

6.4 Completeness

Theorem 2 *If the DKG protocol suite from Section 5 is instantiated with concrete protocols that securely realize $\mathcal{F}_{\mathrm{pki}}$ and $\mathcal{F}_{\mathrm{tob}}$ and that satisfy their respective completeness properties, then the resulting DKG protocol satisfies the completeness property from Section 6.1.*

Proof. We first prove the first part of the completeness property, namely that in a generation instance where at least $f_\mathrm{D}+1$ honest dealers $\bar{\boldsymbol{\mathcal{D}}} \subseteq \boldsymbol{\mathcal{D}}$ invoke `generate`, then all honest recipients eventually receive a share.

All dealers $\bar{\mathcal{D}} \in \bar{\boldsymbol{\mathcal{D}}}$ who invoke `generate` will first broadcast an honestly generated encrypted dealing $(OD_{\bar{\mathcal{D}}}, EID_{\bar{\mathcal{D}}})$. By the completeness of the total-order broadcast protocol, all dealers in $\boldsymbol{\mathcal{D}'}$ therefore eventually receive at least $f_\mathrm{D}+1$ encrypted dealings.

By the correctness of the broadcast protocol (i.e., because it securely realizes $\mathcal{F}_\mathrm{tob}$), all dealers receive messages in the same order, so all dealers in $\bar{\boldsymbol{\mathcal{D}}}$ agree on the list of first $f_\mathrm{D}+1$ valid encrypted dealings $\boldsymbol{L}$. Note that $\boldsymbol{L}$ may but doesn't have to contain dealings by dealers in $\bar{\boldsymbol{\mathcal{D}}}$.

Each dealer $\bar{\mathcal{D}} \in \bar{\boldsymbol{\mathcal{D}}}$ then decrypts its shares from the outer dealings in $\boldsymbol{L}$ and broadcasts $(z_{\mathcal{D}',\bar{\mathcal{D}}})_{\mathcal{D}' \in \boldsymbol{L}}$. By the validity of the outer dealings and the soundness of the VESS scheme, all of these shares $z_{\mathcal{D}',\bar{\mathcal{D}}}$ are valid in the sense that they satisfy $\mathcal{VESS}.\mathsf{VerifyShare}$.

By the completeness of the broadcast protocol, all dealers in $\bar{\boldsymbol{\mathcal{D}}}$ eventually receive $f_\mathrm{D}+1$ valid sets of decryption key shares from different dealers that can be combined to recover the inner dealings $ID_{\mathcal{D}'}$, $\mathcal{D}' \in \boldsymbol{L}$. By verifying the inner dealings using $\mathcal{VESS}.\mathsf{VerifyDealing}$, they all compose the same list $\boldsymbol{L'} \subseteq \boldsymbol{L}$ of inner dealings $ID_{\mathcal{D}'}$, $\mathcal{D}' \in \boldsymbol{L'}$.

Since the original list $\boldsymbol{L}$ contained $f+1$ encrypted dealings by different dealers, at least one of them must have originated from an honest dealer. At least one inner dealing must therefore be valid as well, so $\boldsymbol{L'}$ is non-empty, and all honest dealers agree on this set $\boldsymbol{L'}$. All $f_\mathrm{D}+1$ dealers in $\bar{\boldsymbol{\mathcal{D}}}$ eventually receive this set $\boldsymbol{L'}$ and broadcast their signatures on it, so that eventually all honest dealers receive at least $f_\mathrm{D}+1$ valid signatures and can compose the dealings package pkg.

Finally, by the completeness of the VESS scheme, all honest recipients can recover a valid share from the dealings in pkg.

The second part of the completeness property is proved analogously. A re-sharing instance with at least t_D participating honest dealers will eventually agree on a set $\boldsymbol{L'}$ of t_D valid inner dealings to include in the dealings package. A notable difference is that it may take up to $f_\mathrm{D}+1$ attempts until a list $\boldsymbol{L'}$ of t_D valid inner dealings can be collected. Eventually, though, all honest dealers will compose a valid dealings package that all recipients can extract a valid share from.

7 Security of a Simple Hash to Curve

In principle, each DKG protocol instance needs a fresh common reference string $crs \leftarrow_\$ \mathbb{G}$. The most practical way to obtain it is to derive crs from a random

oracle $\mathsf{H}_{\text{crs}} : \{0,1\}^* \to \mathbb{G}$ on the public instance identifier *iid*, i.e., $crs \leftarrow \mathsf{H}_{\text{crs}}(iid, \texttt{inner})$ and $crs \leftarrow \mathsf{H}_{\text{crs}}(iid, \texttt{outer})$ for the inner and outer dealings in a DKG instance.

Standards for hashing into elliptic-curve groups [FHSS+23, BCI+10] are rather intricate because they mandate *constant-time* implementations to protect against side channels that leak a secret hash input. In our use case, the hash input (the identifier *iid*) is entirely public, so side-channel leakage does not pose a threat. Consequently, the constant-time machinery of the standard constructions is unnecessary overhead here.

Boneh, Lynn, and Shacham [BLS04] described the simpler "hash-and-check" algorithm that hashes to an elliptic curve based on a hash function that maps to the underlying prime field. More specifically, let $E/\mathbb{F}_q$ be an elliptic curve of prime order p, i.e., of cofactor 1, over a prime field $\mathbb{F}_q$ defined by $y^2 = f(x) \bmod q$, so that $\mathbb{G} = E/\mathbb{F}_q$ and every point on the curve lies in $\mathbb{G}$. Given a hash function $\mathsf{H}_q : \{0,1\}^* \to \mathbb{F}_q \times \{0,1\}$, the hash-to-curve algorithm $\mathsf{H}_{crs} : \{0,1\}^* \to \mathbb{G}$ works as follows:

$\mathsf{H}_{crs}(m)$:
  for $i = 0, \ldots, I-1$:
    $(x, b) \leftarrow \mathsf{H}_q(i \| m)$
    if $f(x)$ is a quadratic residue in $\mathbb{F}_q$ then:
      let y_0, y_1 be the two square roots of $f(x)$ in some full ordering
      return $(x, y_b) \in E/\mathbb{F}_q$
  return failure

The parameter I bounds the number of trials and is chosen so that the failure probability 2^{-I} is negligible; since each trial finds a quadratic residue with probability roughly $1/2$, the algorithm returns failure only with this negligible probability.

Boneh et al. proved that the above construction is safe to use in the BLS signature scheme if H_q is a random oracle. We prove a stronger property here, namely that it is also indifferentiable [MRH04] from a random oracle, which essentially means that if H_q behaves like a random oracle, then $\mathsf{H}_{\mathbb{G}}$ behaves like a random oracle too. We prove this property in the UC framework in the following lemma.

Theorem 3 *The hash function* H_{crs} *securely realizes* $\mathcal{F}_{\text{ro}}^{\mathbb{G}}$ *in the* $\mathcal{F}_{\text{ro}}^{\mathbb{F}_q}$*-hybrid model, where* H_q *is modeled as a random oracle.*

Proof sketch. We have to build a simulator $\mathcal{S}$ that acts as the adversary in the ideal world with $\mathcal{F}_{\text{ro}}^{\mathbb{G}}$, and presents any hybrid-world adversary $\mathcal{A}$ and environment $\mathcal{E}$ with a proper simulation in the $\mathcal{F}_{\text{ro}}^{\mathbb{F}_q}$-hybrid world.

When $\mathcal{A}$ queries $\mathcal{F}_{\text{ro}}^{\mathbb{F}_q}(sid, m)$, $\mathcal{S}$ proceeds as follows. If m can be parsed as $i \| m'$ with $0 \le i < I$, then it checks whether $T[0, m']$ has already been defined

in a table T that it keeps in memory. If not, then for $j = 0, 1, \ldots$, it chooses $(x_j, b_j) \leftarrow_{\$} \mathbb{F}_q \times \{0,1\}$ until $f(x_j)$ is a quadratic residue or $j = I$. If $f(x_j)$ is not a quadratic residue, then $\mathcal{S}$ sets $T[j, m'] \leftarrow (x_j, b_j)$. If it is, then $\mathcal{S}$ queries $(x, y) \leftarrow \mathcal{F}_{\text{ro}}^{\mathbb{G}}(sid, m')$, computes the ordered square roots y_0, y_1 of $f(x)$, sets b such that $y = y_b$, and sets $T[j, m'] \leftarrow (x, b)$. If now $T[i, m']$ is still not defined, it sets $T[i, m'] \leftarrow_{\$} \mathbb{F}_q \times \{0,1\}$. It then returns $T[i, m']$.

One can see that, by proceeding this way, the outputs of $\mathcal{F}_{\text{ro}}^{\mathbb{F}_q}$ are correctly distributed as random elements of $\mathbb{F}_q \times \{0,1\}$, and that if $\mathcal{F}_{\text{ro}}^{\mathbb{G}}(m') = (x, y_b)$, then $\mathcal{F}_{\text{ro}}^{\mathbb{F}_q}(i \| msg') = (x, b)$ for the smallest i where $f(x_i)$ is a quadratic residue.

8 OCR Integration

We implement the total-order broadcast channel with Chainlink's Offchain Reporting (OCR) 3.1 protocol [BCC+25, BCK+24, Cha26a]. More particularly, we integrate the DKG protocol suite into the OCR 3.1 protocol as a so-called *reporting plugin*. While doing so, we "piggy-back" on OCR 3.1 by re-using some of its internal signatures in the DKG protocol.

8.1 OCR 3.1 Protocol and Reporting Plugins

OCR 3.1 is a Byzantine fault-tolerant offchain protocol run by n_{D} oracle nodes (in our case, the dealers $\boldsymbol{\mathcal{D}}$) that jointly form a decentralized oracle network (DON). The OCR 3.1 protocol operates as a replicated state machine that processes observations by individual nodes and produces attested reports that can be consumed by on-chain and off-chain targets. It tolerates $f_{\text{D}} < n_{\text{D}}/3$ Byzantine oracles and operates under partial synchrony.

Each OCR 3.1 protocol instance is parameterized by a *reporting plugin* that defines observation, aggregation, state transition, and report filtering logic.

High-Level architecture. OCR 3.1 is a leader-driven consensus protocol that consists of five concurrent sub-protocols:

- The *atomic outcome generation* protocol executes in global *rounds* with a designated leader for each round and reaches consensus on actions associated to contiguously increasing *sequence numbers*. During each round, the leader collects signed *observations* from other nodes and, if conditions for proceeding are met, *aggregates* these into a *state transition*, and produces a vector of *reports*. These four operations are specified by the reporting plugin, which also maintains some state encoded as a key-value state.
- The *pacemaker protocol* assigns a leader for each epoch to ensure progress. Epoch changes can be triggered by a configurable leader tenure limit or occur when lack of progress is detected within a round.

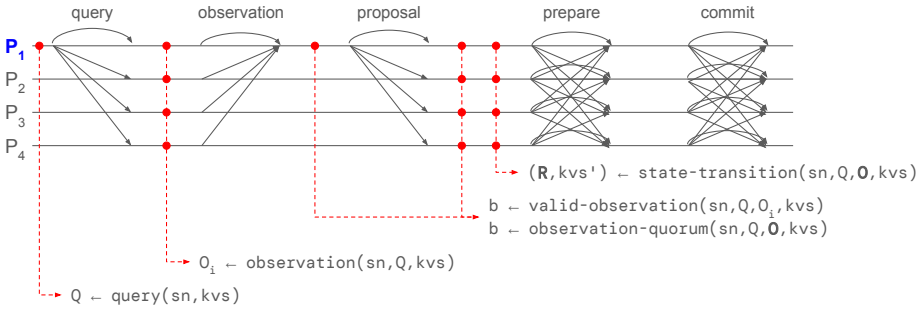


Figure 6: Overview of an OCR 3.1 protocol round and hooks for the reporting plugin interfaces, with party $\mathcal{P}_1$ acting as the leader.

- The *state synchronization* protocol enables lagging nodes to catch up with the latest agreed-on state of the replication plugin, so they can continue participating in the atomic outcome generation protocol.
- The *report attestation* protocol collects a configurable quorum of signatures (called an *attestation*) on the reports. A quorum of $f_{\mathrm{D}}+1$ signatures ensures that at least one honest oracle produced the report in the outcome generation protocol.
- The *transmission* protocol ensures a staggered submission of attested reports on chain.

Reporting plugin interface. Reporting plugins make OCR 3.1 configurable for a specific use case: it defines what to observe, how to validate and aggregate observations to mutate the replicated key-value state, and which reports to emit.

Figure 6 depicts the protocol flow of an OCR 3.1 round. The plugin's state is captured by the key-value state *kvs*. OCR 3.1 calls the following interfaces of the reporting plugin for each round:

$Q \leftarrow$ query(sn, kvs)**:** At the start of the round, the leader creates the query Q for this round by calling the query interface of the plugin with the sequence number sn and kvs as input. It then sends Q to all other participants.

$O_i \leftarrow$ observation(sn, Q, kvs)**:** Each participant $\mathcal{P}_i$ then calls the observation plugin interface to create its observation O_i and sends it to the leader.

$b \leftarrow$ validate-observation(sn, Q, O_i, kvs) **and** $q \leftarrow$ observation-quorum$(r, Q, \boldsymbol{O}, kvs)$**:** The leader verifies each incoming observation by calling the validate-observation plugin interface, which returns a boolean b, and includes at most one valid observation per participant in an observation vector $\boldsymbol{O}$. This process continues until $q =$ observation-quorum valid observations have been collected. The leader then sends $\boldsymbol{O}$ as a proposal to all participants, who each check that q entries in $\boldsymbol{O}$ satisfy validate-observation.

$(\boldsymbol{R}, kvs') \leftarrow$ state-transition$(sn, Q, \boldsymbol{O}, kvs)$**:** After verifying that validate-observation holds for observation-quorum observations in $\boldsymbol{O}$, each participant locally invokes the plugin's state-transition interface. This produces a (possibly empty) vector of reports $\boldsymbol{R}$ and tentatively updates the key-value state to kvs'.

When a participant commits the observations for the round, which occurs after the two voting phases with prepare and commit messages as shown in Figure 6, the updated key-value state kvs' is persisted.

In addition, and not shown in the figure, the report attestation protocol collects a number of signatures for each report in $\boldsymbol{R}$ to further process the reports.

Implementing total-order broadcast. One can use the plugin interfaces of OCR 3.1 to implement the total-order broadcast functionality $\mathcal{F}_{\text{tob}}$ from Figure 1 as follows.

Party $\mathcal{P}$ maintains a local buffer of messages, which stores messages that it has broadcast but that have not yet been delivered. Whenever $\mathcal{P}$ broadcasts a message m, it adds m to the buffer. At every invocation of its observation() interface, the plugin function then returns all messages in the buffer as the observation $O_{\mathcal{P}}$.

According to OCR 3.1, the leader aggregates sufficiently many observations in an observation vector $\boldsymbol{O}$ that is included in a proposal.

Whenever some $\boldsymbol{O}$ is committed, party $\mathcal{P}$ writes the messages contained in $\boldsymbol{O}$ to its key-value state, with increasing sequence numbers and in the order that they appeared in the leader's proposal. Furthermore, when a message m of $\mathcal{P}$ appears in a committed observation vector $\boldsymbol{O}$, then $\mathcal{P}$ removes m from its buffer, so that duplicate messages are filtered out.

The delivery of each message can be done by triggering a corresponding event whenever OCR 3.1 outputs a report. Alternatively, each party may look up the history of observations in its key-value state directly and deliver them to an application.

As an additional feature, not modeled by $\mathcal{F}_{\text{tob}}$, OCR 3.1 can emit attested (i.e., signed by $f+1$ parties) reports derived from the observation vector $\boldsymbol{O}$ and the key-value state. We leverage this to obtain, in-band and within the same round, a quorum of oracle signatures on the dealings package, thereby avoiding a separate OCR round devoted solely to signature collection.

8.2 The DKG Reporting Plugin

We now describe how the DKG protocol suite from Section 5 is embedded into an OCR 3.1 reporting plugin. We first do so for a fresh dealing; we'll explain the differences in a re-sharing immediately after.

Plugin overview. The dealers $\boldsymbol{\mathcal{D}}$ in a DKG protocol instance also form the oracle nodes in an OCR 3.1 DON. All dealers read their initial configuration from a smart contract on chain that instructs the nodes to start OCR with the DKG reporting plugin. The nodes will execute a single instance of the $\mathcal{DKG}$.Deal protocol and emit the resulting dealing package *pkg* as an attested report. The on-chain configuration specifies the public keys of all dealers $\boldsymbol{\mathcal{D}}$ and recipients $\boldsymbol{\mathcal{R}}$, together with the number of corrupt dealers f_{D} and the recipients' reconstruction threshold t_{R}. The instance identifier *iid* is uniquely determined by the config contract address and the config digest. The nodes obtain their secret keys from a local key store.

The DKG protocol is implemented as a state machine, where the current state is kept in the replicated key-value state *kvs*. The plugin interfaces behave differently in each phase. Each phase takes at least one OCR round, but could take multiple rounds if no progress was made in the previous round.

All nodes are initially in the `DEALING` state to perform the first phase of the $\mathcal{DKG}$.Deal protocol, where each dealer contributes an outer and encrypted inner dealing as its observation. The leader compiles a vector of $f_{\mathrm{D}} + 1$ such valid dealings that is written to the key-value state and proceeds to the `DECRYPTING` state.

In the `DECRYPTING` state, the nodes perform the second phase of the $\mathcal{DKG}$.Deal protocol, where all nodes contribute their decryption shares for the outer dealings as observations and the leader compiles a vector of $f_{\mathrm{D}} + 1$ of such observations. If successful, all nodes proceed to the `FINISHED` state by locally recovering the inner dealings as a report and handing it over to the report attestation protocol.

In the `FINISHED` state, nodes continuously pass the decrypted inner dealings to the report attestation protocol. This protocol invokes the reports interface to produce a report for transmission and collects $f_{\mathrm{D}} + 1$ signatures over it. Once a quorum is reached, the transmission protocol writes a key-value pair to the local database, where the key is given by the instance identifier *iid*, and the value contains the dealings package, which includes both the inner dealings and the collected signatures.

Detailed steps. More specifically, in the `DEALING` state, the dealers proceed as follows:

1. This epoch's leader calls the plugin's query interface that returns an empty query Q. The leader sends Q to all other dealers.
2. Each dealer $\mathcal{D}$, upon receiving Q, calls the observation interface of the DKG plugin. It then runs the first phase of the $\mathcal{DKG}$.Deal protocol to return its outer and encrypted inner dealing $(OD_{\mathcal{D}}, EID_{\mathcal{D}})$ as its observation $O_{\mathcal{D}}$. The OCR protocol signs this observation $O_{\mathcal{D}}$ and sends it back to the leader.

3. Upon receiving an observation $O_{\mathcal{D}} = (OD_{\mathcal{D}}, EID_{\mathcal{D}})$ from a dealer $\mathcal{D}$, the leader verifies $\mathcal{D}$'s signature and calls the validate-observation interface that verifies the outer dealing as

$$\mathcal{VESS}.\mathsf{VerifyDealing}(OD_{\mathcal{D}}, \boldsymbol{\mathcal{D}}, f_{\mathrm{D}} + 1, \boldsymbol{ek}_{\boldsymbol{\mathcal{D}}}, ad)$$

for $ad = (iid, \texttt{outer}, \mathcal{D}, attempt)$ and returns the result.

If it is valid, the leader adds the signed observation $O_{\mathcal{D}}$ to its list of valid observations $\boldsymbol{O}$. It then calls the observation-quorum interface, which returns 1 if $\boldsymbol{O}$ contains at least $f_{\mathrm{D}}+1$ observations. Once it does, the leader sends $\boldsymbol{O}$ to all dealers.

4. Upon receiving a vector of signed observations $\boldsymbol{O}$, each dealer verifies the signatures on the observations, calls the validate-observation interface described above on each individual observation, and calls the observation-quorum interface to check that $\boldsymbol{O}$ indeed contains at least $f_{\mathrm{D}}+1$ observations from different dealers. If so, it continues with the rest of the OCR round and eventually reaches consensus on the vector $\boldsymbol{O}$.

5. If the round finishes successfully with a set of observations $\boldsymbol{O}$, all dealers call the state-transition function that writes the first $f_{\mathrm{D}}+1$ observations in $\boldsymbol{O}$ as vector $\boldsymbol{L}$ and the new state `DECRYPTING` to the replicated key-value state, without creating any reports.

Once they're in the `DECRYPTING` state, the dealers in the next OCR round perform the following steps:

1. The leader calls the query interface that simply outputs an empty query Q again and sends it to all other dealers.

2. Each dealer $\mathcal{D}$ calls the observation interface that performs the second phase of the $\mathcal{DKG}.\mathsf{Deal}$ protocol by

   - looking up the vector $\boldsymbol{L} = \{(OD'_{\mathcal{D}}, EID'_{\mathcal{D}})\}$ from the key-value state,
   - computing decryption shares

   $$z_{\mathcal{D}',\mathcal{D}} \leftarrow \mathcal{VESS}.\mathsf{Decrypt}(dk_{\mathcal{D}}, OD_{\mathcal{D}'}, \mathcal{D}, ad)$$

   for $ad = (iid, \texttt{outer}, \mathcal{D}', attempt)$
   - returning the package of decryption shares $(z_{\mathcal{D}',\mathcal{D}})_{\mathcal{D}' \in \boldsymbol{L}}$ as its observation.

3. Upon receiving such an observation, the leader calls the validate-observation interface that verifies each decryption share by checking that

$$\mathcal{VESS}.\mathsf{VerifyShare}(z_{\mathcal{D}',\mathcal{D}}, OD_{\mathcal{D}'}, \mathcal{D}) = 1$$

for all dealers $\mathcal{D}$ appearing in $\boldsymbol{L}$. If it passes, the leader adds the observation to a list $\boldsymbol{O}$. The observation-quorum interface returns 1 if $\boldsymbol{O}$ contains at least $f_{\mathrm{D}} + 1$ observations, upon which the leader sends $\boldsymbol{O}$ to all dealers.

4. Upon receiving $\boldsymbol{O}$, each dealer verifies the observations using the same validate-observation and observation-quorum interfaces as described above.

5. If the round finishes successfully with a set of observations $\boldsymbol{O}$, the state-transition interface runs the last phase of the $\mathcal{DKG}$.Deal protocol by reconstructing the decryption keys $z'_{\mathcal{D}}$, recovering and verifying the inner dealings $ID'_{\mathcal{D}}$, and compiling the list of inner dealings $\boldsymbol{ID}$ as described in Section 5.

   It writes $\boldsymbol{ID}$ and the new state `FINISHED` to the replicated key-value state and creates a report containing $\boldsymbol{ID}$. This will trigger the report attestation protocol to collect a set of $f_{\mathrm{D}}+1$ signatures by different dealers on it, which thereby forms the dealings package pkg.

The plugin interfaces for the resharing protocol follow the $\mathcal{DKG}$.Reshare protocol in a similar way as they did for the $\mathcal{DKG}$.Deal protocol above.

One notable change is that the key-value state now also stores the attempt counter *attempt* and the list of banned dealers $\boldsymbol{\mathcal{B}}$ with the current state that are initially set to zero and the empty set, respectively.

In the validate-observation interface during the `DEALING` state, any observation originating from a dealer listed in $\boldsymbol{\mathcal{B}}$ is deemed invalid and is discarded.

If the state-transition interface in the `DECRYPTING` state fails to recover t_{D} valid inner dealings, it increments *attempt* in the key-value state, adds the dealers with invalid inner dealings to $\boldsymbol{\mathcal{B}}$, and resets the state to `DEALING`.

Blob dissemination. The OCR 3.1 protocol supports an optimization where the nodes can asynchronously disseminate arbitrary blobs of data. Specifically, a node can broadcast a blob to all other participants and wait for signatures from a Byzantine quorum, which attest to the full receipt of the blob. The combination of the blob's digest and the collected signatures forms a blob handle, which serves both as a proof of availability and as a reference for retrieving the blob content.

In the DKG case, dealers can leverage this mechanism to disseminate their dealings $(OD_{\mathcal{D}}, EID_{\mathcal{D}})$ as blobs, including only the corresponding blob handles in their observations. This approach significantly reduces the size of observations and, consequently, lowers the bandwidth requirements for the leader during the protocol execution.

By contrast, decryption share packages are not disseminated via blobs, as they are already smaller than blob handles themselves.

9 Evaluation

We evaluate the performance of our DKG protocol suite implemented as an OCR 3.1 reporting plugin in Go [Cha26b]. We measure artifact sizes, end-to-end latency, and network bandwidth across different committee sizes ($n_{\mathrm{D}} \in$

$\{4, 7, 10, \ldots, 31\}$), modes (fresh generation and resharing), and reconstruction thresholds ($t_R \in \{f_D + 1, 2f_D + 1\}$), where the corruption threshold is set to $f_D = \lfloor (n_D - 1)/3 \rfloor$ in all experiments, consistent with the OCR 3.1 Byzantine fault tolerance bound.

For the distributed experiments, we deployed up to 31 DigitalOcean droplets of type `s-4vcpu-8gb` (4 vCPUs, 8 GB RAM, 160 GB disk, Ubuntu 22.04 LTS), distributed evenly across 8 data centers in 7 regions: New York (2 data centers), Singapore, Amsterdam, Frankfurt, Toronto, San Francisco, and Bangalore. Each experiment runs the DKG protocol for over a hundred cycles to collect statistically meaningful measurements; we report the 95th percentile (p95) latency to capture tail performance.

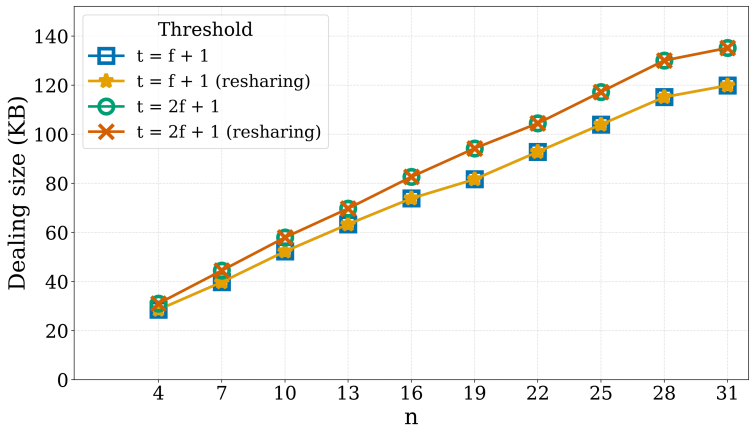
Artifacts and Core Computation Times. Figure 7 illustrates the sizes of the main DKG protocol artifacts: dealings and result packages. Dealing sizes grow linearly [4] in n_D (due to the per-recipient ciphertexts), from approximately 29 KB for $n_D = 4$ to 120–135 KB for $n_D = 31$. The result package, i.e., the aggregated output of the DKG protocol that each recipient needs to recover their share, grows quadratically, from about 29 KB to over 1.5 MB in the most expensive resharing configuration ($n_D = 31$, $t_R = 2f_D + 1$). Decryption shares are compact at under 1 KB in all configurations.

The computation time of the core operations of creating and verifying dealings for different committee sizes and thresholds is depicted in Figure 8. All of these were measured on the same DigitalOcean droplet used for distributed experiments. Creating a dealing takes 180–2000 ms depending on committee size, with verification taking roughly half that time. The operations to decrypt or verify a share remain within 10 ms for committees up to 16 and around 23 ms for the largest committee ($n_D = 31$), making them negligible relative to the network latency.

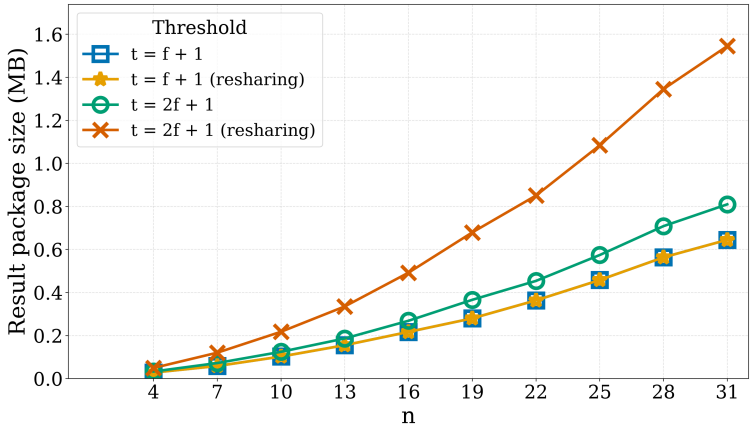
End-to-end latency. Figure 9 summarizes the end-to-end DKG latency, broken down into the latency of the dealing phase and of the decrypting phase. The end-to-end latency grows quadratically with the committee size, consistent with the $O(n_D^2)$ communication and computation complexity of the protocol: the DKG protocol finishes in under 2 s for $n_D = 4$, under 3 s for $n_D = 7$, roughly 6 s for $n_D = 16$, and about 22 s for $n_D = 31$ at threshold $f_D + 1$. Increasing the reconstruction threshold to $2f_D + 1$ has a moderate impact for smaller committees but increases latency more significantly for $n_D = 31$, where the latency rises to about 40 s.

In the dealing phase, fresh generation at both thresholds and resharing at threshold $f_D + 1$ exhibit very similar latencies. Resharing at threshold $2f_D + 1$ is slower, because more dealings must be collected.

[4] There is a slight deviation from strict linear growth at $n_D = 31$, where the dealing size grows more slowly and falls below the linear trend. This is because in our implementation, the repetition parameter N and subset size parameter M in VESS are not static; we tune them for each setting of n and t to optimize for bandwidth while maintaining 128-bit security.

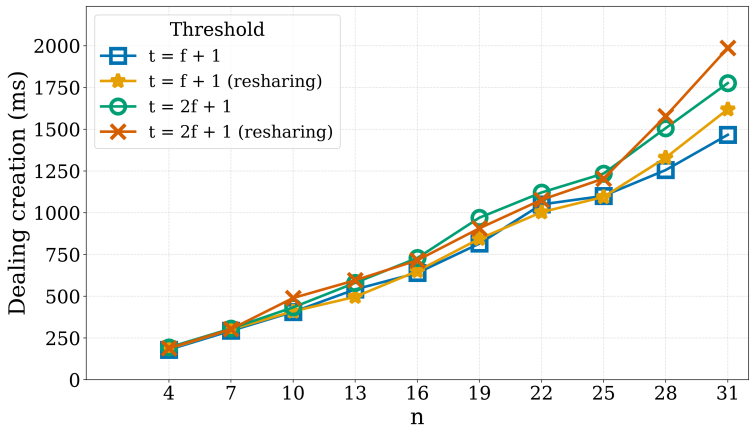


(a) Size of a single dealing produced by a dealer.

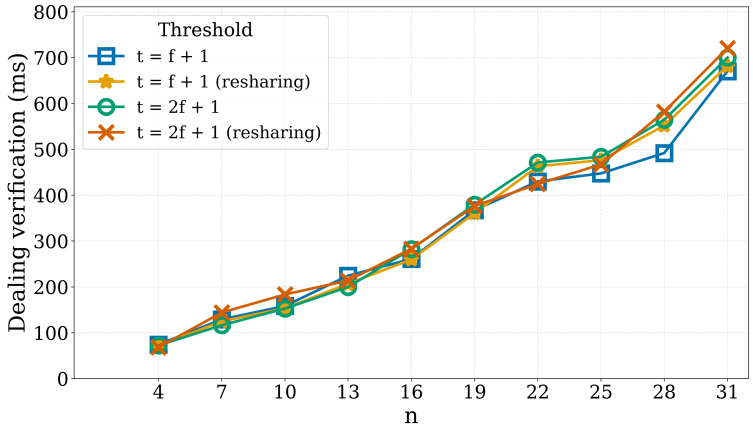


(b) Size of a result package.

Figure 7: Artifact sizes across different committee sizes and reconstruction thresholds.



(a) Time for a dealer to create a dealing.



(b) Time to verify a dealing.

Figure 8: Core computation times across different committee sizes and reconstruction thresholds.

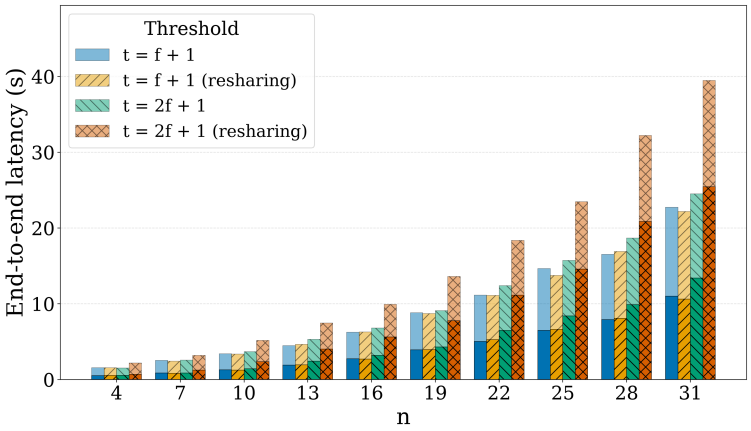


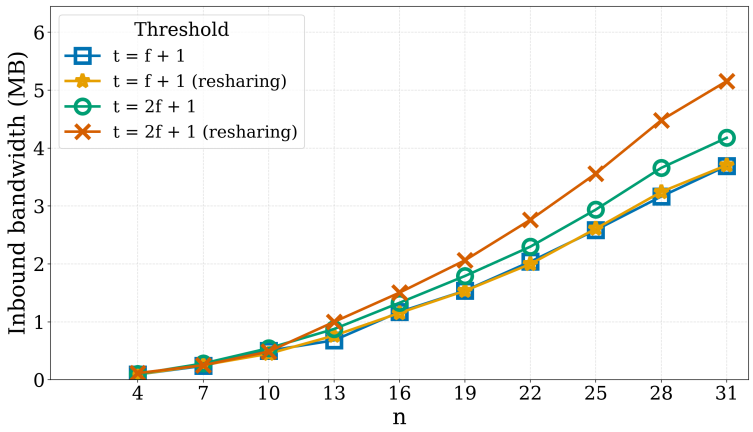
Figure 9: End-to-end latency broken down by phase: the dealing phase (lighter, upper bars) and the decrypting phase (darker, lower bars).

In the decrypting phase, fresh generation at threshold $2f_{\mathrm{D}} + 1$ becomes slower, owing to the higher threshold of the inner dealings that must be decrypted and verified. Resharing at threshold $2f_{\mathrm{D}} + 1$ exhibits an even higher latency, since resharing the master secret requires more inner dealings to be decrypted and verified. For the 31-node committee, the decrypting phase grows from approximately 11 s at threshold $f_{\mathrm{D}} + 1$ to 13.4 s for fresh generation and 25.5 s for resharing at threshold $2f_{\mathrm{D}} + 1$.

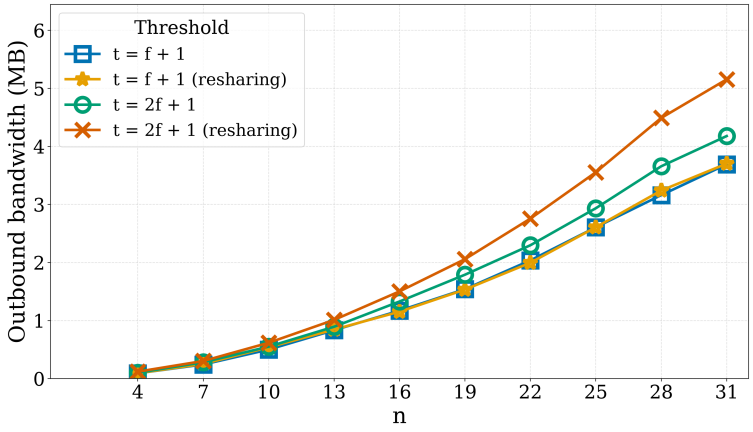
Network bandwidth. Figure 10 reports the average bytes sent and received per node per DKG protocol. As expected, network bandwidth grows quadratically with committee size: each node's outgoing traffic grows from approximately 117 KB for $n_{\mathrm{D}} = 4$ to roughly 1.4 MB for $n_{\mathrm{D}} = 16$ and 4.9 MB for $n_{\mathrm{D}} = 31$, reflecting the $O(n_{\mathrm{D}}^2)$ communication complexity of the protocol where each dealer broadcasts dealings containing ciphertexts for all recipients.

Discussion. The results demonstrate that our DKG protocol achieves practical performance for realistic committee sizes. For a 16-node committee (a typical size for Chainlink DON), a full DKG cycle at threshold $f_{\mathrm{D}} + 1$ completes in roughly 6 seconds and produces a result package of 222 KB. The bandwidth requirements of approximately 1.1 MB per node are well within the capacity of modern networks.

The protocol can thus be run periodically to refresh keys without noticeable service interruption.



(a) Inbound bandwidth per node.



(b) Outbound bandwidth per node.

Figure 10: Bandwidth usage across different committee sizes and reconstruction thresholds.

Acknowledgements

We would like to thank Dan Boneh and Victor Shoup for their very helpful feedback and discussions.

References

[ASW00] N. Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures. *IEEE J. Sel. Areas Commun.*, 18(4):593–610, 2000.

[BCC+25] Lorenz Breidenbach, Christian Cachin, Alex Coventry, Yan Ji, Kostis Karantias, Philipp Schindler, Chrysoula Stathakopoulou, and Alexandru Topliceanu. Chainlink offchain reporting protocol 3.0. `https://research.chain.link/ocr3.pdf`, February 2025.

[BCI+10] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 237–254. Springer, Berlin, Heidelberg, August 2010.

[BCK+24] Lorenz Breidenbach, Christian Cachin, Kostis Karantias, Philipp Schindler, and Chrysoula Stathakopoulou. OCR3.1 [draft]. Chainlink internal report, December 2024.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.

[BS23] Dan Boneh and Victor Shoup. *A Graduate Course in Cryptography, version 0.6*. 2023.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[Cha26a] Chainlink Labs. libocr. `https://github.com/smartcontractkit/libocr`, 2026.

[Cha26b] Chainlink Labs. smdkg: A Go implementation of the Chainlink DKG protocol. `https://github.com/smartcontractkit/smdkg`, 2026.

[Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th FOCS*, pages 427–437. IEEE Computer Society Press, October 1987.

[FHSS+23] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. Hashing to Elliptic Curves. RFC 9380, August 2023.

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

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[GS22a] Jens Groth and Victor Shoup. Design and analysis of a distributed ECDSA signing service. Cryptology ePrint Archive, Report 2022/506, 2022.

[GS22b] Jens Groth and Victor Shoup. On the security of ECDSA with additive key derivation and presignatures. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 365–396. Springer, Cham, May / June 2022.

[HJKY95] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 339–352. Springer, Berlin, Heidelberg, August 1995.

[Kat24] Jonathan Katz. Round-optimal, fully secure distributed key generation. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VII*, volume 14926 of *LNCS*, pages 285–316. Springer, Cham, August 2024.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Berlin, Heidelberg, March 2013.

[MRH04] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 21–39. Springer, Berlin, Heidelberg, February 2004.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Berlin, Heidelberg, August 1992.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[Sho25] Victor Shoup. Simple VESS. Cryptology ePrint Archive, Paper 2025/1175, 2025.

[SS24] Victor Shoup and Nigel P. Smart. Lightweight asynchronous verifiable secret sharing with optimal resilience. *Journal of Cryptology*, 37(3):27, July 2024.