CFB-DKG

Complaint-Free Distributed Key Generation using Blockchain and ZK Proofs

...

Vocdoni

(Draft v0.1 - February 2026)

Abstract

We present a blockchain-assisted distributed key generation (DKG) protocol that eliminates interactive complaint procedures using zero-knowledge proofs. Existing smart-contract DKGs detect invalid contributions only after submission, requiring dispute phases that introduce timing constraints and attack surfaces. In our protocol, each participant submits a single zk-SNARK proving correctness of their contribution: polynomial commitment consistency, Feldman verification equations, and correct share encryption. The smart contract rejects any invalid proof, simplifying the protocol to non-interactive phases delimited by block numbers.

The protocol relies on standard primitives: Shamir's secret sharing, Feldman commitments, hashed ElGamal encryption, and Chaum-Pedersen discrete-log equality proofs, with on-chain verification via zk-SNARKs on EVM-compatible chains. Circuit specifications are provided for key generation, threshold decryption, and optional secret key disclosure. We also discuss strategies to reduce the number of public inputs in ZK proofs to keep on-chain verification practical.

1 Introduction

Distributed Key Generation (DKG) enables a group of participants to jointly generate cryptographic keys without relying on a trusted dealer. This primitive underpins threshold signatures, randomness beacons, and privacy-preserving protocols. A compromised key generation phase can undermine all subsequent security guarantees, making the correctness and robustness of DKG a critical concern.

1.1 From DKG to ADKG

Classical DKG protocols, beginning with Pedersen's 1991 construction [15], assume synchronous communication: messages arrive within a known bounded delay. This assumption can fail in practice, as real networks exhibit unbounded delays and partitions, and can be subject to adversarial scheduling.

Asynchronous DKG (ADKG) removes the synchrony assumption: the protocol must terminate and produce correct output regardless of message scheduling, provided that fewer than one-third of participants are Byzantine. The first ADKG construction was proposed by Kokoris-Kogias et al. [14] in 2020. Since then, progress has been rapid. Communication complexity has dropped from $O(\kappa n^4)$ to $O(\kappa n^3)$ [4, 8, 9, 21], and recent breakthroughs have achieved nearly-quadratic $\tilde{O}(n^2)$ communication [3, 11]. Adaptive security is now achievable with minimal or silent setup [2, 11]. A recent systematization of knowledge by Bacho and Kavousi [6] provides a comprehensive analysis of the DKG landscape.

However, a significant gap remains between theory and practice. The new constructions rely on sophisticated primitives such as packed AVSS, multi-valued validated Byzantine agreement and asynchronous common subset protocols. These primitives have yet to see widespread implementation or sustained cryptanalytic scrutiny.

1.2 Blockchain-assisted DKG

On the practical side, several works have explored using blockchains as a coordination layer for DKG. Schindler et al. [16] introduced ETHDKG, implementing Joint-Feldman DKG on Ethereum. ETHDKG requires interactive dispute resolution: a party that receives an invalid share posts a complaint on-chain, and the dealer must respond within a timeout. Sober et al. [18] improved on this by using zk-SNARKs for dispute resolution, but the protocol still retains a complaint phase in which the ZK proof is generated reactively, in response to a dispute.

In the public bulletin board model, Applebaum and Pinkas [5] showed that per-party communication can be made independent of the number of DKG participants using LDPC codes, though their construction uses techniques that are far from standard practice. Zhang et al. [22] explored decentralized CP-ABE for single-round sharing with $O(n)$ reconstruction complexity.

The only major production deployment, Shutter Network [19], has operated on Gnosis Chain since October 2022, using Feldman VSS with interactive complaints. While Shutter demonstrates the viability of on-chain DKG, it has known limitations: $O(n^2)$ communication, a synchrony assumption, and interactive dispute resolution that increases gas costs and protocol duration.

There remains a gap for protocols that are practical and rely on well-known, battle-tested cryptographic primitives, while still achieving public verifiability, eliminating interactive complaint procedures, and being compatible with EVM-based smart contract verification.

1.3 Our contribution

We present a blockchain-assisted DKG protocol for EVM-compatible chains whose central design principle is the *complete elimination of interactive complaint procedures* through systematic use of zero-knowledge proofs.

In all prior smart-contract DKG protocols [16, 18, 19], correctness of a participant's contribution is verified after the fact: other participants detect invalid shares and file complaints, triggering an interactive dispute resolution phase. Even in the zk-SNARK-enhanced protocol of Sober et al. [18], zero-knowledge proofs are used reactively: a dealer generates a proof only when challenged. The complaint mechanism introduces an entire protocol phase with its own timing constraints, and opens a class of attacks that exploit the interactive structure: griefing (filing false complaints to waste gas), timing attacks (filing complaints near deadlines), and exclusion of honest participants whose dispute responses are delayed.

In our protocol, every participant proactively proves the correctness of their entire contribution in a single zk-SNARK submitted alongside the contribution data. The smart contract accepts a submission only if the proof verifies. There is no complaint phase because there is nothing to complain about: invalid contributions are rejected at submission time.

This has several consequences. First, the protocol structure simplifies: Each participant's actual DKG contribution obligation reduces to a single blockchain transaction, and phases are delimited by block numbers with no interactive back-and-forth. Second, the entire DKG transcript becomes publicly verifiable. Any observer can confirm correct execution by inspecting the on-chain record. Third, the attack surface associated with interactive dispute resolution is eliminated by construction, not by parameter tuning.

The protocol uses standard primitives: Shamir secret sharing, Feldman-style polynomial commitments [10], hashed ElGamal encryption, Chaum-Pedersen discrete-log equality proofs, and zk-SNARKs. We also provide an overview of on-chain verification costs and strategies for reducing the number of public inputs to ZK proofs, including binding random linear combinations (BRLC). The protocol supports threshold decryption and optional secret key disclosure as post-DKG operations, with corresponding ZK proofs for each phase.

1.4 Asynchrony model

Our protocol uses the blockchain as a public bulletin board, inheriting its ordering, availability, and persistence guarantees. Phases are delimited by block numbers, and participants have a known window to submit contributions. A participant who fails to submit is simply excluded. The absence of a complaint phase means there is no risk of excluding honest participants due to interactive dispute deadlines. While this is not fully asynchronous in the theoretical sense, as the protocol may inherit the underlying chain's partial synchrony, it matches the operational model of blockchain-native applications and avoids the complexity of asynchronous agreement primitives.

1.5 Outline

Section 2 provides background on secret sharing, verifiable secret sharing, and distributed key generation. Section 3 discusses node selection strategies. Section 4 presents the DKG protocol and its use for threshold decryption and secret key disclosure. Section 5 gives the implementation outline for EVM chains, including ZK circuit specifications. Section 6 analyzes public input reduction strategies and gas costs.

2 Background

2.1 Secret sharing

Secret sharing refers to methods for distributing a secret among a group of participants, each of whom is given a share of the secret. The secret can be reconstructed only when predetermined groups of participants combine their shares.

The following case is of particular importance. There are n participants, and we require that any set of t or more shares can be used to reconstruct the secret, while any set of $t-1$ (or fewer) shares reveals nothing about the secret. This is called a (t, n) *threshold scheme*.

The fundamental construction is due to Shamir [17]. A trusted dealer holds a secret $\sigma \in \mathbb{F}_q$ and generates a random polynomial $f(x) \in \mathbb{F}_q[X]$ of degree $t-1$ such that $f(0) = \sigma$. The shares $s_i = f(i)$ for $i \in [n]$ are then privately distributed to n parties. Any set of t parties can recover the secret by combining their shares using Lagrange interpolation. However, $t-1$ (or fewer) parties can learn nothing about σ. Shamir secret sharing is used, in some form, by almost all practical secret sharing and DKG protocols.

2.2 Verifiable secret sharing (VSS)

With Shamir secret sharing, a faulty dealer could distribute spurious or random shares to participants. In that case, different subsets of t participants could reconstruct different secrets, rendering the protocol useless. Verifiable secret sharing (VSS) overcomes this limitation by allowing participants to verify that their shares are consistent.

Feldman [10] proposed the first non-interactive VSS protocol. The dealer publishes commitments to the polynomial coefficients:

$$C(k) := a(k)\, G \quad \text{for } k \in \{0, \ldots, t-1\}$$

where G is a generator of a cyclic group $\mathcal{G}$ of prime order q in which the discrete logarithm problem is hard. Any participant j holding share $s_j = f(j)$ can then verify its correctness by checking the *Feldman verification equation*:

$$s_j\, G \stackrel{?}{=} \sum_{k=0}^{t-1} j^k\, C(k) \tag{1}$$

This equation is central to our protocol: proving that it holds for all participants is the main task of the ZK proof in the key generation phase.

Note that Feldman's scheme reveals $C(0) = \sigma G$, the public key corresponding to the secret. For our application this is acceptable and indeed required.

2.3 Distributed Key Generation (DKG)

A DKG protocol allows a group of n participants to jointly generate a shared public key and the corresponding secret shares of a private key, without any trusted dealer. The participants do not know the private key, but any subset of t or more participants is able to reconstruct it or to collaboratively perform cryptographic operations such as signing or decrypting messages.

The first DKG protocol was proposed by Pedersen [15]. It consists of the parallel invocation of n Feldman VSS instances: every participant generates a random polynomial of degree $t-1$, publishes commitments to its coefficients, and privately sends shares to all other participants. The shared public key can then be computed from the published commitments to the constant terms.

Pedersen's protocol assumes synchronous communication and uses an interactive complaint mechanism to handle faulty dealers. If a participant receives an invalid share, it broadcasts a complaint; the accused dealer must respond within a timeout or be disqualified. This interactive dispute resolution introduces timing assumptions and is the source of several practical complications, as discussed in the introduction.

3 Node Selection

The security of any DKG protocol ultimately depends on the assumption that fewer than some threshold of participants are malicious. The cryptographic protocol enforces this guarantee given an honest majority, but it cannot create one. Node selection, i.e. how participants are chosen and incentivised, is a deployment concern that sits outside the protocol specification but is essential to its security in practice.

While the specific node selection strategy will depend on the application, collusion and coercion resistance will always be critical challenges. We outline several complementary approaches that can be combined to strengthen both collusion and coercion resistance.

Identity/Vetting Node operators must undergo identity verification to prevent Sybil attacks.

Reputation Choose well-known entities with significant reputations at stake, creating strong disincentives for malicious behavior.

Organization Size Large organizations are generally considered to be less coercion-resistant than smaller organizations or individuals.

Diversity Nodes should be diverse both geographically and sector-wise (e.g., academic institutions, NGOs, private companies, independent operators). This

provides resilience against regional internet outages, political or legal pressure, and reduces the likelihood of collusion.

Randomness Nodes participating in a DKG round should be selected using verifiable random functions (VRFs) or blockchain-based randomness beacons to prevent adversaries from predicting or influencing selection.

Economic stake Requiring nodes to lock collateral makes collusion economically costly and enables penalties for misbehavior through slashing mechanisms.

If we want a more diverse node ecosystem, we could define a two-tier structure: a core group of highly trusted nodes from established organizations, and a second tier of vetted ordinary nodes. For any DKG execution, we could require that a minimum fraction (e.g., half) of participants come from the highly trusted tier. This will provide a baseline security guarantee while maintaining openness.

4 Our DKG and decryption protocol

This section describes the distributed key generation protocol and its subsequent usage for threshold decryption and (optionally) secret key disclosure. The main output of the DKG is a public key PK whose corresponding secret key is not known to anyone.

Diagrams summarizing these protocols can be found in Appendix 1 (DKG process), Appendix 2 (threshold decryption process) and Appendix 3 (secret key disclosure process).

4.1 Setting

Let $\mathcal{G}$ be a cyclic group of prime order q, written additively, with generator $G \in \mathcal{G}$. Scalars are elements of $\mathbb{F}_q$.

We assume a smart-contract-enabled blockchain on which group operations in $\mathcal{G}$ (point addition and scalar multiplication) can be efficiently executed by smart contracts. Additionally, we assume that other operations, including the computation of hash function values, can also be performed efficiently.

We further assume a set of eligible nodes is known, and for each eligible node i there exists a public key pub_i for an encryption scheme used to encrypt shares. We denote this encryption function by $\mathsf{Enc}_{\mathtt{share}}$. In the following, we assume that this encryption function is Hashed ElGamal, which is described in Appendix 4.

Suitable smart contracts have been deployed, and the public keys of eligible nodes have been recorded.

4.2 Key Generation Process

4.2.1 Phase 1: Initiation

When initiating a DKG process, the initiator submits the protocol parameters (n, t) as well as policy parameters. Such policy parameters could include:

- Minimum participation criteria;
- Duration of phases 2 and 4;
- Minimum number of valid contributions in phase 4;
- Whether the secret key should remain secret "forever", or whether it will need to be disclosed at some stage - either at a future time or block number, or via a request of the initiator.

The smart contract should ensure that parameters are within reasonable limits. For every DKG process, a unique pseudo-random round identifier `rid` is also generated. This identifier can be used as a domain separator.

4.2.2 Phase 2: Show of Hands

This phase lasts for a predetermined number of blocks.

Before the DKG starts, the protocol determines which nodes are ready to participate. Eligible nodes monitor the blockchain and record their readiness to participate in round `rid`. This prevents the protocol from stalling due to offline nodes.

Optionally, these candidate nodes could be required to deposit collateral.

4.2.3 Phase 3: Determine participating nodes (or abort)

At the beginning of this phase we know the number and composition of the set of candidate nodes.

If some predetermined policy criteria are not met, the process aborts. This should happen if there are not enough candidate nodes, i.e., fewer than n. Another reason for aborting could be that the number of highly trusted nodes among the candidate nodes is not sufficient, for example, less than $\lceil n/2 \rceil$. Aborting is expected to be rare, as there should be some easy off-chain way to determine how many eligible nodes are available at a given time before initiating a DKG process.

If the criteria are met, then a set of n selected nodes is determined. Some randomness available on the blockchain should be used to make this selection unpredictable.

On some blockchains, the transaction that determines participating nodes can be triggered automatically, for example, one block after the end of Phase 1. However, on most blockchains, the selection of participating nodes requires submitting a transaction.

4.2.4 Phase 4: Main DKG process

This phase lasts for a predetermined number of blocks.

Each participant $i \in [n]$ generates a random polynomial in $\mathbb{F}_q[X]$ of degree $t-1$:

$$f_i(x) := \sum_{k=0}^{t-1} a_{i,k}\, x^k$$

where $a_i(k)$ are the coefficients randomly chosen from $\mathbb{F}_q$. The participant also samples a set of random scalars to be used for encrypting messages for other participants:

$$\{r_{i,j}\} \quad \text{for } j \neq i$$

The participant then publishes the following data on the blockchain:

(1) Public commitments of the polynomial coefficients

$$C_i(k) := a_{i,k}\, G \qquad \text{for } k \in \{0, \ldots, t-1\}$$

(2) Encrypted shares for each of the other participants. The share sent by participant i to participant j is $s_i(j) := f_i(j)$. Instead of sending the share via a private channel, an encrypted version of the share is published on-chain:

$$\begin{aligned} E_i(j) &:= \mathsf{Enc}_{\mathtt{share}}[\mathsf{pub}_j, s_i(j)] \quad \text{for } j \neq i \\ &:= (R_i(j), \sigma_i(j)) \end{aligned}$$

(3) ZK proof π_i for the following two statements:

(a) for all $j \neq i$, the "Feldman verification equation" holds:

$$s_i(j)\, G \stackrel{?}{=} \sum_{k=0}^{t-1} j^k\, C_i(k)$$

This proves that the shares are consistent with the committed polynomial coefficients.

(b) For all $j \neq i$, $E_i(j)$ is a correct encryption of $s_i(j)$.

The smart contract will only accept a submission if π_i is correct. If the proof is not correct, the transaction fails and no data is written to the blockchain.

Commitments to polynomial coefficients must be stored on-chain because they will be used in the next stage. Encrypted shares and ZK proofs do not need to be stored as long as they remain accessible as calldata.

4.2.5 Phase 5: Compute public key and shares of the secret

At the beginning of Phase 5, the number of successful contributions is known. Let $\mathcal{I} \subseteq [n]$ be the set of indices of these nodes. The process aborts if an insufficient number of contributions have been submitted. The initiator of the process may require the participation of all selected nodes or of some minimum number of nodes.

Let's assume that a sufficient number of contributions were submitted. The shared secret polynomial $F(x)$ is then:

$$F(x) := \sum_{\ell \in \mathcal{I}} f_\ell(x) \tag{2}$$

We have:

$$F(x) = \sum_{\ell \in \mathcal{I}} a_{\ell,0} + x \sum_{\ell \in \mathcal{I}} a_{\ell,1} + x^2 \sum_{\ell \in \mathcal{I}} a_{\ell,2} + \cdots + x^{t-1} \sum_{\ell \in \mathcal{I}} a_{\ell,t-1}$$

And therefore:

$$\begin{aligned} F(x)\, G &= \sum_{\ell \in \mathcal{I}} a_{\ell,0} G + x \sum_{\ell \in \mathcal{I}} a_{\ell,1} G + x^2 \sum_{\ell \in \mathcal{I}} a_{\ell,2} G + \cdots + x^{t-1} \sum_{\ell \in \mathcal{I}} a_{\ell,t-1} G \\ &= \sum_{\ell \in \mathcal{I}} C_\ell(0) + x \sum_{\ell \in \mathcal{I}} C_\ell(1) + x^2 \sum_{\ell \in \mathcal{I}} C_\ell(2) + \cdots + x^{t-1} \sum_{\ell \in \mathcal{I}} C_\ell(t-1) \end{aligned}$$

Define aggregated commitments as follows:

$$\overline{C}(k) := \sum_{\ell \in \mathcal{I}} C_\ell(k)$$

Then $F(x)\, G$ can be written as:

$$F(x)\, G = \sum_{k=0}^{t-1} x^k \overline{C}(k) \tag{3}$$

Privately compute shares of the secret key

Each of the participating nodes can compute their share of the secret as follows:

$$d_i := \sum_{\ell \in \mathcal{I}} s_\ell(i)$$

Recall that $s_\ell(i) = f_\ell(i)$, so that $d_i = F(i)$. Note that this computation is performed completely off-chain, and that these shares must be kept secret.

Publish the public key

$F(0)$ is the secret key and $F(0)\, G$ is the public key, which can be computed by the smart contract as follows:

$$\mathsf{PK} = \overline{C}(0) \tag{4}$$

Publish commitments to secret shares

It will also be useful to publish the values of $D_i := d_i\, G = F(i)\, G$. These are essentially commitments to the secrets d_i, but they can be computed publicly. Using Equation (3) we have:

$$D_i = \sum_{k=0}^{t-1} i^k \overline{C}(k)$$

These commitments $\{D_i\}_{i \in \mathcal{I}}$ can be computed by the smart contract.

4.3 Threshold Decryption

An ElGamal ciphertext of message M under public key PK has the form:

$$\mathsf{Enc}_{\mathtt{msg}}(M) = (C_1, C_2) = (rG,\ M + r\mathsf{PK}),$$

and decryption with secret key sk satisfying $\mathsf{PK} = \mathsf{sk}\,G$ yields:

$$M = C_2 - \mathsf{sk}\,C_1.$$

Here $\mathsf{sk} = F(0)$ is not known to anyone. The goal is to compute $\Delta := \mathsf{sk}\,C_1$ without disclosing sk.

4.3.1 Phase 1: Publication of Ciphertext

A ciphertext (C_1, C_2) is published together with the round identifier `rid`. The nodes that participated in the relevant DKG process monitor the blockchain and know when such ciphertexts are published.

4.3.2 Phase 2: Publication of Partial Decryptions

Participating nodes publish a partial decryption of C_1:

$$\delta_i := d_i C_1$$

together with a proof of discrete log equality:

$$(A_i, B_i, z_i)$$

Details of the Chaum-Pedersen proof

We assume that commitments D_i to the secret shares d_i have been published. Node i then needs to prove that

$$(Y_1, Y_2) := (D_i, \delta_i)$$

have the same discrete log with respect to G and C_1. Node i samples a random r and computes:

$$\begin{aligned} A_i &= r\,G \\ B_i &= r\,C_1 \\ e &= \mathsf{Hash}(\mathtt{rid} \,\|\, Y_1 \,\|\, Y_2 \,\|\, A_i \,\|\, B_i) \\ z_i &= r + e\,d_i \end{aligned}$$

The proof is the tuple (A_i, B_i, z_i), and its verification requires:

$$\begin{aligned} e' &:= \mathsf{Hash}(\mathtt{rid} \,\|\, Y_1 \,\|\, Y_2 \,\|\, A_i \,\|\, B_i) \\ z\,G &\stackrel{?}{=} A_i + e'\,Y_1 \\ z\,C_1 &\stackrel{?}{=} B_i + e'\,Y_2 \end{aligned}$$

This verification can be done directly by the smart contract.

4.3.3 Phase 3: Decryption

Once t or more partial decryptions have been published, the smart contract can be called to recover the message. Let $\mathcal{Q} = \{x_1, \ldots, x_t\} \subseteq \mathcal{I}$ be a qualifying set of nodes that have published partial decryptions $\{\delta_{x_k}\}_{k\in[t]}$ and define $\mathcal{Q}_k$ as follows:

$$\mathcal{Q}_k = \{x_1, .., x_t\} \setminus \{x_k\}$$

We know that:

$$\mathsf{sk} = F(0) = \sum_{k\in[t]} \lambda_k \, d_{x_k} \quad \text{where} \quad \lambda_k = \prod_{u\in\mathcal{Q}_k} \frac{u}{u - x_k}$$

We therefore have:

$$\Delta := \mathsf{sk}\, C_1 = \sum_{k\in[t]} \lambda_k d_{x_k} C_1 = \sum_{k\in[t]} \lambda_k \delta_{x_k}$$

The decrypted message is then:

$$M = C_2 - \Delta$$

These computations can also be done directly by the smart contract.

4.4 Secret Key Disclosure

In some use cases, the secret key needs to be published. The recovery of the secret key can either be triggered by an external event monitored by the nodes or after a predetermined block number.

Every participant knows the value of $F(x)$ at a single point. Because F is of degree $t - 1$, any subset of t participants is therefore able to reconstruct this polynomial.

4.4.1 Phase 1: Request secret key disclosure

The request to disclose the secret key can be sent by the organizer or triggered automatically, assuming this is allowed by the policy parameters.

4.4.2 Phase 2: Publication of secret key shares

Every participating node $i \in \mathcal{I}$ publishes their share of the secret d_i. The smart contract makes sure that the submitted value is correct by verifying:

$$d_i \, G \stackrel{?}{=} D_i$$

4.4.3 Phase 3: Computation of the secret key

Once t or more values have been submitted, anyone can compute the secret key $F(0)$. The smart contract should also be called to reconstruct the private key $F(0)$ in order to have a public record. Let $Q = \{x_1, \ldots, x_t\} \subseteq \mathcal{I}$ represent a qualifying set of nodes that have published their shares of the secret. We have, by Lagrange interpolation, and with λ_k defined as above:

$$F(0) = \sum_{k \in [t]} F(x_k)\lambda_k$$

The secret key can therefore be computed by the smart contract as:

$$\mathsf{sk} = \sum_{k \in [t]} d_{x_k}\lambda_k$$

5 Implementation outline for EVM chains

In Section 4, we assumed that smart contracts can perform group operations in $\mathcal{G}$. On many smart-contract platforms (e.g. EVM chains), this is not the case: the chain cannot efficiently add points or multiply by scalars in $\mathcal{G}$. In this case, it makes sense to move all computations and checks involving group operations in $\mathcal{G}$ into ZK proofs. The smart contract only performs: (i) bookkeeping, (ii) verification of SNARK proofs over a curve supported by the execution environment (on EVM chains, this is BN254 or BLS12-381), and optionally (iii) hashing or other operations to compute commitments if necessary.

5.1 Data model and circuit specifications

The smart contract treats group elements in $\mathcal{G}$ as opaque encodings, i.e., byte strings. The meaning of these encodings is enforced only through ZK proofs.

We assume that the set of eligible nodes, along with their public encryption keys, is stored on-chain. We define the vector of these public keys as:

$$\mathbf{pub} \triangleq \{\mathsf{pub}_j\}_{j \in [n]}$$

We further define the following vectors for each $i \in [n]$:

$$\begin{aligned}
\mathbf{a}_i &\triangleq \{a_{i,k}\}_{k \in \{0,\ldots,t-1\}} \\
\mathbf{C}_i &\triangleq \{C_i(k)\}_{k \in \{0,\ldots,t-1\}} \\
\mathbf{s}_i &\triangleq \{s_i(j)\}_{j \in [n] \setminus \{i\}} \\
\mathbf{r}_i &\triangleq \{r_{i,j}\}_{j \in [n] \setminus \{i\}} \\
\mathbf{E}_i &\triangleq \{(R_i(j), \sigma_i(j))\}_{j \in [n] \setminus \{i\}}
\end{aligned}$$

The vector of commitments to the values d_j will be denoted:

$$\mathbf{D} \triangleq \{D_j\}_{j \in \mathcal{I}}$$

Note also that for every interaction with the blockchain, nodes will need to identify themselves. A node can be identified by its blockchain address, in which case simply submitting a transaction from that address identifies the node. A more flexible alternative is to identify nodes by a signing public key, in which case a transaction can be submitted from any address, but must be accompanied by a valid signature.

5.2 ZK Proofs - Distributed key generation

DKG-1 After a new DKG process is initiated (phase 1), its parameters are stored on-chain. The three parameters that are relevant for zk-proofs are:

$$\mathbf{params} \triangleq (\mathsf{rid}, n, t)$$

DKG-2 During phase 2, the nodes send readiness signals to the blockchain. At the end of this phase, the list of candidate nodes is known.

DKG-3 In phase 3, the organizer triggers the node selection process. As a result, the ordered list **pub** of n selected nodes is published.

5.2.1 ZK Proof - DKG phase 4

During this phase, every selected node i samples a random vector $\mathbf{a}_i$ and $\mathbf{r}_i$ and computes the vectors $\mathbf{C}_i$, $\mathbf{s}_i$ and $\mathbf{E}_i$. The computation makes use of **params** and **pub**, which are public information.

The node then computes the proof π_i with private inputs $\mathbf{a}_i$ and $\mathbf{r}_i$ and public inputs **params**, **pub**, $\mathbf{C}_i$ and $\mathbf{E}_i$. Note that the number of public inputs can be reduced by computing vector commitments. This will be discussed below, in Section 6.

The proof π_i together with $\mathbf{C}_i$ and $\mathbf{E}_i$ is sent to the smart contract, which stores $\mathbf{C}_i$ on-chain if the proof is correct. The following describes the circuit for which the proof is generated. Note that the public inputs that are sent as calldata are marked with an asterisk.

Private inputs

$\mathbf{a}_i$	(polynomial coefficients)
$\mathbf{r}_i$	(encryption randomness)

Public inputs

params	(protocol parameters)
pub	(public keys)
$* \mathbf{C}_i$	(polynomial commitments)
$* \mathbf{E}_i$	(encrypted shares)

In-circuit computations/validations

Compute shares for all $j \neq i$:

$$s_i(j) := \sum_{k=0}^{t-1} a_i(k)\, j^k \pmod q$$

Verify commitments to polynomial coefficients for all $k \in \{0, \ldots, t-1\}$:

$$C_i(k) \stackrel{?}{=} a_i(k)\, G$$

Verify Feldman equations for all $j \neq i$:

$$s_i(j)\, G \stackrel{?}{=} \sum_{k=0}^{t-1} j^k\, C_i(k)$$

Verify knowledge of encryption randomness for all $j \neq i$:

$$R_i(j) \stackrel{?}{=} r_{i,j}\, G$$

Verify correct encryption for all $j \neq i$:

$$\sigma_i(j) \stackrel{?}{=} s_i(j) + \mathsf{Hash}[r_{i,j}\, \mathsf{pub}_j] \pmod q$$

5.2.2 ZK Proof - DKG phase 5

Once the set $\mathcal{I}$ of participating nodes is known, the public key PK as well as commitments to private shares can be computed and published by anyone. However, to have a public reference, it is important to publish it on-chain as well.

The corresponding zk-proof can be described as follows.

Private inputs

-

Public inputs

params	(protocol parameters)
$\mathcal{I} \subseteq [n]$	(indices of participating nodes)
$\{\mathbf{C}_i\}_{i \in \mathcal{I}}$	(all polynomial commitments)
$* \mathbf{D}$	(commitments to private shares)
$* \mathsf{PK}$	(public key)

In-circuit computations/validations

Compute aggregated commitments for $k \in \{0, \ldots, t-1\}$:

$$\overline{C}(k) := \sum_{\ell \in \mathcal{I}} C_\ell(k)$$

Verify Public Key:

$$\mathsf{PK} \stackrel{?}{=} \overline{C}(0)$$

Verify commitments to private shares, for $i \in \mathcal{I}$:

$$D_i \stackrel{?}{=} \sum_{k=0}^{t-1} i^k \overline{C}(k)$$

5.3 ZK Proofs - Threshold decryption

5.3.1 ZK Proof - Decryption phase 2

In this phase, every node i provides a partial decryption δ_i of the ciphertext.

Private inputs

d_i	(private share)

Public inputs

params	(protocol parameters)
C_1	(first part of ciphertext)
D_i	(commitment to secret share)
$* \, \delta_i$	(partial decryption of C_1)
$* \, (A_i, B_i, z_i)$	(proof of DLEQ)

In-circuit computations/validations

The circuit computes:

$$e' := \mathsf{Hash}(\mathtt{rid} \,\|\, D_i \,\|\, \delta_i \,\|\, A_i \,\|\, B_i)$$

The two constraints are then:

$$z_i \, G \stackrel{?}{=} A_i + e' D_i$$
$$z_i \, C_1 \stackrel{?}{=} B_i + e' \delta_i$$

5.3.2 ZK Proof - Decryption phase 3

Once at least t partial decryptions have been published in phase 2, anyone can decrypt the ciphertext.

PRIVATE INPUTS

-

PUBLIC INPUTS

(C_1, C_2)	(ciphertext)
$\{\delta_i\}$	(partial decryptions)
$* \mathcal{Q}$	(qualifying set)
$* M$	(decrypted message)

IN-CIRCUIT COMPUTATIONS/VALIDATIONS

Compute Lagrange coefficients for $k \in [t]$:

$$\lambda_k := \prod_{u \in \mathcal{Q}_k} \frac{u}{u - x_k}$$

Verify decryption:

$$M \stackrel{?}{=} C_2 - \sum_{k \in [t]} \lambda_k \, \delta_{x_k} \pmod q$$

5.4 ZK Proof - Secret key disclosure

5.4.1 ZK Proof - Secret key disclosure phase 2

PRIVATE INPUTS

-

PUBLIC INPUTS

$* d_i$	(secret key share)
D_i	(commitment to secret key)

IN-CIRCUIT COMPUTATIONS/VALIDATIONS

Verify that the secret key share is correct:

$$d_i \, G \stackrel{?}{=} D_i$$

5.4.2 ZK Proof - Secret key disclosure phase 3

PRIVATE INPUTS

-

PUBLIC INPUTS

$* \mathsf{sk}$	(secret key)
$* \mathcal{Q}$	(qualifying set)
$\{d_j\}_{j \in \mathcal{Q}}$	(shares of secret key)

IN-CIRCUIT COMPUTATIONS/VALIDATIONS

Compute Lagrange coefficients for $k \in [t]$:

$$\lambda_k := \prod_{u \in \mathcal{Q}_k} \frac{u}{u - x_k} \quad \text{where} \quad \mathcal{Q}_k = \mathcal{Q} \setminus \{x_k\}$$

Verify decryption:

$$\mathsf{sk} \stackrel{?}{=} \sum_{k \in [t]} d_{x_k} \lambda_k \pmod q$$

6 Public Input Reduction Strategies

In the preceding section, to make the protocol explicit, we took the naïve approach of sending all public inputs directly to the verifier. Many of these public inputs are actually elliptic curve points. For a BN254 verifier, these will be Baby Jubjub points, which are represented as two 32-byte scalars. If we were using a BLS12-381 verifier, the curve points would be on Jubjub and require four scalars. In addition, every encrypted share requires one curve point and one scalar.

From now on, let's assume that the circuit is over BN254. The two critical phases are DKG phases 4 and 5, as they require the most public inputs. Assuming that $t \approx n/2$, an estimate of the number of inputs counted as 32-byte scalars is given in Table 1.

Table 1: Estimated Number of Public Inputs (32-byte scalars)

Phase	**Inputs**	**n=10**	**n=20**	**n=40**
DKG 4 (per participant)	$\approx 6n$	≈ 120	≈ 240	≈ 480
DKG 5 (organizer)	$\approx n^2/2$	≈ 50	≈ 200	≈ 800

Now let's look at two widely used verifiers: Groth16 [13] and FFLONK [12]. Their basic characteristics in terms of cost and public inputs are shown in Table 2. The data is taken from [20].

This shows that FFLONK, which is limited to about 38 public input scalars, cannot be used even if there are only $n = 10$ participants in the DKG. Groth16 could be

Table 2: BN254 Verifier Comparison

Property	**Groth16**	**FFLONK**
Fixed gas cost	207,000	200,000
Per-input gas cost	6,650	400
+ calldata (sparse)	∼140	
+ calldata (dense)	∼512	
+ storage access (cold)	2,100	
Max. inputs	∼700	∼38

used for up to $n \approx 35$ but at the cost of an overall very high gas expenditure: approximately 3.5m gas per participant in DKG phase 4, and 5m gas for the organizer in phase 5, for an overall total of almost 130m gas.

Methods for reducing the number of public inputs must therefore be analyzed. If a commitment of a vector of public inputs can be efficiently computed on-chain and efficiently verified in-circuit, then that vector can be replaced by the commitment. As a result, the original public input can be treated as private input, thus reducing the number of public inputs.

6.1 Hash-to-Single-Signal

A first approach is to reduce l public inputs $\{x_1, \ldots, x_l\}$ to a single signal using a hash function:

$$h = \mathsf{Hash}(x_1, x_2, \ldots, x_l)$$

The circuit then proves knowledge of $\{x_i\}$ such that $H(x_1, \ldots, x_l) = h$.

The main choices for the hash function are Keccak-256, which is the native hash function of EVM blockchains, and Poseidon, a well-known zk-friendly hash function for which several Solidity implementations are available. A cost comparison is shown in Table 3.

Table 3: Hashing Cost Comparison

Hash function	**In-circuit**	**On-chain**
Keccak-256	140,000-150,000	30 gas
Poseidon	250-300	$\sim 10,000$ gas

A first observation is that Keccak cannot be reasonably used in-circuit. As for Poseidon, its gas cost is higher than the per-input gas cost of a Groth16 verifier. It therefore only makes sense if an input vector is used multiple times, or if a reasonable on-chain randomness is not available. Otherwise, using BRLC (cf below) is the preferred option.

In our case, we could use a hash function to commit to the list of public keys **pub** during DKG phase 3. This is a vector of n public keys, represented by $2n$ scalars. As these public keys will then be used by all n participants in DKG phase 4, reducing them to a single scalar will reduce the number of scalar inputs in DKG phase 3 by $2n - 1$.

6.2 Binding Random Linear Combination (BRLC)

BRLC means committing to a vector $\vec{v} = (v_1, \ldots, v_l)$ using a random linear combination:

$$C = \sum_{i=1}^{l} \rho^i \, v_i$$

The challenge ρ must be unpredictable at the time the inputs are committed, e.g., derived from a block hash or Fiat-Shamir. If that is the case, BRLC provides computational binding under the discrete log assumption.

In-circuit cost. The circuit must recompute the sum $\sum \rho^i \, v_i$. Computing ρ^i corresponds to $l - 1$ field multiplications, and computing products to l field multiplications. Ignoring the cost of the $l - 1$ additions, we have therefore $\sim 2l$ constraints, which is quite low.

On-chain cost. The ADDMOD and MULMOD operations (addition and multiplication modulo an arbitrary integer) cost only 8 gas each, so the cost of computing the commitment for a vector of length l will only be $3l \times 8 = 24l$ gas. Adding loop overhead gets us to $\approx 70l$ gas.

Bottom line: the cost of committing an element using BRLC is only about 70 gas. This excludes calldata or storage access costs, but these would also have to be paid for direct public inputs. Thus, there is significant gas-saving potential, especially in DKG phases 4 and 5, given the large number of public inputs required in these phases. There is a catch, however: the challenge ρ should not be known when the participant computes the public inputs, but it must be known to compute the ZK proof.

One solution is to replace a single blockchain interaction with three. First, store all relevant public inputs on-chain. Second, call the smart contract again in a later block to set ρ as the hash of the previous block[1] This second step is necessary because when calling a smart contract at block b, the block hash at block $b - 1$ is already known. Finally, submit the zk proof computed using ρ.

Apart from increased complexity and gas costs of additional interactions with the smart contract, this solution has another important drawback. Data cannot be submitted as calldata, which costs at most 512 gas, but must first be stored and later read from storage. This costs a total of $2 \times 2,100 = 4,200$ gas per scalar, significantly reducing potential gas savings.

The second solution is to use Fiat-Shamir and set ρ to be the hash of one or more values that are part of the commitments. These values should be carefully selected and also include a unique identifier, such as the block hash from a previous interaction with the smart contract. At a cost of only a few Poseidon hashes, any number of public inputs can thus be efficiently converted to just one input.

[1] It can be any block, as long as it's a later block. This is secure for this application even though miners can influence block contents. If a uniform distribution is preferred, the block hash can be hashed again with Poseidon.

As a result, the protocol can be efficiently implemented even up to ≈ 50 participants, which requires sending $\approx 1,250$ scalars as calldata to the smart contract in DKG phase 5. Based on the default transaction size limit of 128KB, up to approx. 4,000 scalars can be sent to a smart contract, so the theoretical maximum is almost 90 nodes; memory expansion costs would then have to be taken into account.

7 Conclusion

We have presented a blockchain-assisted DKG protocol that eliminates interactive complaint procedures entirely through systematic use of zero-knowledge proofs. Every participant proves the correctness of their contribution in a single zk-SNARK submitted alongside their data. The smart contract rejects invalid contributions at submission time, removing the need for dispute resolution phases and the class of attacks they enable.

The protocol is built from standard primitives and targets deployment on EVM-compatible chains. As shown in Section 6, naïvely passing all public inputs to the verifier is impractical for committees of any realistic size: DKG phase 4 alone requires $\approx 6n$ scalar inputs per participant. Using binding random linear combinations with Fiat-Shamir challenges, the number of public inputs can be reduced to a small constant, bringing per-participant verification costs close to the base cost of a single zk-SNARK verification plus calldata costs that scale linearly in n. Concrete gas figures will depend on implementation choices and are left to future benchmarking work.

Limitations. Several limitations should be noted. First, the protocol inherits the timing model of the underlying blockchain, typically partial synchrony rather than full asynchrony. Phases are delimited by block numbers, and participants who miss their window are excluded. This is appropriate for blockchain-native applications but does not provide the guarantees of fully asynchronous ADKG protocols. Second, the protocol does not provide adaptive security: the set of participants is fixed before the protocol begins, and security relies on the assumption that the adversary's corruption choices are static. Finally, while the public input reduction strategies analyzed in Section 6 substantially reduce gas costs, on-chain verification may remain prohibitive on Ethereum mainnet (Layer 1). Deployment on Layer 2 scaling networks, which offer significantly lower execution costs, is more practical for larger values of n.

Future work. Concrete implementation and benchmarking is the most immediate next step: circuit sizes, proof generation times, and end-to-end gas costs for realistic parameter choices ($n \in \{10, 20, 40\}$) would complement the qualitative cost analysis presented here. Formal security analysis of the protocol, including a precise characterisation of the assumptions under which the ZK-ification preserves the security of the underlying Pedersen DKG, is an important theoretical direction. Integration with proactive secret sharing would enable periodic key refresh, limiting the window of adversarial advantage in long-lived deployments. Finally, the gas economics of L2 rollups may shift the optimal tradeoffs for public input reduction

and deserve dedicated analysis.

The gap between sophisticated but undeployed ADKG protocols and simple but limited production systems remains wide. This work contributes a protocol that occupies a practical middle ground: not optimal in any single theoretical dimension, but deployable, auditable, and built on primitives whose security properties are well understood.

Appendix 1 DKG Process Diagram

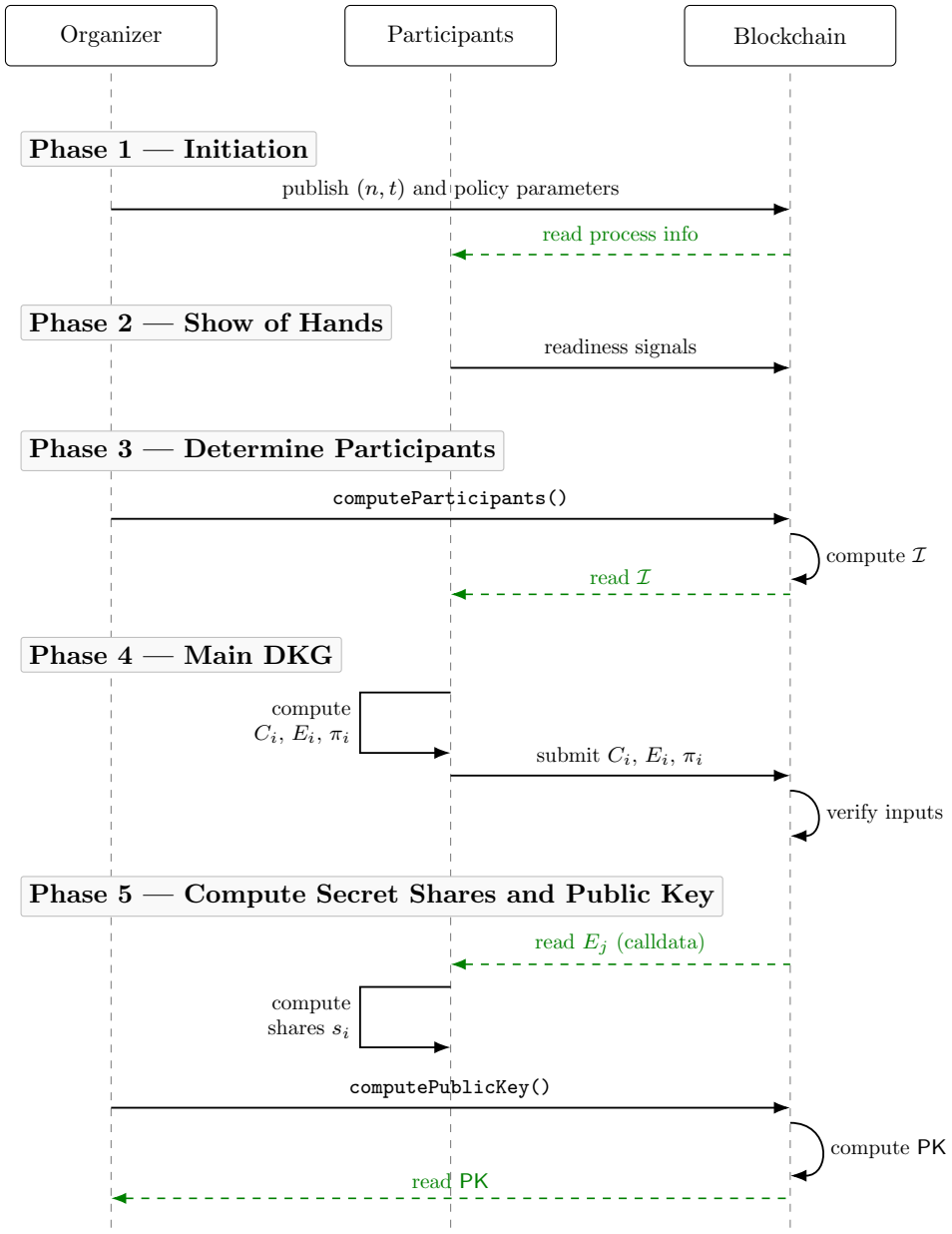

Figure 1: DKG protocol sequence diagram

Appendix 2 Threshold Decryption Process Diagram

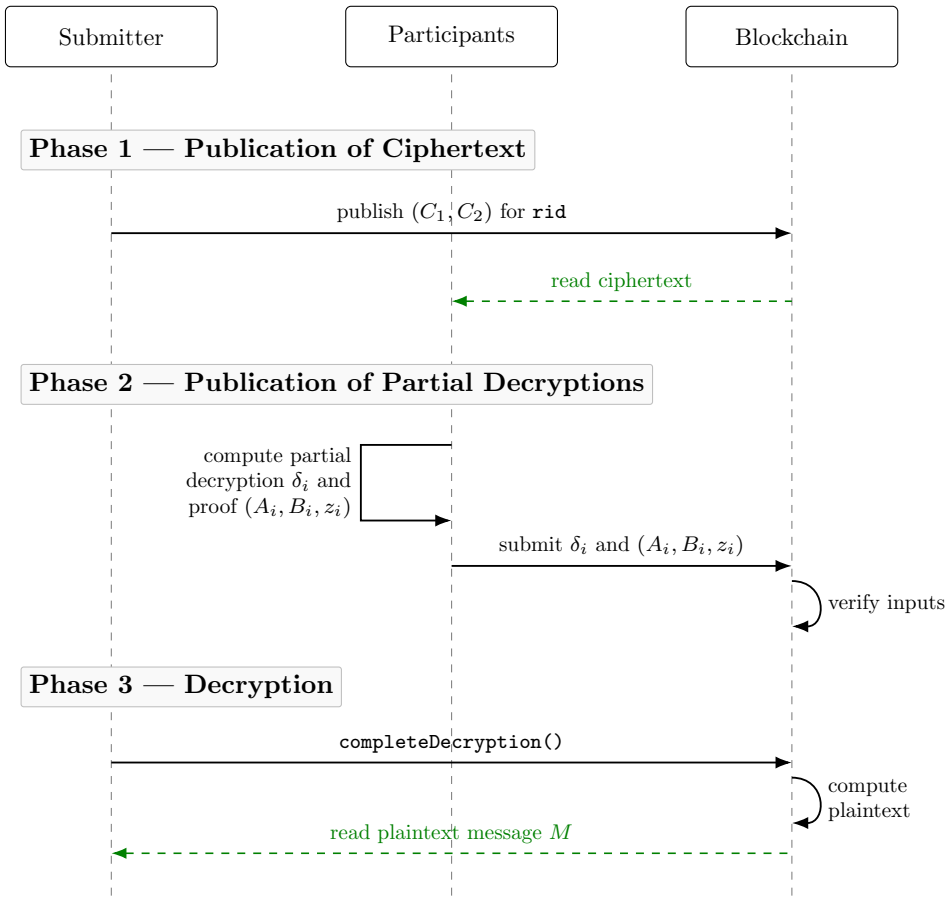

Figure 2: Threshold decryption sequence diagram

Appendix 3 Appendix 3: Secret Key Disclosure Process Diagram

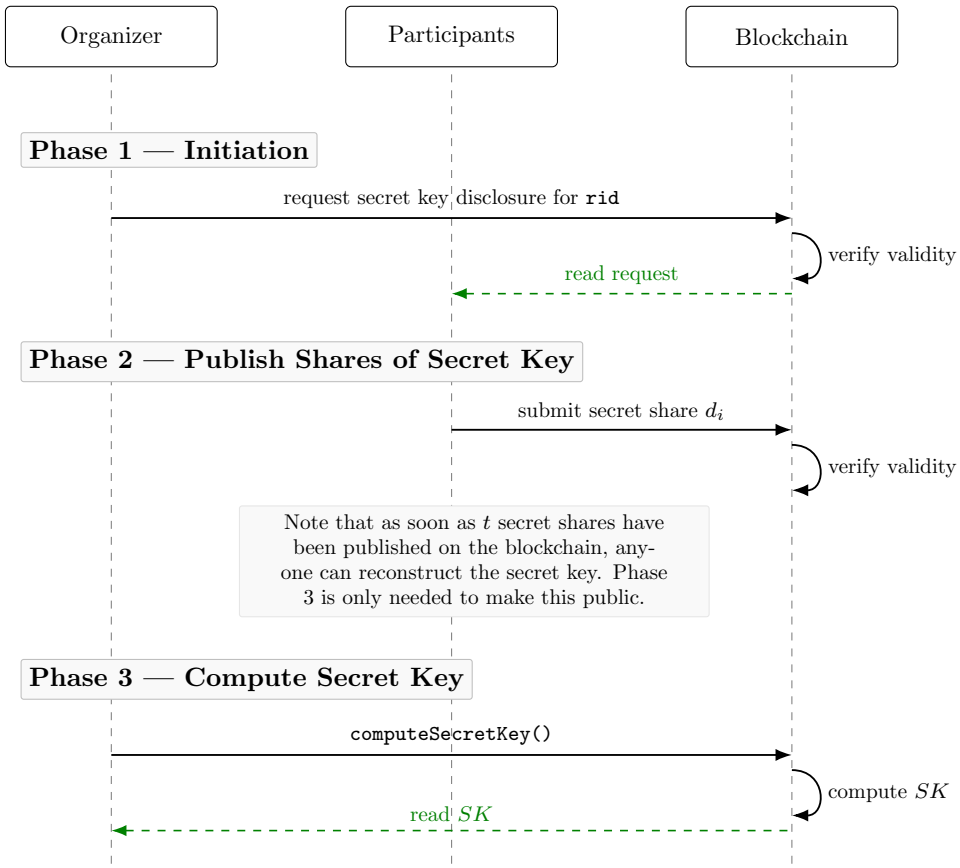

Figure 3: Secret key disclosure sequence diagram.

Appendix 4 Hashed ElGamal Encryption

For the encryption of the shares, a straightforward choice is the standard Hashed ElGamal construction, which is the Diffie–Hellman KEM–DEM core underlying DHIES / ECIES [7, 1]. Since our plaintext space is $\mathbb{F}_q$, we instantiate the DEM as additive masking $c = m + \mathsf{HashToField}(S) \bmod q$, where the hash output is reduced modulo q.

Participant i must encrypt the share $s_i(j) \in \mathbb{F}_q$ in a ZK-friendly manner. Let participant j have the BabyJubjub keypair:

$$\mathsf{sk}_j \in \mathbb{F}_q, \qquad \mathsf{pub}_j = \mathsf{sk}_j\, G$$

To encrypt a message $m \in \mathbb{F}_q$ for participant j, pick $r \in \mathbb{F}_q$ (with $r \neq 0$) and compute:

$$\begin{aligned} R &= r\, G && \text{(ephemeral public key)} \\ S &= r\, \mathsf{pub}_j && \text{(ephemeral secret point)} \\ s &= \mathsf{Hash}(S) && \text{(shared secret scalar)} \\ c &= m + s \pmod q && \text{(masked message scalar)} \end{aligned}$$

Note that Hash is a hash function that maps BabyJubjub elliptic curve points to scalars in $\mathbb{F}_q$. As is common in modern presentations, one may instead derive the pad as $s := \mathsf{Hash}(R \parallel S)$ in order to bind the derived scalar explicitly to the ephemeral public key.

The ciphertext is then:

$$\mathsf{Enc}_{\mathtt{share}}[\mathsf{pub}_j, s_i(j)] = (R, c)$$

The decryption works as follows:

$$\begin{aligned} S' &= \mathsf{sk}_j\, R && \text{(recover secret point)} \\ s' &= \mathsf{Hash}(S') && \text{(recover secret scalar)} \\ m &= c - s' \pmod q && \text{(unmask message scalar)} \end{aligned}$$

To verify correctness of (R, c), the prover supplies a zero-knowledge proof of knowledge of r such that:

$$\begin{aligned} R &\stackrel{?}{=} r\, G \\ c &\stackrel{?}{=} m + \mathsf{Hash}(r\, \mathsf{pub}_j) \bmod (q) \end{aligned}$$

All operations remain inside BabyJubjub and $\mathbb{F}_q$.

Note that Hashed ElGamal is IND-CPA secure but malleable. In our protocol, the contract accepts only ciphertexts accompanied by valid zero-knowledge proofs of correct formation. This prevents adversaries from submitting malformed or adaptively manipulated ciphertexts. Within this restricted protocol model, the malleability of Hashed ElGamal does not yield practical attacks.

References

[1] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. *DHIES: An Encryption Scheme Based on the Diffie–Hellman Problem*. Tech. rep. Full version dated September 18, 2001. Cryptology ePrint / Full version on the authors' webpages, Sept. 2001. URL: `https://web.cs.ucdavis.edu/~rogaway/papers/dhies.pdf`.

[2] Ittai Abraham et al. "Bingo: Adaptivity and Asynchrony in Verifiable Secret Sharing and Distributed Key Generation". In: *Advances in Cryptology – CRYPTO 2023*. Springer, 2023, pp. 39–70.

[3] Ittai Abraham et al. *Nearly Quadratic Asynchronous Distributed Key Generation*. IACR ePrint 2025/006. 2025.

[4] Ittai Abraham et al. "Reaching consensus for asynchronous distributed key generation". In: *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 2021, pp. 363–373.

[5] Benny Applebaum and Benny Pinkas. "Distributing Keys and Random Secrets with Constant Complexity". In: *Theory of Cryptography Conference (TCC 2024)*. Vol. 15366. LNCS. Springer, 2024, pp. 478–510.

[6] Renas Bacho and Alireza Kavousi. "SoK: Dlog-based Distributed Key Generation". In: *IEEE Symposium on Security and Privacy (S&P)*. 2025, pp. 614–632.

[7] David Cash, Eike Kiltz, and Victor Shoup. "The Twin Diffie–Hellman Problem and Applications". In: *Advances in Cryptology – EUROCRYPT 2008*. Vol. 4965. Lecture Notes in Computer Science. Springer, 2008, pp. 127–145. DOI: `10.1007/978-3-540-78967-3_8`.

[8] Sourav Das et al. "Practical asynchronous distributed key generation". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 2518–2534.

[9] Sourav Das et al. "Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling". In: *32nd USENIX Security Symposium*. 2023, pp. 5359–5376.

[10] Paul Feldman. "A practical scheme for non-interactive verifiable secret sharing". In: *28th Annual Symposium on Foundations of Computer Science*. IEEE. 1987, pp. 427–438.

[11] Hao Feng and Qiang Tang. "Asymptotically Optimal Adaptive Asynchronous Common Coin and DKG with Silent Setup". In: *Advances in Cryptology – CRYPTO 2025*. Springer, 2025, pp. 647–680.

[12] Ariel Gabizon and Zachary J. Williamson. *fflonk: a Fast-Fourier Inspired Verifier Efficient Version of PlonK*. Cryptology ePrint Archive, Report 2021/1167. 2021. URL: `https://eprint.iacr.org/2021/1167`.

[13] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *Advances in Cryptology – EUROCRYPT 2016*. Vol. 9666. Lecture Notes in Computer Science. Springer, 2016, pp. 305–326. DOI: `10.1007/978-3-662-49896-5_11`.

[14] Eleftherios Kokoris-Kogias, Dahlia Malkhi, and Alexander Spiegelman. "Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 1751–1767.

[15] Torben Pryds Pedersen. "A threshold cryptosystem without a trusted party". In: *Advances in Cryptology—EUROCRYPT'91*. Springer, 1991, pp. 522–526.

[16] Philipp Schindler et al. *ETHDKG: Distributed Key Generation with Ethereum Smart Contracts*. Cryptology ePrint Archive, Report 2019/985. 2019.

[17] Adi Shamir. "How to Share a Secret". In: *Communications of the ACM* 22.11 (1979), pp. 612–613. DOI: 10.1145/359168.359176.

[18] Michael Sober et al. "Distributed Key Generation with Smart Contracts using zk-SNARKs". In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC)*. Best Paper Award, Distributed Systems Track. ACM. 2023, pp. 229–238. DOI: 10.1145/3555776.3577677.

[19] Shutter Network Team. *Shutter: Distributed Key Generation with Applications to Front-Running Protection*. Cryptology ePrint Archive, Report 2024/1981. Deployed on Gnosis Chain since October 2022. 2024.

[20] Vocdoni. "Gas Cost Analysis for ZK Proof Verification on Ethereum". Internal technical report. 2026.

[21] Haibin Zhang, Sisi Duan, et al. "Practical Asynchronous Distributed Key Generation: Improved Efficiency, Weaker Assumption, and Standard Model". In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2023, pp. 568–581.

[22] Liang Zhang et al. "1-Round Distributed Key Generation with Efficient Reconstruction Using Decentralized CP-ABE". In: *IEEE Transactions on Information Forensics and Security* 17 (2022), pp. 894–907. DOI: 10.1109/TIFS.2022.3152356.