D-KODE: Mechanism to Generate and Maintain a Billion Keys

Easwar Vivek Mangipudi   Aniket Kate
Purdue University
{emangipu,aniket}@purdue.edu

ABSTRACT

This work considers two prominent key management problems in the blockchain space: (i) allowing a (distributed) blockchain system to securely airdrop/send some tokens to a potential client Bob, who is yet to set up the required cryptographic key for the system, and (ii) creating a (distributed) cross-chain bridge that allows interoperability at scale by allowing a (changing) set of nodes in a blockchain to perform transactions on the other blockchain.

The existing solutions for the first problem need Bob to either generate and maintain private keys locally for the first time in his life—a usability bottleneck—or place trust in third-party custodial services—a privacy and censorship nightmare. Towards solving both problems in a distributed setting against a threshold-bounded adversary, distributed key generation (DKG) based solutions are actively employed; here, a set of servers generate the transaction keys in a distributed manner and link them to clients' ids. Nevertheless, these solutions introduce computation and communication overhead that is linear in the number of keys and do not scale well even for a million keys, especially for proactive security against a mobile adversary.

This work presents a Keys-On-Demand (D-KODE) [1] distributed protocol suite that lets the blockchain system securely generate the public key of any Bob against a mobile, threshold adversary. Multiple servers, here, compute discrete-log private/public keys on the fly through distributed pseudo-random function evaluations on the queried public string. D-KODE also introduces a proactive security mechanism for the employed black-box secret-sharing based DKG to maintain the system's longitudinal security. The proposed protocol scales well for a very high number of keys as its communication and computation complexity is independent of the number of keys. Our experimental analysis demonstrates that, for a 20-node network with 2/3 honest majority, D-KODE starts to outperform the state of the art as the number of keys reaches 94K. D-KODE is practical as it takes less than 100msec to generate a secret key for a single-threaded server in a 20-node setup.

1 INTRODUCTION

As blockchain systems proliferate, we increasingly tokenize financial and supply-chain assets using cryptographic (private/signing) keys. The total number of keys generated in the cryptocurrency systems is increasing rapidly. According to a recent repor [8], in the year 2021, roughly 500, 000 Bitcoin keys have been generated per day, amounting to around 88 million keys in the first half of the year 2021 for Bitcoin alone. This extensive use of cryptographic keys brings interesting security and scalability challenges that need immediate attention. For example, if a user loses their private key, they lose the associated assets—there is no recovery mechanism as with the typical password-based authentication. Given the general lack of familiarity with the technical aspects of cryptographic key management and maintenance, most first-time users choose custodial wallets [2, 6, 10], where a third party controls their keys. However, these third parties become single points of failure for large-scale thefts, financial surveillance, and censorship. In general, this key management problem, combined with a lack of simpler tools for key setup, is a bottleneck of blockchain adoption.

Cross-chain bridges and Airdrops. Allowing communication between two or more blockchains (or blockchain interoperability) brings further challenges. Today, this is typically achieved through smart contracts called "bridges" and nodes that track a transaction initiating blockchain and a destination blockchain. The nodes here are trusted with the keys and are ideal targets of the adversary as it is expected that they may soon be responsible for significant cross-chain currency transfers. In this cross-chain bridge scenario, a group of servers of one blockchain sign transactions (on behalf of a user) on a different blockchain. When a user requests, the servers generate private key shares corresponding to the user and threshold sign the transaction.

In the airdrop [1, 5, 7] scenario, a crypto firm wishes to send some funds to Bob, who does not have a public key address on their system yet. This can be because Bob either has never generated a key pair and is not available to engage immediately, or Bob is offline with his already generated public key not being available. The firm should be able to compute the public key corresponding to Bob's public string (identity) such that later Bob can use the same string to generate the related private key and claim funds.

Existing distributed key generation (DKG) approaches. Current solutions [12, 14] off-load the key generation and storage to a set of n servers while preserving the secrecy of the keys against any t compromised servers. The servers generate key-shares in a distributed form by running a distributed key generation (DKG) [40] instance for each identity and providing the secret key or public key shares for the identity as required. These architectures do not scale well because the servers have to perform several DKG instances to generate the key shares for all the keys resulting in high computational and communication overhead. The overhead further amplifies if the system, over longer terms, attempts to provide proactive security [45] against mobile adversary [57]: All the millions of key shares need to be refreshed periodically, giving rise to issues of availability while the computation and communication-intensive refreshing process are in progress.

Start-ups such as Torus [14], Keep Network [9], Chainlink [3] are developing similar threshold cryptographic solutions towards maintaining secrecy and availability of the clients' keys; the motivating factor for this work is that their current approaches do not scale well with keys and cross-chain bridges. This work aims to provide a scalable key management system to generate keys on the

[1] D in D-KODE indicates discrete-log. This is to differentiate from other key types.

fly for the rapid proliferation of blockchains to millions of users and bridges amounting to millions or even billions of keys.

Employing distributed PRF. In this work, we generate keys on-the-fly as pseudo-random function (PRF) [22, 42, 53] evaluations. A PRF is a deterministic function of a master (private) key and an input tag indistinguishable from a truly random function of the input. We plan to use the PRF output as a private/signing key. As a single node holding a master key K introduces a key escrow and a single-point-of-failure for PRFs, we distribute the trust using distributed PRF (DPRF) such that a set of servers holds the master key K in a secret shared fashion and generates shares of the client's private keys as partial PRF evaluations. Indeed, generating private keys using DPRFs [28, 33, 54] is considered in the literature; however, none of the existing solutions is suitable for the scenario involving any Alice obtaining public keys of an offline Bob.

As an illustrative example consider private key generation for an identity (tag) ID_A using the well-known PRF by Naor *et al.* [33, 54]. This involves computing $sk_A = H_2(F(K, \mathsf{ID}_A)) = H_2(H_1(\mathsf{ID}_A)^K)$, where hash functions $H_1(\cdot)$ and $H_2(\cdot)$ map to a multiplicative group (of elliptic curve points) $\mathbb{G}$ and a scalar additive group $\mathbb{Z}_p$ respectively. When the key K is shared among multiple servers, computing her secret key sk_A from partial evaluations is straightforward for Alice: she first computes $H_1(\mathsf{ID}_A)^K$ using Lagrange interpolations and then applies H_2 to the output locally. The airdrop scenario, however, asks to securely provide Alice the *public key* pk_B of an offline party Bob with identity ID_B. To ensure that Alice cannot determine sk_B, computation of $pk_B = g^{H_2(F(K,\mathsf{ID}_B))}$ involves computing hash function $H_2(\cdot)$ through multi-party computation (MPC)—a highly expensive process in the threshold setting [18, 43].

To generate the public keys efficiently, we need a PRF whose output is a scalar value in $\mathbb{Z}_p$ and does not involve $H_2(\cdot)$ hash computations in the multi-party setting. We observe that most other distributed PRF [38, 55, 56] and easy-to-distribute key-homomorphic PRF constructions [37] in literature do not satisfy this requirement.

Our Approach. We employ the PRF in the lattice-based cryptography setting [28], $F(X, \mathbf{k}) = \left\lfloor H(X) \cdot \mathbf{k} \right\rfloor_p \in \mathbb{Z}_p, \mathbf{k} \in \mathbb{Z}_q^u, H(\cdot) \in \mathbb{Z}_q^u, p < q$ to generate keys as in Figure 1. It is an almost-key homomorphic PRF, with an error $\{0, 1\}$ in the evaluation for every additive term. The master key $\mathbf{k}$ is threshold-shared among the servers. However, unlike standard threshold designs [14, 17], we cannot employ Shamir secret sharing (SSS) [59] for sharing $\mathbf{k}$ in almost-homomorphic PRF as the reconstruction (Lagrange) coefficients blow up the error (and error combinations) when computing the PRF output from the partial evaluations. Another common secret sharing mechanism, replicated secret sharing (RSS) [34, 47] may be employed as the RSS shares need to be simply added to compute the value, which ensures that the error remains bounded within the range $[-n, n]$. However, the number of RSS shares grows exponentially as $\binom{n-1}{t}$ for an (n, t) threshold structure among servers with $t = O(n)$; this has high share-refreshing computation overhead and RSS-based distributed PRF can only be applied to settings with ten or lower servers. Therefore, solving our distributed PRF problem requires going beyond the commonly employed SSS and RSS schemes.

In this work, we demonstrate that the black-box secret sharing (BBSS) approach [35] can be made practical towards catering to a higher number of servers; this is the first effort that realizes its utility in practice. We propose the D-KODE protocol, which generates discrete-log private and public keys using almost key-homomorphic PRF evaluations, where key-sharing among the servers is performed through BBSS. Our BBSS instantiation ensures that the evaluation coefficients are in the set $\{-1, 0, 1\}$, resulting in the output key being in a very small range of keys linear in the number of servers such that Bob can efficiently compute the private key associated with the public key employed by Alice to pay Bob.

In D-KODE, a single master key vector of 8192 $\mathbb{Z}_q$ elements is BBSS-shared among the servers making proactive secret sharing independent of the number of keys; resulting in only constant overhead for share refreshing. To allow the clients to verify the evaluations while generating keys, we propose a verifiability mechanism for the almost key-homomorphic PRF employed. Our prototype implementation provides D-KODE protocol with BBSS-DKG mechanism for network size up to 50 servers. We observe that D-KODE starts to outperform the state of the art at 94K keys for a 20-server system. Moreover, using D-KODE, a server supports generating upto ten `secp256k1` keys per second per thread.

In summary,

- We propose a solution D-KODE making airdrops of crypto funds possible for users who are not yet in the system. D-KODE helps generating keys where two parties like to transact when either or both the parties do not have mechanisms for locally generating keys; even when one of them is offline and the other party only knows his verifiable identity. D-KODE solution also achieves cross-chain bridges where a client can request a group of servers to sign transactions on their behalf.
- As a key step in D-KODE, we propose efficient approaches to realize black box secret sharing (BBSS) for practical setting, which can be of independent interest to threshold cryptography [11] community.
- We instantiate the first DKG mechanism using BBSS scheme and provide a dynamic committee proactive secret sharing scheme. Our scheme offers constant computational overhead and hence scales well with a large number of keys in the system.

2 SYSTEM SETUP AND SOLUTION OVERVIEW

2.1 System Setup

Consider a system of n servers $\{P_1, P_2, \cdots, P_n\}$ that share a master secret (vector) $\mathbf{k}$[2] through a (n, t)-threshold scheme. The servers interact with clients who join and leave the network anytime. All the servers have access to a broadcast channel and the network is synchronous. We consider a $t-$bounded static adversary that corrupts up-to t servers at the start of the protocol. Corrupted servers remain so through-out the protocol run. Each pair of servers is connected through a secure channel that provides secrecy and authenticity; this is typically achieved through TLS channels [15] which mitigate any man-in-the-middle attacks. While we consider a static adversary model for the distributed key generation mechanism, we

[2] We denote all vectors in bold font small and matrices in bold font capital letters.

extend it to a mobile adversary model for the proactive secret sharing mechanism discussed in Section 7. The secrecy/confidentiality of the secret key in the D-KODE-protocol is based on the discrete logarithm (DLog) and Learning-with-rounding assumptions:

Definition 2.1. The *Discrete Logarithm* (DLog) assumption [52]: For a generator $g \in \mathbb{G}$ and $a \xleftarrow{\$} \mathbb{Z}_q$, given the value g^a, the probability of a ppt algorithm $\mathcal{A}_{\text{DLog}}$ to output the value a, $\Pr[\mathcal{A}_{\text{DLog}}(g, g^a) = a]$ is negligible.

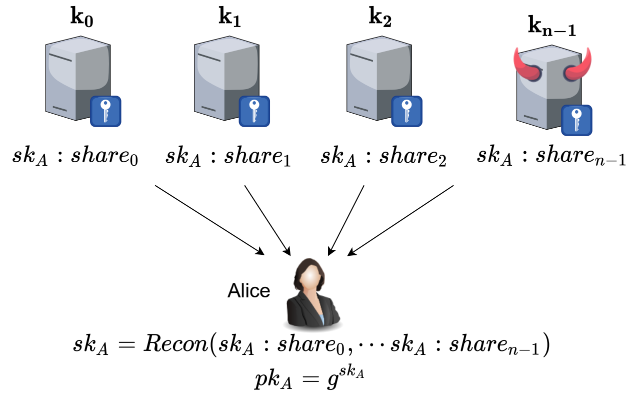
Definition 2.2. The *Learning-with-rounding* (LWR) [50] problem consists of distinguishing the distribution $(\mathbf{A}, \lfloor \mathbf{As} \rceil_p)$ where $\mathbf{A} \sim U(\mathbb{Z}_q^{\mathsf{m}\times\mathsf{n}})$, $\mathbf{s} \sim U(\mathbb{Z}_q^{\mathsf{n}})$ and the uniform distribution $U(\mathbb{Z}_q^{\mathsf{m}\times\mathsf{n}} \times \mathbb{Z}_p^{\mathsf{m}})$; $q \geq 2$. We say that the $\text{LWR}_{(q,\mathsf{m},\mathsf{n})}$ is hard if for all ppt algorithm $\mathcal{A}$, the advantage $\boldsymbol{Adv}^{\text{LWR}}_{q,\mathsf{m},\mathsf{n}}(\mathcal{A}) = |\Pr[\mathcal{A}(\mathbf{A}, \lfloor \mathbf{As} \rceil_p) = 1] - \Pr[\mathcal{A}(\mathbf{A},\mathbf{u}) = 1]|$ is negligible, with the probabilities taken over $\mathbf{A} \sim U(\mathbb{Z}_q^{\mathsf{m}\times\mathsf{n}})$, $\mathbf{s} \sim U(\mathbb{Z}_q^{\mathsf{n}})$, and $\mathbf{u} \sim U(\mathbb{Z}_p^{\mathsf{m}})$.

2.2 Design Overview

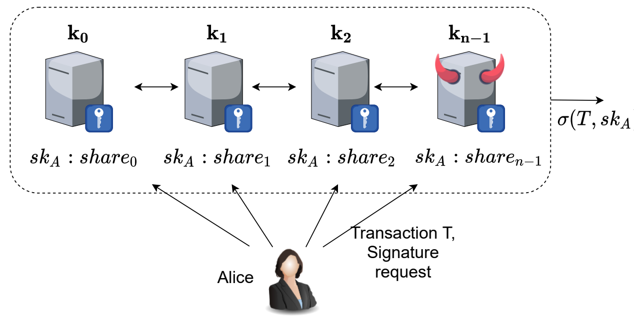
In the D-KODE protocol, a master key $\mathbf{k}$ is (n, t)-threshold secret-shared among n servers and the client private key is computed as the PRF [28] evaluation $F(X, \mathbf{k}) = \left\lfloor H(X) \cdot \mathbf{k} \right\rceil_p \in \mathbb{Z}_p$, for $X \in \mathcal{X}$ where $\mathcal{X}$ is the client-input space, $\mathbf{k} \in \mathbb{Z}_q^u$ the server key and $H : \{0, 1\}^* \rightarrow \mathbb{Z}_q^u$ a cryptographic hash function. $(\cdot)$ indicates the vector dot product computation (see Appendix A for details on the employed PRF). The master key vector $\mathbf{k}$ is shared among the servers with each server P_i obtaining the share matrix $\mathbf{K}_i$. The shares $\mathbf{K}_i$ are generated in a distributed manner using distributed key generation (DKG) involving verifiable black box secret sharing (BBSS) scheme (elaborated in Section 3). The BBSS scheme involves a *distribution matrix* which is constructed such that the reconstruction coefficients for the shares are in the set $\{-1, 0, 1\}$. It is done by realizing the (n, t)-threshold access structure as a threshold circuit and expressing it as a monotone boolean function. This function is then converted to a distribution matrix using the Benaloh and Leichter [24] construction (recalled in Appendix C).

Let each server P_i be associated with a set T_i such that P_i receives the matrix $\mathbf{K}_i = \{\mathbf{k}_j, j \in T_i\}, \mathbf{k}_j \in \mathbb{Z}_q^u$. The partial evaluations of server P_i upon client input X is a vector of evaluations $\{F(X, \mathbf{k}_j), j \in T_i\}$. To compute the required keys, the client forwards the public string X, obtains partial evaluations and reconstructs the corresponding keys. Let $y = F(X, \mathbf{k})$ and $y_\ell = F(X, \mathbf{k}_\ell), \ell \in \cup_i T_i$ be the set of all partial evaluations received from the servers. To generate the private key the client obtains a linear combination $\tilde{y} = \sum_{i \in S} \lambda_i \cdot y_i$ where each $\lambda_i \in \{0, 1, -1\}$. $\tilde{y}$ differs from y by a small error $\theta < |\sum_i T_i|$ depending the evaluations used for the computation.

(Scenario 1A) Private key generation. Alice securely authenticates herself to the servers (using email-login, OAuth tokens etc.) and forwards her public string ID_A (for example, her email ID), obtains the partial evaluations $y_\ell = F(\text{ID}_A, \mathbf{k}_\ell)$ from servers and computes the private key as $sk_A = \sum_i \lambda_i \cdot y_i$ as depicted in Figure 1a. The values of λ_i are determined by the qualified set of servers whose evaluations are utilized in the reconstruction (refer Section 3). From the private key sk_A, she can compute the public key as $pk_A = g^{sk_A}$. With the key pair (sk_A, pk_A), she can perform any required transaction.



(a) Scenario 1a: Alice uses her public string ID_A, obtains evaluations and reconstructs private key sk_A after authentication



(b) Scenario 1b: Alice uses her public string ID_A, sends a transaction T and requests the servers for a (threshold) signature on T after authentication

Figure 1: Private key and signature generation using servers with shares k_i of a master key K shared with a linear threshold scheme.

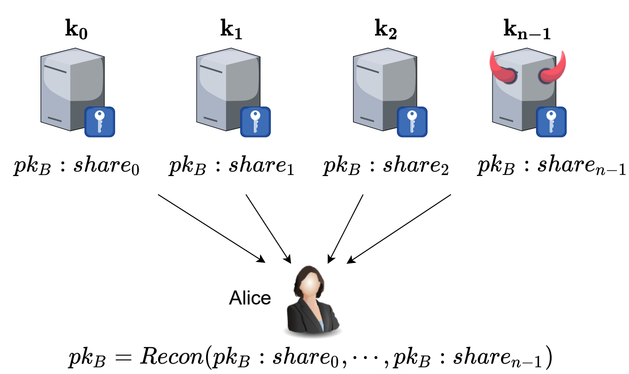


Figure 2: Scenario 2: Alice uses Bob's public string ID_B to obtain his public key shares and compute the public key pk_B

(Scenario 1B) Partial signature generation. Instead of requesting for the secret key shares to reconstruct the secret key, Alice can request the servers to generate shares and generate the signature on her behalf. Upon request, the servers can generate secret key shares and generate partial signatures using the secret key shares (see Figure 1b). These partial signatures from different servers are threshold-combined [29] to generate valid signature and authenticate any transaction. Alice forwards an identity string and a formed transaction to the servers, similar to the previous scenario. The servers generate the partial signature using the identity and sign the transaction. They also publish the public key corresponding to the secret key generated.

When a party wishes to verify the transaction by generating Alice's public key on the fly, the generated public key will have a slight 'error' of 2θ. Hence the verifying party generates a list of 4θ public keys and confirms the transaction if at least one matches the published public key and verifies the signature.

(Scenario 2) Public key of an offline Bob. When Alice tries to pay Bob, she forwards Bob's public string ID_B to the servers and obtains the evaluations $z_\ell = g^{y'_\ell}$ where $y'_\ell = F(\mathsf{ID}_B, \mathbf{k}_\ell)$ as depicted in Figure 2. She computes a public key of Bob as $pk_B = \prod_i (z_i)^{\lambda_i}$ and proceeds to pay Bob using the computed public key pk_B.

When Bob tries to compute his private key later corresponding to this public key pk_B, he authenticates to the servers and obtains a private key sk'_B which differs from the private key sk_B (corresponding to the public key pk_B), by a maximum of 2θ. He simply computes all the private keys in the range $[sk'_B - 2\theta, sk'_B + 2\theta]$, obtains the corresponding public keys $[g^{sk'_B - 2\theta}, g^{sk'_B + 2\theta}]$. For example, for twenty servers, θ is distributed among $[-216, 216]$ and highly concentrated around 0; each of the key can be generated by one multiplication from pk'_B. pk_B will be in that set of 4θ keys, and since he has private keys corresponding to all of them, he can utilize the funds transferred by Alice to pk_B. Note that only Bob owns these secret keys.

Thus Alice can airdrop cryptocurrency to Bob by computing pk_B. Bob can later compute the corresponding key sk_B and retrieve the funds whenever necessary. This solution does not involve any interaction between the servers for the computation of client keys, since they just evaluate $y'_i = F(\mathsf{ID}_B, \mathbf{k}_i)$ and forward $g^{y'_i}$ to the client non-interactively. In summary, the proposed solution for the two scenarios consists of the following steps:

- The servers $P_i, i \in [n]$ participate in DKG involving BBSS and obtain shares $\mathbf{K}_i = \{\mathbf{k}_j, j \in T_i\}$ of a master key $\mathbf{k}$.
- For Scenario 1: The servers generate partial evaluations $y_\ell = F(X, \mathbf{k}_\ell)$ using the server key shares $\mathbf{K}_i$ and public input string input X from the client. The client combines the shares to compute the private key evaluation $y = F(X, \mathbf{k})$.
- For Scenario 2: The servers evaluate $y'_i = F(X', \mathbf{k}_i)$ and forward $g^{y'_i}$ for the evaluation of public key $z = g^{y'}$ for the input X' from any client.

Since we envisage a full-fledged deployment where the servers are used to evaluate keys for a large number of clients over a long period, we propose a proactive secret sharing mechanism for BBSS. The servers store only one set of key shares corresponding to the master key $\mathbf{k}$ and perform share-refreshing periodically using the proposed Proactive BBSS scheme (refer Section 7). For share refreshing, the servers re-share each of their share elements to the set of servers in the next period. The servers then compute the new shares from the shares of the share-elements.

We implement the full protocol and extract many interesting aspects of BBSS scheme in the practical regime. While the existing works discussing BBSS and the related Linear Integer secret sharing (LISS) scheme [36, 50] have shown that the circuit size for the construction of distribution matrix varies from $O(n^{5.3}) - O(n^{2.414})$, we show that for certain threshold access structures, efficient construction can be achieved bringing the sharing scheme into a practical regime.

3 BLACK BOX SECRET SHARING—BBSS

In a secret sharing scheme [23, 25, 26, 59], a designated *dealer* shares a secret among a set of parties such that a certain subset of parties can interact to reconstruct the secret. All the subsets designated to reconstruct the secret are *qualified* sets, and the set of all qualified sets is called an access structure. The threshold-t access structure $T_{(n,t)}$ is the collection of subsets of parties of cardinality greater than t. Any subset of parties outside the access structure has no information about the secret. When the total number of parties is n, we denote such a scheme as (n, t)-secret sharing, where at least $t + 1$ parties are needed for reconstruction.

A black-box secret sharing scheme is a linear secret sharing scheme over a finite Abelian group; it can be instantiated with just black-box access to group operations and random group elements. The secret generation and reconstruction are by a linear combination of share elements; the mechanism is independent of the group used for the secret sharing. We use a construction of the black-box secret sharing scheme such that the reconstruction coefficients lie in the set $\{-1, 0, 1\}$.

In black-box secret sharing [35], the dealer shares an element of an Abelian group (e.g., $\mathbb{Z}_q$ with publicly known q) where the share elements are computed as a linear combination of the secret value and random elements chosen by the dealer. They are computed by multiplication of a distribution matrix $\mathbf{M}$ and the random element vector $\boldsymbol{\rho}$. Any set of parties from the qualified set can reconstruct the secret as a linear combination of their shares.

Share generation. Consider a dealer sharing a secret $s \in \mathbb{Z}_q$ with a set of parties over the (monotone) access structure denoted by Γ. To generate shares for the parties in BBSS, the dealer uses a distribution matrix $\mathbf{M} \in \mathbb{Z}^{d \times e}$ and a distribution vector $\boldsymbol{\rho} = (s, \rho_2, \rho_3, \cdots, \rho_e)^T$ with secret s, $\{\rho_i\}_{i=2}^{e}$ uniform randomly chosen from $\mathbb{Z}_q$. The vector of share elements $\mathbf{s} = (s_1, s_2, \cdots, s_d)^T$ is computed as $\mathbf{s} = \mathbf{M} \cdot \boldsymbol{\rho}$.

Each party $P_i, i \in \{1, 2, \cdots, n\}$ is assigned a set of share elements using a surjective function $\psi : \{1, \cdots, d\} \rightarrow \{1, \cdots, n\}, d > n$. The i^{th} share element s_i is assigned to the party $\psi(i)$ who is said to *own* the i^{th} row of the matrix $\mathbf{M}$. For any subset of shareholders A, $\mathbf{M}_A \in \mathbb{Z}^{d_A \times e}$, $\mathbf{s}_A \in \mathbb{Z}^{d_A}$ denote the set of rows of $\mathbf{M}$ and elements of $\mathbf{s}$ jointly owned by the parties in A. We let $T_j = \psi^{-1}(j)$ be the set of all row indices held by party P_j. Any set $A \in \Gamma$ is a qualified set and sets $A \notin \Gamma$ are *forbidden* sets. The j^{th} share holder holds $d_j = |\psi^{-1}(j)|$ number of share-units.

The tuple $\mathcal{M} = (\mathbf{M}, \psi, \epsilon)$ is called an Integer span program (ISP) when $\mathbf{M} \in \mathbb{Z}^{d \times e}$ and the rows of $\mathbf{M}$ are labelled by the surjective function ψ. $\boldsymbol{\varepsilon} = \{1, 0, \cdots, 0\} \in \mathbb{Z}^e$ is called the target vector. When $\mathcal{M}$ is an ISP for Γ, the conditions specified by Definition 3.1 hold and $\mathbf{M}$ can be used as a distribution matrix to realize the access structure. This defines a reconstruction vector, which is used to reconstruct the secret when $\mathbf{M}$ is used as distribution matrix to share the secret value.

Definition 3.1. An integer span program (ISP) [35, 36] $\mathcal{M} = (M, \psi, \epsilon)$ is an ISP of the access structure Γ if for all $A \in \{1, 2, \cdots, n\}$ the following holds: If $A \in \Gamma$, then there exists a *reconstruction* vector $\boldsymbol{\lambda}_A \in \mathbb{Z}^{d_A}$ such that $\mathbf{M}_A^\top \boldsymbol{\lambda}_A = \boldsymbol{\varepsilon}$, where $\boldsymbol{\varepsilon} = \{1, 0, \cdots, 0\}$. If $A \notin \Gamma$, there exists a *sweeping* vector $\mathbf{k} = (k_1, k_2, \cdots, k_e) \in \mathbb{Z}^e$ such that $\mathbf{M}_A \mathbf{k} = \mathbf{0} \in \mathbb{Z}^d$ with $\mathbf{k}^\top \cdot \boldsymbol{\varepsilon} = 1$.

The first condition states that for every qualified set, there exists a reconstruction vector, thereby making the reconstruction of the shared secret possible.

Reconstruction. For a qualified set A, the secret value s is reconstructed as $s = \mathbf{s}_A^\top \cdot \boldsymbol{\lambda}_A$. Here $\mathbf{s}_A$ is the vector of all share elements (subset of vector $\mathbf{s}$) held by the parties in the set A and $\boldsymbol{\lambda}_A$ is the corresponding reconstruction vector.

To realize a threshold access structure, one needs to compute the corresponding distribution matrix $\mathbf{M}$. For that, we use the Benaloh-Leichter (BL) secret sharing construction [24, 36] where the access structure is expressed as monotone boolean formulae. The BBSS scheme using the BL construction ensures that elements of the reconstruction vector $\boldsymbol{\lambda}$ are small and in $\{-1, 0, 1\}$. We recall the BL construction of generating a distribution matrix from a monotone boolean formula representation of threshold structure in Appendix C. Due to space constraints, we also shift the verifiable BBSS description to Section 5.1.

4 DISTRIBUTION MATRIX FROM THRESHOLD FUNCTION

To generate the distribution matrix $\mathbf{M}$ for a (n, t) threshold BBSS scheme used in the DKG mechanism, we realize the (n, t) threshold access structure as a *threshold circuit* of *sufficient* depth. We convert the monotone boolean function representation of the circuit to the distribution matrix using the Benaloh-Leichter (BL) [24, 36] construction (recalled in Appendix C). Much of the previous works [30, 36, 63] suggest realizing the threshold access structure using a majority function [63]. Valiant [63] first proved that a polynomial size monotone circuit is realizable for majority circuit and provided a construction of size $O(n^{5.3})$, while the work by Hooray *et al.* [46] further improved the size of the circuit to $O(n^{1+\sqrt{2}})$. Valiant[63] suggested realizing threshold function using majority circuit of $2n$ variables [3] which was adapted by other works like Damgard *et al.* [36] following similar approach. Also, the proposed constructions [46, 63] are probabilistic, and the depth of the circuits is such that the probability with which the circuit outputs 1, on a majority in the n input variables, is $1 - \mathsf{e}$ where $\mathsf{e} = 2^{-n}$. This work computes the required threshold circuit directly instead of realizing the threshold circuit using the majority circuit. Also, we report that choosing $\mathsf{e} = 2^{-n}$ is indeed an overkill increasing the depth of the circuit.

Table 1: m values obtained through threshold circuit for different n, p values and error margins

n	$\mathsf{e} = 2^{-n}$		$\mathsf{e} = 2^{-\frac{n}{4}}$	
	$\mathsf{p} = 0.5$	$\mathsf{p} = 0.66$	$\mathsf{p} = 0.5$	$\mathsf{p} = 0.66$
5	81	9	9	9
10	2187	81	81	27
20	59049	729	2187	27
30	177147	2187	19683	81

Table 2: Dimensions of Distribution matrix M for different m

m	Rows	Columns
3	6	4
9	36	22
27	216	130
81	1296	778
243	7776	4666

Larger $\mathsf{e} > 2^{-n}$ is sufficient to realize the required access structure in the practical system profiles considered. Essentially, we relax the error to achieve efficient implementation while still reconstructing the secret for all the qualified sets of the access structure.

We adapt the construction provided by Goldreich [41] for the majority circuit construction that uses a MAJ3 probability amplifier node[4] (Refer Appendix B for a brief description of Goldreich's [41] construction and analysis of the majority circuit). The construction as depicted in Figure 3 in Appendix consists of n variables $x_i, i \in [n]$ and m variables $y_j, j \in [m]$ are assigned as follows: choose random indices i uniformly between 1 and n and assign the corresponding x_i to each $y_j, j \in [m]$ sequentially. Construct a 3-ary tree of MAJ3 nodes with y_j as leaves. The probability $\mathsf{p} = \Pr(y_j = 1)$ is taken as 0.5 for designing a majority circuit. However, we choose the value of p as $\frac{t}{n}$ for the threshold access structure (n, t), we also compute depth with e= $2^{-\frac{n}{4}}$. To see why this is significant, we first present how the dimensions of the distribution matrix $\boldsymbol{M}$ are related to the value m, the number of leaves in the circuit. Table 2 presents m values and the dimensions of $\mathbf{M}$ when the circuit is constructed using MAJ3 nodes and the distribution matrix is constructed by BL construction [24, 36] from the monotone boolean formula representation of the circuit. With the above construction, the number of rows of matrix $\mathbf{M}$ grow as $6^{\log_3(m)}$. Table 1 depicts the value of m needed to represent the threshold access structure for different values of p and e. For instance, from Table 2 for $m = 243$, the number of rows of $\mathbf{M}$ is 7776. Observe from Table 1 that for $(n, \mathsf{p}, \mathsf{e}) = (20, 0.5, 2^{-n})$, the value $m = 59049$. For $m = 243$ itself, the number of rows is 7776, for $m = 59049$ the number of rows make it extremely difficult (almost impossible) to perform the secret sharing on a laptop or a phone using a majority circuit implementation ($\mathsf{p} = 0.5$) with $\mathsf{e} = 2^{-n}$. However, through implementation (by computing different threshold combinations) we find that $\mathsf{e} = 2^{-\frac{n}{4}}$ is indeed sufficient to successfully reconstruct the secret for the qualified sets for number of servers n up to 50.

[3]For (n, t) threshold function, take n extra variables (total $2n$ variables), fix $n - t$ of them to be 1 and the rest t to 0; whenever there are more than t 1s in the original n variables, the majority function outputs 1.

[4]The MAJ3 node realizes majority of 3 variables (x_1, x_2, x_3) as $x_1x_2 + x_2x_3 + x_1x_3$

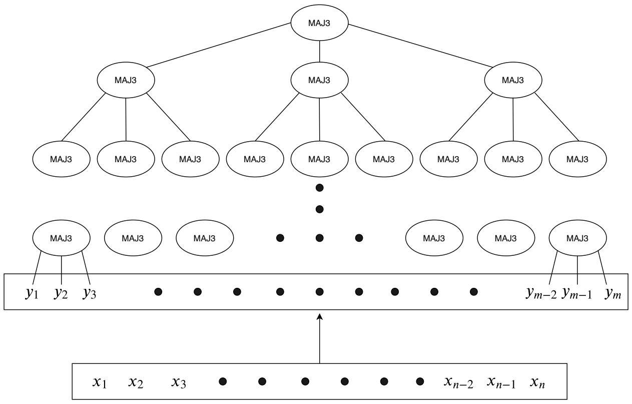


Figure 3: Majority circuit realization using MAJ3 nodes. The variables $x_i, i \leq n$ are mapped to $y_j, j \leq m$ uniformly randomly. MAJ3 tree is formed from y_j.

In this work we consider the $(n, \lfloor \frac{2n}{3} \rfloor)$ access structure and generate the matrix $\mathbf{M}$ with depth analysed using $\mathsf{p} = \frac{2}{3}$. The distribution matrix size is dependent on the computed m value rather than directly on the value n. That is to say, multiple n values may result in similar m value computed and hence will have similar distribution matrices. Since the designed circuit is a 3-ary tree, the m value chosen will be a power of 3 for any given n. Table 5 in Appendix compares the value of m needed for different n, p values using majority circuit and threshold circuits to achieve error margin $\mathsf{e} = 2^{\frac{-n}{4}}$.

4.1 Search for Distribution Matrix

We realize the threshold circuit using MAJ_3 internal nodes and compute the distribution matrix for different values of n. To generate the matrix, different random instances of assignment of y_i values of Figure 3 from x_i values are considered. A distribution matrix is taken as the matrix $\mathbf{M}$ for the access structure if any secret shared using the matrix $\mathbf{M}$ can be successfully reconstructed by any qualified subset of nodes.

We consider a $(n, \lfloor \frac{2n}{3} \rfloor)$ access structure and compute the distribution matrix $\mathbf{M}$ for different number of nodes. A random instance of mapping from literals $x_i, i \in [n]$ to literals $y_j, j \in [m]$ needs to be fixed for the computation, to do so one needs to search across the possible random instances of mapping when each y_j is assigned a uniformly sampled x_i. Since for each y_j, any of the x_i values can be assigned, the size of the assignment space is n^m, however the search space can be drastically reduced when considering the number of occurrences of each literal among x_is. Each literal x_i corresponds to the node with index i, hence in an ideal scenario, all the nodes need to occur "uniformly" among the literals y_j, that is to say, the number of occurrences/assignments of each x_i to certain y_j should be almost equal. Thus we look at only those random instances where each literal x_i occurs $\sim \frac{m}{n}$ times, so we restrict ourselves to those instance where each literal is assigned literals between $\left[\lfloor \frac{m}{n} \rfloor, \lceil \frac{m}{n} \rceil + 1\right]$, for each of the instance of random mapping, the distribution matrix is constructed and checked against all the possible threshold combinations.

For an access structure (n, t), there are $\sum_{k=t+1}^{n} \binom{n}{k}$ qualified sets that can reconstruct the secret value, however if the reconstruction is successful for all the $t+1$ element subsets, it will successful for any of the subsets with more than $t+1$ elements. Thus a distribution matrix is declared to be valid if all the $t+1$ element subsets result in correct reconstruction.

Reconstruction. When a subset $\mathcal{T}$ of nodes come together to reconstruct a secret, they first compute the vector $\boldsymbol{\lambda}_{\mathcal{T}}$ such that $\mathbf{M}_{\mathcal{T}}^{\top} \boldsymbol{\lambda}_{\mathcal{T}} = (1, 0, \cdots, 0)^{\top}$. As can be observed from the dimensions of the matrix $\mathbf{M}$, for a threshold access structure $(n, \frac{2n}{3})$, $\boldsymbol{\lambda}_{\mathcal{T}}$ is a solution for under-determined system of linear equations with solution $\{\lambda_i\} \in \{0, 1, -1\}$.

5 VERIFIABLE BBSS (V-BBSS) AND DISTRIBUTED KEY GENERATION

5.1 Verifiable BBSS

Verifiability of a secret sharing scheme is the property that lets the parties receiving the shares from a dealer verify the shares' validity. Several verifiability techniques [39, 58, 60] have been proposed for different secret sharing schemes, here we discuss the verifiability of the BBSS scheme. After generating the share elements by performing $\mathbf{s} = \mathbf{M} \cdot \boldsymbol{\rho}$, for a distribution matrix $\mathbf{M}$ and a random vector $\boldsymbol{\rho} = \{\rho_1, \rho_2, \cdots, \rho_e\} \in \mathbb{Z}_q^e$, the dealer commits to each element of the vector $\boldsymbol{\rho}$ and forwards the commitments to all the parties receiving the shares. The matrix $\mathbf{M} = m_{i,j}, i \in [d], j \in [e]$ is public and known to all the parties. We briefly sketch the different steps of the Verifiable-BBSS scheme [61]:

Share Generation. The dealer samples a random vector $\boldsymbol{\rho} = \{\rho_1, \rho_2, \cdots, \rho_e\} \in \mathbb{Z}_q^e$ and sets the element ρ_1 to the desired secret value s to be shared. For a (n, t) threshold sharing, he computes the distribution matrix $\mathbf{M}$ and generates share element vector $\mathbf{s} = \mathbf{M} \cdot \boldsymbol{\rho}, \mathbf{s} = \{s_i\}, i \in [d]$. The dealer generates a commitment vector C consisting of commitments C_i to each element of the vector $\boldsymbol{\rho}$. The element ρ_i is committed using Pedersen commitment as $C_i = g^{\rho_i} h^{\rho_i'}$ using random $\rho_i' \in \mathbb{Z}_q$. The dealer also computes the vector $\mathbf{s}' = \mathbf{M} \cdot \boldsymbol{\rho}'$ where $\boldsymbol{\rho}' = (\rho_1', \rho_2', \cdots, \rho_e')$ and $\mathbf{s}' = \{s_i'\}, i \in [d]$

The dealer forwards the share vectors $\mathbf{s}_i = \{s_j\}, \mathbf{s}_i' = \{s_j'\}, j \in T_i$ to party P_i where T_i is the set of all row indices owned by party P_i. The dealer also *broadcasts* the commitment vector C to all the parties.

Verification. Each party P_i receives the share vector $\mathbf{s}_i$ and the broadcast commitment vector $\mathbf{c}$. All the parties compute the matrix $\mathbf{M}$ corresponding to the access structure. The parties verify each of the received share elements as follows: let the i^{th} row of matrix $\mathbf{M}$ be $(m_{i1}, m_{i2}, \cdots, m_{ie})$, the party with share element s_i (and s_i') verifies the share using the following verification: $g^{s_i} h^{s_i'} = \prod_{j=1}^{e} c_j^{m_{ij}}$

$$\prod_{j=1}^{e} C_j^{m_{i,j}} = \prod_{j=1}^{e} \left(g^{\rho_j} h^{r_j}\right)^{m_{i,j}} = \prod_{j=1}^{e} \left(g^{\rho_j m_{i,j}}\right)\left(h^{r_j m_{i,j}}\right)$$
$$= g^{\sum_{j=1}^{e} \rho_j m_{i,j}} h^{\sum_{j=1}^{e} r_j m_{i,j}} = g^{s_i} h^{s_i'}$$

If the verification does not hold, the party with the share element s_i broadcasts a complaint along with the share elements (s_i, s_i'). If more than $t+1$ complaints are broadcast in the system, the dealer is deemed malicious; else the dealer responds to the complaint by broadcasting the share forwarded to the party.

5.2 Distributed Key Generation using BBSS

A distributed key generation (DKG) [40] protocol allows a set of nodes to share a secret among themselves without a trusted third party such that any qualified subset of nodes can use/reveal their shares to compute the secret. However, any subset of nodes outside the set of qualified sets has no information about the shared secret. For a (n, t)−DKG, any subset of $t + 1$ or more nodes constitutes the qualified subset. At the heart of any DKG is a verifiable secret sharing (VSS) scheme. To achieve a (n, t)-DKG protocol, we consider a (n, t)-VSS scheme; unlike a VSS scheme which requires a trusted dealer, the DKG mechanism distributes the trust among the nodes removing the requirement of a trusted party. In this work, we consider a DKG protocol resistant to f malicious nodes with the total number of nodes $n = 3f + 1$ in the network.

Using the verifiable BBSS scheme (refer Section 5.1), we obtain a DKG on the lines of the scheme by Gennaro *et al.* [40]. The protocol proceeds in two phases, in phase 1, each party P_i performs a verifiable secret sharing of a random value z_i and every party verifies the received shares using the broadcast commitments. After this, every party P_j forms the qualified set of parties Q whose shares are verified and compute its share $\mathbf{sk}_j$ by locally adding the verified shares. The computed shares correspond to shares of a random secret key $sk \in \mathbb{Z}_q$. In Phase 2, the parties of the qualified set forward the exponentiation of their shared secret z_i and a zero-knowledge proof that the forwarded Pedersen commitment in Phase 1 corresponds to the same. Every party computes the public key $pk = g^{sk}$ after verifying the zero-knowledge proofs. The complete DKG protocol based on BBSS sharing is described in Figure 4.

The proposed DKG mechanism offers the following properties:

- *Correctness:* All qualified subsets of shares provided by honest parties define the same unique secret key sk; All honest parties compute the same public key $pk = g^{sk}$ value corresponding to the secret key sk
- *Secrecy:* No information on sk can be obtained by the t−limited adversary except what can be inferred from the public information.

THEOREM 5.1. *Given a correct and secure (n, t) verifiable BBSS scheme, the DKG protocol of Figure 4 satisfies correctness and secrecy properties under the Dlog assumption (Theorem 2.1)*

All the proofs have been postponed to Appendix G owing to space constraints.

6 D-KODE PROTOCOL

By D-KODE protocol we refer to set of all algorithms for generating client keys in a distributed fashion. These algorithms include generation of shares of master key $\mathbf{k}$ at the servers using BBSS-DKG, PRF evaluation upon user input and algorithms to combine the partial evaluations to compute keys at the client. Since BBSS-DKG and PRF are run on the server, we refer to them as server-side algorithms and the algorithms for combining the partial evaluations for computing keys at the client as client-side algorithms. On the client side, we have two different versions corresponding to offline or online client. Offline client refers to the one to whose public key the payment has been made and wishes to retrieve the funds by generating the corresponding secret key. Online client computes the private key of self and public key of another client to process payment etc. For the ease of exposition, we present the verifiability of the PRF evaluation as a separate subsection. The D-KODE protocol consists of following algorithms.

6.1 Server Side Algorithms

Cryptographic Setup. $\mathsf{Setup}(\lambda, n, t)$: It takes as input the security parameter λ, the threshold t and the number of servers n. It outputs the public parameters $\mathsf{pp} := \{H(\cdot), p, q, q', u, \mathbb{G}, g, \mathrm{G}, \mathrm{g}, \mathrm{h}, \boldsymbol{M}, \psi(\cdot)\}$.

Distributed Key Generation. DKG (n, t, q, u): The servers run the BBSS-DKG mechanism among themselves using (n, t)-BBSS to generate shares of a master key $\mathbf{k} \in \mathbb{Z}_q^u$.

The BBSS-DKG mechanism presented in Section 5.2 (Figure 4) provides shares corresponding to a *single element* $sk \in \mathbb{Z}_q$ to all the servers. However, for the PRF evaluation, $F(X, \mathbf{k}) = \lfloor H(X) \cdot \mathbf{k} \rfloor_p$ introduced in Appendix A, the key $\mathbf{k}$ is a vector of length u. Hence, initially, the servers run u instances of DKG to generate shares of elements of vector in $\mathbb{Z}_q^u$. Let the share element matrix obtained by each server P_i be $\mathbf{E}_i$.

PRF evaluation. The servers run the PRF service through the ParSecretKeyEval and ParPubKeyEval algorithms to compute private key or public key shares respectively for an identity forwarded by the client.

$\mathsf{ParSecretKeyEval}(X, \mathbf{E}_i, \mathsf{pp})$: Sever P_i takes the client input string X, share matrix $\mathbf{E}_i$, the public parameters pp and returns the evaluation of the PRF as the vector z_i. The matrix $\mathbf{E}_i^\top$ is parsed into d_i columns of u length each while input X is hashed to a vector of length u using the hash $H : \{0, 1\}^* \to \mathbb{Z}_q^u$. d_i is the number of rows of matrix $\mathbf{M}$ owned by P_i. ParSecretKeyEval is shown in Algorithm 1.

Algorithm 1 ParSecretKeyEval $(X, \mathbf{E}_i, \mathsf{pp})$

Parse the matrix $\mathbf{E}_i^\top \sim \mathbb{Z}_q^{u \times d_i}$ as $[\mathbf{k}_{i1} | \mathbf{k}_{i2} | \cdots | \mathbf{k}_{id_i}]$
for $1 \le j \le d_i$ **do**
  $z_{ij} = \left\lfloor H(X) \cdot \mathbf{k}_{ij} \right\rfloor_p \in \mathbb{Z}_p$
return $\mathbf{z}_i = \{z_{i1}, z_{i2}, \cdots, z_{id_i}\} \in \mathbb{Z}_p^{d_i}$

$\mathsf{ParSig}(X, \mathbf{E}_i, \mathsf{pp}, \mathsf{msg})$: To generate a partial signature on the message msg, the server first generates the secret key share of the user by invoking $\mathsf{ParSecretKeyEval}(X, \mathbf{E}_i, \mathsf{pp})$. This secret key share is used to generate a partial signature. $\sigma'(\mathsf{msg}, X, \mathbf{E}_i) = \{\sigma(\mathsf{msg}, z_{i1}), \sigma(\mathsf{msg}, z_{i2}), \cdots, \sigma(\mathsf{msg}, z_{id_i})\}$. The partial signature vectors from the servers are threshold combined to form the final signature on the message.

$\mathsf{ParPubKeyEval}(X', \mathbf{E}_i, \mathsf{pp})$: Partial evaluation for public key generation is similar to that of the secret key except that the final vector is the exponentiated version of partial secret key evaluation. Server P_i takes the client input string X', share matrix $\mathbf{E}_i$, the public parameters pp and returns a vector $\mathbf{y}_i$. The matrix $\mathbf{E}_i^\top$ is parsed into d_i columns of u length each while input X is hashed to a vector of length u using $H : \{0, 1\}^* \to \mathbb{Z}_q^u$. d_i is the number of rows of matrix $\mathbf{M}$ owned by P_i. Each of the elements of the PRF evaluation

BBSS-DKG

pp: $\{n, t, q, p, \mathbf{M} \in \{0,1\}^{d\times e}, \psi(\cdot)\}$

Phase 1: Generating shares of $sk \in \mathbb{Z}_q$:

(1) Each party P_i performs a Verifiable BBSS of a random value $z_i \in \mathbb{Z}_q$:

(a) P_i chooses two random vectors $\boldsymbol{\rho}_i = \{\rho_{i1}, \rho_{i2}, \cdots, \rho_{ie}\}$ and $\boldsymbol{\rho}'_i = \{\rho'_{i1}, \rho'_{i2}, \cdots, \rho'_{ie}\}; \boldsymbol{\rho}_i, \boldsymbol{\rho}'_i \in \mathbb{Z}_q^e$. Sets the first element $\boldsymbol{\rho}_{i1} = z_i$.

(b) P_i computes two vectors $\mathbf{s}_i = \mathbf{M} \cdot \boldsymbol{\rho}_i$ and $\mathbf{s}'_i = \mathbf{M} \cdot \boldsymbol{\rho}'_i$, generates commitment vector $\mathbf{c}_i$ consisting of commitments to each of the elements of the vector $\boldsymbol{\rho}_i$ as $c_{il} = g^{\rho_{il}} h^{\rho'_{il}}; l \in [e]$ where g, h are generators of a multiplicative group $\mathbb{G}$. Let the computed vectors be $\mathbf{s}_i = \{s_{i1}, s_{i2}, \cdots, s_{ie}\}, \mathbf{s}'_i = \{s'_{i1}, s'_{i2}, \cdots, s'_{ie}\}$.

(c) P_i forwards the shares $\mathbf{s}_{i,j}$, a subset of the vector $\mathbf{s}_i$ to P_j consisting of share elements $s_{ik}, k \in \{T_j = \psi^{-1}(j)\}$ and it also forwards the corresponding $\mathbf{s}'_{i,j}$, a subset of the vector $\mathbf{s}'_i$ to the $P_j, j \in [n]$.

(d) P_i broadcasts its commitment vector $\mathbf{c}_i$ with elements $c_{il}, l \in [e]$ to every other party $P_j, j \in [n]$.

(e) P_j verifies the shares it received from the other parties using the specified verification procedure. s_{ik} (corresponding to the row k of the vector $\mathbf{s}_i$ of P_i) received by P_j from P_i is verified as: $g^{s_{ik}} h^{s'_{ik}} = \prod_{l=1}^{e} c_{il}^{m_{kl}} \bmod p$. (Here row k is held by $P_j, k \in T_j$).
If any verification fails, party P_j broadcasts a complaint against party P_i by broadcasting the shares (s_{ik}, s'_{ik}).

(f) On receiving a compliant against self from P_j for any row k, P_i reveals the shares by broadcasting s_{ik}, s'_{ik}.

(2) Every party maintains a set of parties *Qualified* Q, any party excluded from the set is disqualified by that particular party. Every party P_j excludes a party P_i if P_i either receives more that t complaints or the broadcasted shares after complaint do not pass the verification. At the end of the complaint and verification phase, every honest party will have the same qualified set Q.

(3) Every party P_j locally forms its shares of the secret key sk by adding element-wise, the shares of the vectors $\mathbf{s}_{i,j}$ received from every other party $P_i, i \in [n]$ i.e., each P_j computes its share as $\mathbf{sk}_j = \{\hat{s}_k | k \in T_j\} = \sum_i s_{ik}$ for each $k \in T_j$. Share of each party P_j is a vector $\mathbf{sk}_j$ of share elements with cardinality $d_j = |T_j|$.

Phase 2: Computing the public key g^{sk}:

(1) Each $P_i, i \in [n]$ broadcasts the values $A_{i1} = g^{\rho_{i1}}$ and a NIZKPoK $\boldsymbol{\pi}_i$ (Refer Appendix E for the proof) proving that the value committed $z_i = \rho_{i1}$ is same value in both A_{i1}, c_{i1} broadcast earlier to every other party $P_j, j \in [n]$.

(2) Each party verifies the broadcast NIZKPoK of every other party and anyone failing verification is disqualified and removed from Q.

(3) Finally the public key is computed as $pk = \prod_{i \in Q} g^{\rho_{i1}}$.

Figure 4: BBSS-DKG Protocol

is exponentiated resulting in a vector of elements of group $\mathbb{G}$ and of length d_i. ParPubKeyEval is shown in Algorithm 2.

Algorithm 2 ParPubKeyEval $(X', \mathbf{E}_i, \text{pp})$

```
1: Parse the matrix E_i^T ~ Z_q^{u×d_i} as [k_{i1}|k_{i2}|···|k_{id_i}]
2: for 1 ≤ j ≤ d_i do
3:     z_{ij} = ⌊H(X')·k_{ij}⌋_p ∈ Z_p
4: return y_i = {g^{z_{i1}}, g^{z_{i2}}, ···, g^{z_{id_i}}} ∈ G^{d_i}
```

6.2 Client Side Algorithms

The client computes the private key by combining the partial evaluations using the CombSecKey algorithm and computes the public key of identity X' by using the CombPubKey algorithm. The offline client after generating private key of his identity searches for the appropriate secret key - public key pair to which payment has been made.

Private key generation. CombSecKey(pp, $\{\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_{|\mathcal{T}|}\}$): Let $\mathcal{T}$ with $|\mathcal{T}| \geq t+1$ be the set of parties whose evaluations are used for reconstruction. CombSecKey() takes-in the partial evaluation vectors z_i received from the servers P_i of the set $\mathcal{T}$ and concatenates them to form $\mathbf{z} = \{\mathbf{z}_1 || \mathbf{z}_2 || \cdots || \mathbf{z}_{|\mathcal{T}|}\}$. Let the set of all the row indices of matrix $\mathbf{M}$ held by the parties in $\mathcal{T}$ be $\mathcal{R} = \bigcup_i T_i, P_i \in \mathcal{T}$. $\mathbf{z}$ is a vector of length $|\mathcal{R}|$. The private key is computed as the linear combination of the vector elements. The reconstruction coefficient vector $\lambda_{\mathcal{T}}$ is computed by solving $\mathbf{M}_{\mathcal{T}}^\top \cdot \boldsymbol{\lambda}_{\mathcal{T}} = \boldsymbol{\varepsilon}$. $\mathbf{M}_{\mathcal{T}}^\top$ is the set of all rows of matrix $\mathbf{M}$ held by the parties in $\mathcal{T}$. $\boldsymbol{\varepsilon} = \{1, 0, \cdots, 0\}$.

Online client : The online client computes the private key sk and the corresponding public key as $pk = g^{sk}$ and uses the key-pair (sk, pk) to perform different transactions as needed.

Offline client: Once the offline client computes the private key sk corresponding to his identity, he computes 4θ secret keys. θ is the total number of values combined by the parties in set $\mathcal{T}$ which is $|\mathcal{R}|$. He computes them as $[sk - 2\theta, \cdots, sk + 2\theta]$ and obtains the corresponding public keys $[g^{sk-2\theta}, \cdots, g^{sk+2\theta}]$. The public key to which funds have been sent will be in this set; he uses the corresponding secret key to transfer the funds. CombSecKey is shown in the Algorithm 3.

Algorithm 3 CombSecKey (pp, $\{z_1, z_2, \cdots, z_{|\mathcal{T}|}\}$)

Compute $\mathbf{z} = \{\mathbf{z}_1||\mathbf{z}_2|| \cdots ||\mathbf{z}_{|\mathcal{T}|}\} \in \mathbb{Z}_p^{|\mathcal{R}|}$
Compute $\boldsymbol{\lambda}_{\mathcal{T}} \in \{-1, 0, 1\}^{|\mathcal{R}|}$ such that $\mathbf{M}_{\mathcal{T}}^{\top} \cdot \boldsymbol{\lambda}_{\mathcal{T}} = \boldsymbol{\varepsilon}$
Compute $sk = \boldsymbol{\lambda}_{\mathcal{T}}^{\top} \cdot \mathbf{z} \in \mathbb{Z}_p$
if Online client **then**
  return sk
if Offline client **then**
  Compute $[sk - 2\theta, \cdots, sk + 2\theta], \theta = |\mathcal{R}|$.
  Compute public keys $\vec{pk} = [g^{sk-2\theta}, \cdots, g^{sk+2\theta}]$
  Check public keys $\vec{pk}$ and find corresponding sk'
  return sk'

Public key generation. CombPubKey(pp, $\{\mathbf{y}_1, \mathbf{y}_2, \cdots, \mathbf{y}_{|\mathcal{T}|}\}$): Let $\mathcal{T}$ with $|\mathcal{T}| \geq t + 1$ be the set of servers whose evaluations are used for reconstruction. CombPubKey takes-in the vector of partial evaluations $\mathbf{y}_i$ received from the servers P_i of the set $\mathcal{T}$ and concatenates them to form $\mathbf{y} = \{\mathbf{y}_1||\mathbf{y}_2|| \cdots ||\mathbf{y}_{|\mathcal{T}|}\}$. The set of all the row indices (of matrix $\mathbf{M}$) held by the parties in $\mathcal{T}$ is $\mathcal{R} = \bigcup_i T_i, P_i \in \mathcal{T}$. $\mathbf{y}$ is a vector of length $|\mathcal{R}|$. Compute the public key as $pk = \prod_{1 \leq j \leq |\mathcal{R}|} y_j^{\lambda_j}$, where $\mathbf{M}_{\mathcal{T}}^{\top} \cdot \boldsymbol{\lambda}_{\mathcal{T}} = \boldsymbol{\varepsilon}$, $\mathbf{M}_{\mathcal{T}}^{\top}$ is the set of all rows of matrix $\mathbf{M}$ held by the parties in $\mathcal{T}$, $\boldsymbol{\lambda}_{\mathcal{T}} = \{\lambda_j, 1 \leq j \leq |\mathcal{R}|\}$, $Y = \{y_j, 1 \leq j \leq |\mathcal{R}|\}$.

Any client can forward the public identity of another client and compute the public key from the obtained partial evaluations using CombPubKey which shown as Algorithm 4.

Algorithm 4 CombPubKey (pp, $\{\mathbf{y}_1, \mathbf{y}_2, \cdots, \mathbf{y}_{|\mathcal{T}|}\}$)

Compute $\mathbf{y} = \{\mathbf{y}_1||\mathbf{y}_2|| \cdots ||\mathbf{y}_{|\mathcal{T}|}\} \in \mathbb{G}^{|\mathcal{R}|}$
Compute $\boldsymbol{\lambda}_{\mathcal{T}} \in \{-1, 0, 1\}^{|\mathcal{R}|}$ such that $\mathbf{M}_{\mathcal{T}}^{\top} \cdot \boldsymbol{\lambda}_{\mathcal{T}} = \boldsymbol{\varepsilon}$
$\boldsymbol{\lambda}_{\mathcal{T}} = \{\lambda_j\}, \mathbf{y} = \{y_j\}, 1 \leq j \leq |\mathcal{R}|$
Compute $pk = \prod_{1 \leq j \leq |\mathcal{R}|} y_j^{\lambda_j} \in \mathbb{G}$
return pk

Using the ring-variant of the PRF. For the simplicity of exposition, we presented the whole key generation using the PRF $F(X, \mathbf{k}) = \lfloor H(\mathrm{X}) \cdot \mathbf{k} \rfloor_p \in \mathbb{Z}_p$ with a single $\mathbb{Z}_p$ element as output. However, one can consider the ring variant of the PRF where the two input vectors of computation $H(X)$ and $\mathbf{k}$ are polynomial ring elements. Then the inner product computation would be replaced by polynomial ring multiplication resulting in a ring element which can be viewed as a vector of u group elements. Thus using the ring variant of the PRF $F(X, \mathbf{k}) = \lfloor H(\mathrm{X}) \circ \mathbf{k} \rfloor_p \in \mathbb{Z}_p^u, H(X), \mathbf{k} \in R_q$, the servers can generate u keys at a time for the user.

7 DYNAMIC-COMMITTEE PROACTIVE BBSS MECHANISM

System attacks are common as flaws in the software realization of the protocols are ubiquitous. While cryptographic secrecy protects again break-ins, its effect is limited over a longer time. This is especially true in-case of a *mobile* attacker [45, 57] who can break into systems one-by-one over a long time. Proactive secret sharing (PSS) guards against these gradual attacks by combining distributed trust with periodic share renewing. When systems store keys for a long time, even when the secret information is threshold-shared, it is imperative to refresh the shares such that the adversary does not eventually gain all the information. In proactive security [32, 45, 57], the nodes modify their secret shares periodically such that the adversary's knowledge of secret information from any previous period is not useful in the next. For the D-KODE protocol, we propose proactive secret sharing for the BBSS scheme.

Adversary. We consider a computationally bounded *mobile* adversary [45] that can corrupt any server any point of time, however, the adversary can corrupt no more than t servers at any instant of time. The adversary after compromising the server has full access to the server's secret information and communication. We consider malicious corruption in which the adversary makes the server deviate arbitrarily from the protocol. The adversary has access to the complete view of the corrupted server's communication, however, he can neither inject, access or deny messages between any two non-compromised nodes nor affect the broadcast channel. The adversary corrupting the servers is removable by a reboot mechanism [32], which is handled by the system management interacting with the servers. The defined protocol provides explicit mechanism to detect malicious behaviour, we assume a reboot is triggered as soon as malicious behaviour is detected which is completed with in that epoch. The system management initializes the system by establishing server to server communication and no secret information of the protocols is available to it.

The aim of the adversary corrupting the servers is to learn the secret information or the secret key shares involved in the protocol. The user or clients interacts with the servers to obtain partial evaluations of the keys. He may try to attack the system by either predicting the server secret key or the evaluations for other clients. At the end of each refresh phase, the servers erase the old information of the previous epochs. This process is assumed reliable; when the server is compromised, the adversary does not have access to the secret information of the previous epochs. If a server is compromised in the refresh phase, the server is assumed to be compromised in both the phases adjacent to that phase.

Protocol. We propose a proactive secret sharing scheme [45] for the black box secret sharing mechanism where the size of share-elements does not increase with each refresh. The protocol proceeds in intervals of time called *epochs*, which are synchronized by the common global clock. The parties participate in a share *refresh* phase at the beginning of each epoch after which every party in the system has access to the new shares. The adversary can corrupt up-to t parties, if it is detected that a certain party is corrupted in an epoch, its shares are renewed in the *share renewal phase* phase of the next epoch, similarly if a node crashes during an epoch, its shares are reconstructed in the *reconstruction phase* of the next epoch. Share renewal and reconstruction are a part of the refresh phase of each epoch.

Without loss of generality, let (n, t) be the access structure of epoch e and (n', t') be the access structure of the epoch $e+1$ with a changing (dynamic) committee. Let the access structures of epochs $e, e+1$ correspond to the share distribution matrices $\mathbf{M}$ and $\mathbf{M}'$. Let $\mathbf{sk}_i$ be the set of share elements held by the party P_i for the

Proactive BBSS

The public parameters $pp = \{n, t, q, p, \mathbf{M}, \mathbf{M'}, \psi(\cdot), \psi(\cdot)'\}$. Each party P_i begins with an initial verified share $\mathbf{sk}_i$(and $\mathbf{sk}'_i$) consisting of elements $\hat{s}_{i,k'}$(and $\hat{s'}_{i,k'}$) $\in \mathbb{Z}_q, 0 \le k' \le |\psi^{-1}(i)|$. $\mathbf{M} \in \{0,1\}^{d \times e}, \mathbf{M'} \in \{0,1\}^{d' \times e'}$. All the honest parties begin with a commitment vector $\mathbf{v} = (v_1, v_2 \cdots v_e)$. Share renewal:

For each k' from above party P_i performs the following:

(1) Performs a Verifiable-BBSS of each of the share elements among all the parties. Samples random vectors $\boldsymbol{\rho}_i, \boldsymbol{\rho'}_i \in \mathbb{Z}_p^{e'}$ with elements $\rho_{il}, \rho'_{il}, l \in [e']$ and computes $\mathbf{s}_i = \mathbf{M'} \cdot \boldsymbol{\rho}_i$ and $\mathbf{s}'_i = \mathbf{M'} \cdot \boldsymbol{\rho'}_i$ with $\rho_{i1} = \hat{s}_{ik'}$ and $\rho'_{i1} = \hat{s'}_{ik'}$

(2) Let the share elements of $\mathbf{s}_i$ and $\mathbf{s}'_i$ be s_{il} and $s'_{il}, l \in [e']$. Forward the share elements s_{ik}, s'_{ik} to party $P_j, k \in T_j = \psi^{-1}(j)$ and commitments $c_{il} = g^{\rho_{il}} h^{s'_{il}}, l \in [e']$ to all the parties.

(3) P_i verifies the shares and the corresponding commitments received from party P_j and broadcasts a complaint against P_j if the verification fails.

(4) P_i computes the qualified set Q' as in Phase 1 of BBSS-DKG, at the end of which all honest parties compute the same set Q'.

(5) P_i computes the new share as follows: Let $\mathbf{M}'_{Q'}$ be the set of rows held by the parties in the set Q'. Each party computes the vector $\boldsymbol{\lambda}_{Q'} \in \{0,1,-1\}^{d_{Q'}}$ such that $\mathbf{M'}^{\top}_{Q'} \cdot \boldsymbol{\lambda}_{Q'} = \boldsymbol{\varepsilon}$. The new share of P_i is $\mathbf{sk}'_i = \tilde{\mathbf{s}}^{\top}_{i,Q'} \cdot \boldsymbol{\lambda}_{Q'}$, where $\tilde{\mathbf{s}}_{i,Q'}$ is the set of all share elements received by party P_i from the parties in the set Q'.

Figure 5: Proactive BBSS Scheme

epoch e. In our proactive protocol, each party re-shares every share element held by the party to all other parties of the next epoch. The Proactive BBSS scheme is presented in Figure 5.

Proactive BBSS offers the following properties [32]:

- *Robustness/Correctness*: The new shares computed at the end of the share renewal phase correspond to the original secret sk shared among the parties i.e., any qualified set of parties ($t+1$ or more) can reconstruct the secret sk.
- *Secrecy:* No information about the secret sk is obtained by the t-limited adversary in any epoch. The adversary who obtains shares of no more than t parties has no information about the secret sk in any epoch.
- *Liveness*: All honest parties complete the refresh of shares (at the beginning) in each epoch.

The proactive BBSS mechanism works mainly in two steps:

- Each party $P_i, i \in [n]$ does verified secret sharing of each of its shares $\mathbf{sk}_i$ among all the parties
- From the obtained verified shares, each party *reconstructs* their new shares $\mathbf{sk}'_i$.

Let $\mathbf{c}_i$ be the vector of commitments to the vector $\boldsymbol{\rho}_i$ by each party P_i in the previous epoch and Q be the qualified set computed during that epoch. Each party stores a vector $\mathbf{v}$ of commitments from the parties of qualified set computed during the re-sharing from the previous epoch for the verifiability of shares for the next epoch. All the honest parties update the commitment vector $\mathbf{v}$ with elements $v_l = \prod_{P_i \in Q} c_{i,\ell}^{\lambda_i}, \ell \in [e]$. When party P_i shares $\hat{s}_{ik}$ (while using $\hat{s'}_{ik}$), each party P_j checks if $g^{\hat{s}_{ik}} h^{\hat{s'}_{ik}} = \prod_k (v_k)^{m_{ik}}$ where $\mathbf{M}_Q^{\top} \boldsymbol{\lambda}_Q = \varepsilon, \boldsymbol{\lambda} = \{\lambda_k, k \in \bigcup_i T_i, P_i \in Q\}$. Let $s_{ik}, k \in T_j$ be the shares received by P_j from party $P_i \in Q'$. $\mathcal{R}' = \{\bigcup_i T_i, P_i \in Q'\}$ is the set of all rows held by Q'. P_j computes the new share element $s_k = \sum_{i \in Q'} \lambda_i s_{ik}, k \in T_j$.

Theorem 7.1. *Given a correct and secure (n,t)-verifiable BBSS scheme, the Proactive BBSS protocol of Figure 5 satisfies correctness and secrecy properties under the discrete logarithm assumption.*

Theorem 7.2. *If the $LWR_{(q,m,n)}$ assumption holds, ParSecretKeyEval$(X, \mathbf{E}, pp)$ is a pseudo-random function.*

Theorem 7.3. *If the $LWR_{(q,m,n)}$ assumption holds, CombSecKey is a (n,t)-threshold evaluation of a pseudo-random function.*

The proofs have been postponed to Appendix G.

8 PERFORMANCE ANALYSIS

We evaluate the performance of D-KODE protocol, using 10 AWS EC2 c5a.8xlarge instances spawning the nodes in the network. Our prototype Python implementation includes BBSS, BBSS-DKG, BBSS-PSS, and the corresponding reference implementations of New-JF-DKG [40] instantiated with Shamir secret sharing and replicated secret sharing (RSS). We use Charm crypto library [4] for the cryptographic implementation.

Distributed Key Generation (DKG). We implement the DKG protocols using Tendermint [13] as a broadcast channel for verifiable secret sharing. Figure 6 provides a logarithmic plot comparing the time taken to run DKG to generate shares of a 256-bit key using Shamir and 283-bit replicated (RSS) and black-box (BBSS) secret sharing schemes for up to 50 nodes. RSS is a well-known scheme (refer Appendix D for a brief description) to share secrets in $\mathbb{Z}_q$ in an additive form. The access structure for the verifiable sharing corresponds to $(n, \lfloor \frac{2n}{3} \rfloor)$ threshold in all the protocols analyzed through Figure 6.

Shamir secret sharing allocates one share element per node, while BBSS and RSS allocate share *vectors*. The vector length for RSS grows exponentially as $\binom{n-1}{t}$ for (n,t)-sharing. The share vector length for a node in BBSS is determined by the distribution matrix and the share allocation function $\psi(\cdot)$. While BBSS allocates more than one share element per user, verifying shares is efficient, involving only multiplications instead of exponentiations since the distribution matrix is a sparse binary matrix. This is reflected in the slightly lesser times recorded compared to Shamir-DKG for up to 27 nodes. The distribution matrix is of dimension 36×22 (with

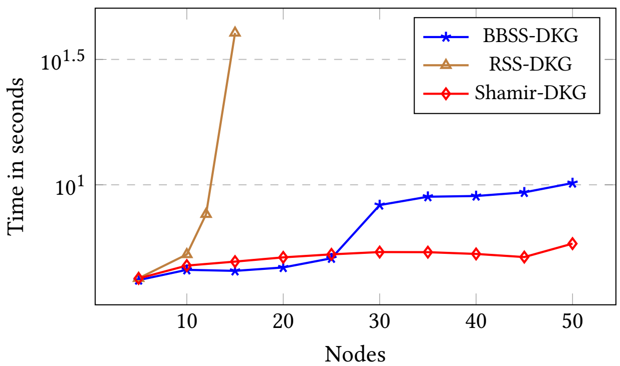


Figure 6: Time taken to perform DKG to generate shares of a 256-bit key for Shamir-DKG and 283-bit value for RSS and BBSS-DKG. The values show the mean of values across nodes for 10 runs of the protocol.

different $\psi(\cdot)$ function) when the number of nodes $n \in [4, 9]$; it is 216×130 and 1296×778 for $n \in [11, 27]$ and $n \in [28, 50]$ respectively. Beyond 28 nodes, the time to perform BBSS-DKG shows a jump due to the distribution matrix size change. Such a change in matrix occurs at 10 nodes as well; however, the change in the time taken is not too significant. While using RSS, the time taken for DKG grows exponentially owing to an exponential increase in the number of shares per node with n, the scheme becomes unviable beyond 12-15 nodes. In Shamir-DKG, since each node provides only one share element for every other node, the time taken is the lowest for higher n. Though the time taken to perform BBSS-DKG can be higher than Shamir-DKG, it is the *number of instances* of the DKG that is significantly lesser while employing D-KODE protocol when compared to the Plain-DKG [5].

Distributed PRF. D-KODE provides key-shares using PRF $F(X, \mathbf{k})$ where $\mathbf{k}$ is a vector. Each element of the vector $\mathbf{k}$ at the server is a share generated using BBSS-DKG. The parameters (LWR) for computing the PRF are chosen as following: $\mathtt{n} = 8192$, q : 283-bit, p : 256-bit. The parameter $q' > pq$ used for commitments is 571-bit with commitments on the curve `secp571r1`. The servers run 8192 instances of BBSS-DKG to generate shares for the key $\mathbf{k}$. The PRF output is a 256−bit key; The corresponding public key is computed on the `secp256k1` curve. In the case of computing the public key of another party, the servers generate the public key share (on the curve `secp256k1`) and forward it to the user. Each server takes < 200 msec to generate shares for a user per thread, for $n \in [5, 50]$ on AWS EC2 c5a.8xlarge . The servers use the BLS signature [27, 29] and the corresponding curve for public keys for the threshold signatures.

While the chosen parameters offer more than 128−bits of LWR security (for solving the LWR instance) as estimated using the LWE-estimator [19], the overall security of the systems is determined by both Discrete Logarithm and LWR parameters. Since the secret keys are 256-bit with public keys on the `secp256k1` curve, the bit-security offered by our system is 128-bits, which is similar to the Plain-DKG approach.

[5]The basic differences between Plain-DKG and D-KODE are recalled in Table 4 of the Appendix.

Table 3: Number of shares per server while using Plain-DKG and D-KODE with either RSS or BBSS for Φ keys. Number of verifiable secret sharing instances for share refreshing is same as the average number of shares stored. The shares are $\mathbb{Z}_q$ elements where q is 256-bit for Plain-DKG and 283-bit for BBSS.

No. of keys (Φ)	No. of Servers (n)	Average number of shares per server		
		Plain DKG	D-KODE	
			With RSS	With BBSS
Φ	5	Φ	32768	58982.4
	10	Φ	688,128	176,947.2
	20	Φ	22.224e+7	88,473.6
	30	Φ	82.016e+9	353,894.4
	40	Φ	66.528e+12	265,420.8
	50	Φ	27.424e+15	212,336.64

D-KODE vs Plain-DKG. D-KODE allows clients to generate private and public keys using partial share-evaluations from different servers. Plain-DKG approach is another way to provide such key shares where one instance of DKG is run *per user* to provide the shares (private or public key shares) whenever requested. In this, for every new user, the servers perform consensus on the index of pre-shared keys and offer the key shares to the user. As Shamir-DKG is efficient even for a higher number of servers as shown in Figure 6, we consider Shamir-DKG for Plain-DKG approach. We compare D-KODE with Plain-DKG as it is the only other major approach available currently in the industry (Torus[14], Sepior [12] etc).

When the servers store keys, either own or user's secret keys for a long-time, proactively refreshing the shares is inevitable. This is one of key phases where D-KODE offers an advantage. To bring this out, we compare the different numbers of shares and commitments stored at each server when using different schemes to provide the key shares in Table 3 (provided in Appendix owing to space constraints). The table compares D-KODE where the master key between servers is shared using RSS and BBSS and Plain-DKG for the different numbers of servers and clients present in the system. For share refreshing, each share value stored at the server is re-shared in the next epoch. Hence the number of shares stored at each server is the same as the number of verifiable secret sharings to be performed in the next round.

Plain-DKG stores $t + 1$ commitments for each (n, t) DKG, hence for c number of clients, stores $c \cdot (t + 1)$ commitments per server. For BBSS with distribution matrix of size $d \times e$, each server stores e commitments per shared value. Hence for 8192-element master key, stores $8192 \cdot e$ commitments. For RSS, each server forwards commitments to each of the share, hence the number of commitments is $8192 \cdot \binom{n}{t}$.

For Plain-DKG, since the number of shares is the same as the number of clients and hence linear with, increasing the share-refresh time with a higher number of clients. D-KODE uses a fixed 1024-element long master key vector shared among the servers. Only shares corresponding to the master key vector need to be refreshed at each round and do not change with the number of clients.

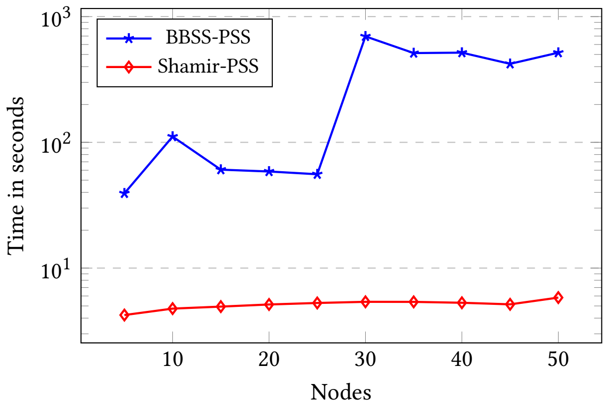


Figure 7: Time taken to perform share refreshing - proactive secret sharing (PSS) of shares corresponding to one element in the master key. These values correspond to re-sharing a total of 216 values for 10-27 nodes and 1296 283-bit values for $28-50$ node network for D-KODE with BBSS. For Shamir secret (re-)sharing, each node re-shares just one 256-bit element per key. The values show the mean of values across nodes for 10 runs of the protocol.

For D-KODE with RSS, the number of shares is constant with respect to the users but increases exponentially with the number of servers. The number of shares stored at the server when D-KODE is used with BBSS is dependent on the distribution vector. Since the actual number of share elements per server may vary depending on the share distribution function, we provide the average number of share elements per server. For the ranges $n \in [4, 9], [10, 27], [28, 50]$, the distribution matrix would be the same within each range. Hence with increasing n in those ranges, the average number of shares per server decreases. The distribution matrix would again change at $n = 82$. The distribution matrices and the different data-sets have been provided at the link https://anonymous.4open.science/repository/KODE/secretsharing/blackbox/

Figure 7 in Appendix shows the time to refresh one share through proactive secret sharing(PSS). BBSS-PSS takes longer as the number of share elements per server is higher whereas it is just one element for Shamir secret sharing while sharing a single secret value. The increase in time at $n = 10$ and $n = 28$ for BBSS-PSS is due to the change in distribution matrix size. Figure 8 shows the estimated time to refresh shares using D-KODE and Plain-DKG for increasing number of keys. We note that any parallelization applied to speed-up can be applied to both schemes. Hence, we provide an estimate of times taken by appropriately scaling the timing values obtained for re-sharing of single share value. D-KODE out-performs Plain-DKG for 94K and higher keys when the number of servers used is below 27. In the range of $28-50$ servers, D-KODE out-performs Plain-DKG from 1 million keys. D-KODE protocol also offers the non-trivial advantages of storing shares of 8192-element key vector versus millions of key-shares and the servers being essentially non-interactive except during the share-refreshing phase. D-KODE is particularly suitable for large-scale service-offering scenarios involving millions of keys.

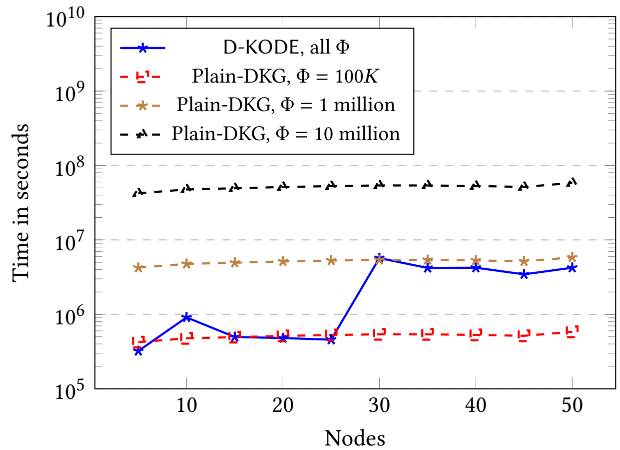


Figure 8: Estimated time to refresh shares through proactive secret sharing (PSS) for D-KODE and Plain-DKG for number of keys $\Phi = 100K$, 1million and 10million. D-KODE re-shares shares of a fixed number of 8192 values; Plain-DKG re-shares values equal to the number of keys.

9 RELATED WORK

Apart from the DKG based approaches studied in this work, firms like ZenGo [17] and Unbound [16] have proposed solutions to solve key-management problem. However, they store a key-share of the secret key on the client device, requiring explicit registration procedure. This prevents other clients from obtaining public keys of parties which have not registered yet.

The other approaches which are closer to the goals of the paper are in the domain of identity-based encryption (IBE) with a distributed private-key generator (PKG). An IBE scheme allows any party to generate a public key associated with a known identity value and employs a trusted PKG node to generated related private key. As it is possible to distribute the trust of a PKG node among a set of servers [49], it seems to directly fit both the scenarios discussed in this work. However, use of IBE presents a nuanced cryptographic challenge: the generated IBE private keys are elliptic curve group elements, while current blockchains employ ECDSA or Schnorr signatures and require private keys to be scalar from $\mathbb{Z}_p$. While theoretically mapping the elliptic curve group elements to $\mathbb{Z}_p$ is possible through hashing, performing such a hash computation in a multi-party setting is computationally expensive in practice[18, 43].

The BBSS scheme has been proposed by Cramer *et al.* [35] who provide a construction of the scheme with reconstruction coefficients in $\mathbb{Z}$. D-KODE uses the Benaloh-Leichter construction [24] in the realization of the scheme to make the reconstruction coefficients small. Another closely related work is by Damgard *et al.* [36] which proposes linear integer secret sharing (LISS) where an integer value is shared instead of a finite group element $\mathbb{Z}_p$. The work proposes to realize the distribution matrix using the mechanism proposed by Valiant [63] and Hooray [46]. A verifiable version of the LISS scheme has been proposed in [51, 61]. Unlike the LISS scheme, we require the secret to be in the group $\mathbb{Z}_p$, hence we use the BBSS scheme.

Distributed PRFs (DPRF) were studied in works like [28, 33, 54] where in [54] the authors use the PRF for a secret key distribution centre. Boneh *et al.* [28] study key homomorphic PRF for DPRF computation, Libert *et al.* [50] propose a DPRF construction secure against adaptive adversary in the standard model, however the PRF proposed requires large groups and computing expensive rounding-down functions in the multi-party setting.

Distributed Key Generation has been well studied both by the academia and industry [40, 48]. Gennaro *et al.* [40] propose a DKG mechanism that utilizes Shamir secret sharing and polynomial commitments for verifiability. DKG for networks involving $15-20$ servers has been attempted in the work [20]. Recently work by Tomescu *et al.* [62] has shown an efficient and fast DKG for large systems. The authors use multi-point evaluation of polynomials to perform efficient verifiable secret sharing and DKG. Another recent work on aggregatable DKG [44] studies DKG with more efficient transcript size and verification time. However, the focus of the authors of [44, 62] is to scale with number of servers instead of clients which we deal through the D-KODE protocol. Proactive secret sharing [45] has been employed by Coca [64] which proposes an online certificate authority with share refreshing. Zhou *et al.* [65] study a proactive secret sharing scheme for asynchronous networks using replicated secret sharing (RSS). However since the sharing is RSS which provides exponential number of shares with increasing number of servers, the scheme becomes unviable beyond $10-15$ servers.

10 CONCLUSION

In this paper we present the D-KODE protocol, an efficient solution for providing keys to parties who wish to transact among themselves and do not have access to key-setup, even when one them is offline. A set of servers with a master secret threshold-shared between them provide partial key shares as verifiable PRF evaluations to the clients who reconstruct the desired keys. We envisage a system where millions of clients avail the service and the solution scales well with the number of keys. In this paper we instantiate a distributed key generation mechanism using black-box secret sharing and propose a proactive sharing mechanism of BBSS shared keys to support the system over long periods of time. Our prototype implementation shows the scalability of our solution as the number of keys reaches 100 – 1000K depending on the number of servers.

ACKNOWLEDGEMENTS

We thank Leonard Tan and Zhen Yu of Torus for the discussions and feedback regarding key management in cryptocurrency wallets. This work has been partially supported by the National Science Foundation under grant CNS-1846316.

REFERENCES

[1] Airdrop king. https://airdropking.io/en/.
[2] Binance. https://www.binance.com/en.
[3] Chainlink. https://chain.link/.
[4] Charm: A Framework for Rapidly Prototyping Cryptosystems. https://github.com/JHUISI/charm.
[5] Cocoricos airdrops. https://cocoricos.io/airdrops.
[6] Coinbase wallet. https://wallet.coinbase.com/.
[7] Coindesk airdrop archive. https://www.coindesk.com/tag/airdrops.
[8] How many bitcoin addresses are being created in 2021. https://appdevelopermagazine.com/how-many-bitcoin-addresses-are-being-created-in-2021/.
[9] Keep network. https://keep.network/.
[10] Kraken. https://www.kraken.com/.
[11] Nist roadmap toward criteria for threshold schemes for cryptographic primitives. https://csrc.nist.gov/publications/detail/nistir/8214a/final.
[12] Sepior. https://sepior.com/.
[13] Tendermint. http://Tendermint.com.
[14] Torus. http://Tor.us.
[15] The transport layer security (tls) protocol. https://tools.ietf.org/html/rfc8446.
[16] Uboundtech. https://www.unboundtech.com/.
[17] Zengo, kzen networks. https://zengo.com/.
[18] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *Advances in Cryptology – ASIACRYPT 2016*, pages 191–219, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
[19] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. https://eprint.iacr.org/2015/046.
[20] Ian Goldberg Aniket Kate, Yizhou Huang. Distributed key generation in the wild.
[21] Abhishek Banerjee and Chris Peikert. New and improved key-homomorphic pseudorandom functions. In *Annual Cryptology Conference*, pages 353–370. Springer, 2014.
[22] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 719–737. Springer, 2012.
[23] Mihir Bellare and Phillip Rogaway. Robust computational secret sharing and a unified account of classical secret-sharing goals. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 172–184, New York, NY, USA, 2007. Association for Computing Machinery.
[24] Josh Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In Shafi Goldwasser, editor, *Advances in Cryptology — CRYPTO' 88*, pages 27–35, New York, NY, 1990. Springer New York.
[25] Josh Cohen Benaloh. Secret sharing homomorphisms: Keeping shares of a secret secret (extended abstract). In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 251–260, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.
[26] G.R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 313–317, Monval, NJ, USA, 1979. AFIPS Press.
[27] Dan Boneh, Manu Drijvers, and Gregory Neven. Short signatures from the weil pairing. In *BLS Multi-Signatures With Public-Key Aggregation*. https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html.
[28] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. Key homomorphic prfs and their applications. In *Annual Cryptology Conference*, pages 410–428. Springer, 2013.
[29] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.
[30] R. B. Boppana. Amplification of probabilistic boolean formulas. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 20–29, 1985.
[31] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.
[32] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, page 88–97, New York, NY, USA, 2002. Association for Computing Machinery.
[33] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptol.*, 18(3):219–246, 2005.
[34] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography Conference*, pages 342–362. Springer, 2005.
[35] Ronald Cramer and Serge Fehr. Optimal black-box secret sharing over arbitrary abelian groups. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 272–287, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
[36] Ivan Damgård and Rune Thorbek. Linear integer secret sharing and distributed exponentiation. In *Public Key Cryptography - PKC 2006*, pages 75–90, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
[37] Yevgeniy Dodis. Efficient construction of (distributed) verifiable random functions. In *International Workshop on Public Key Cryptography*, pages 1–17. Springer, 2003.

[38] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *International Workshop on Public Key Cryptography*, pages 416–431. Springer, 2005.
[39] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438. IEEE, 1987.
[40] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptol.*, 20(1):51–83, January 2007.
[41] Oded Goldreich. Valiant?s polynomial-size monotone formula for majority. 2011.
[42] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 241–264. 2019.
[43] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. Mpc-friendly symmetric key primitives. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 430–443, New York, NY, USA, 2016. Association for Computing Machinery.
[44] Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. Cryptology ePrint Archive, Report 2021/005, 2021. https://eprint.iacr.org/2021/005.
[45] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Annual International Cryptology Conference*, pages 339–352. Springer, 1995.
[46] Shlomo Hoory, Avner Magen, and Toniann Pitassi. Monotone circuits for the majority function. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 410–425. Springer, 2006.
[47] Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.
[48] Aniket Kate and Ian Goldberg. Distributed key generation for the internet. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 119–128. IEEE, 2009.
[49] Aniket Kate and Ian Goldberg. Distributed private-key generators for identity-based cryptography. In Juan A. Garay and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 436–453, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
[50] Benot Libert, Damien Stehl, and Radu Titiu. Adaptively secure distributed prfs from lwe. In *Theory of Cryptography Conference*, pages 391–421. Springer, 2018.
[51] Chuangui Ma and Xiaofei Ding. Proactive verifiable linear integer secret sharing scheme. In *Information and Communications Security*, pages 439–448, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
[52] Alfred J Menezes, Jonathan Katz, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.
[53] M. Naor and O. Reingold. Synthesizers and their application to the parallel construction of pseudo-random functions. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 170–181, 1995.
[54] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and kdcs. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 327–346. Springer, 1999.
[55] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *J. ACM*, 51(2):231–262, 2004.
[56] Jesper Buus Nielsen. A threshold pseudorandom function construction and its applications. In *Annual International Cryptology Conference*, pages 401–416. Springer, 2002.
[57] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 51–59, 1991.
[58] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85, 1989.
[59] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.
[60] Markus Stadler. Publicly verifiable secret sharing. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 190–199, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
[61] Rune Thorbek. Proactive linear integer secret sharing. 2009.
[62] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. Towards scalable threshold cryptosystems. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.
[63] Leslie G. Valiant. Short monotone formulae for the majority function. *Journal of Algorithms*, 5(3):363–366, 1984.
[64] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. Coca: A secure distributed online certification authority. *ACM Trans. Comput. Syst.*, 20(4):329–368, 2002.
[65] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. Apss: Proactive secret sharing in asynchronous systems. *ACM Trans. Inf. Syst. Secur.*, 8(3):259–286, 2005.

A PSEUDO RANDOM FUNCTION

A pseudo-random function (PRF) family [22, 42, 53] is a set of keyed functions with a common domain and range, such that no efficient algorithm can distinguish between a randomly chosen function from the PRF family and a random oracle. A key homomorphic PRF [21, 28] is a PRF that is homomorphic with respect to the key-input of the function. It can be used as a distributed PRF [33, 54, 56] when the key is shared among the servers. With an almost-key homomorphic PRF, the computed evaluation from the shares may differ (may not be equal) from the evaluation of the PRF.

In this work, we use an almost-homomorphic PRF which is based on the Learning-with-rounding (LWR) assumption (Definition 2.2).

Employed PRF [28]. Given a hash function $H : \mathcal{X} \to \mathbb{Z}_q^u$, a key vector $\mathbf{k} \in \mathbb{Z}_q^u$ and with $p < q$, the PRF evaluation $F : \mathcal{X} \times \mathbb{Z}_q^u \to \mathbb{Z}_p$ of the string X is $F(X, \mathbf{k}) = \lfloor H(X) \cdot \mathbf{k} \rfloor_p \in \mathbb{Z}_p$. That $F(\cdot, \cdot)$ is a PRF in the random oracle model follows directly from the LWR assumption discussed above, where $H(X)$ corresponds to matrix $\mathbf{A}$ with a single row ($m = 1$) and the secret key $\mathbf{k}$ to vector $\mathbf{s}$.

In this work, we use the above PRF in the distributed setting, with a key $\mathbf{k}$ shared among them and among n servers and each server locally computing a (partial) evaluation of the PRF on some input X non-interactively. Any party wishing to compute the PRF $F(X, \mathbf{k})$ inputs X to the servers and obtains evaluations from the each of them to reconstruct $F(X, \mathbf{k})$.

Table 4: Comparison of Plain-DKG and D-KODE with BBSS for n servers and Φ keys and a constant k. Refer Table 3 for the exact value of k for different n.

	Plain-DKG	D-KODE (with BBSS)
Key-generation	DKG and Consensus for every new-id	Non-interactive PRF evaluation
Storage - shares	Φ	k
Communication complexity (PSS)	$\Phi \cdot O(n^3)$	$k \cdot O(n^3)$
Security assumptions	DLog	DLog and LWR
Preferable for number of keys	$100 - 1000$K	$>100 - 1000$K

B MONTONE BOOLEAN FORUMULA FOR MAJORITY FUNCTION

Majority function [63] of n variables with values in $\{0, 1\}$ is defined as taking the value 1 if at least $n/2$ number of variables are 1 and 0 otherwise. Let $\{x_i\}_{i=1}^n$ be the n variables over which Majority function $Maj(\cdot)$ is being computed, then

$$Maj(x_1, x_2, \cdots, x_n) = \begin{cases} 1 & \text{if } \sum_i x_i \geq \frac{n}{2};\ x_i \in \{0, 1\} \\ 0 & \text{if otherwise} \end{cases}$$

While majority function of n variables can be realized using non-monotone circuits of size $O(\log n)$, monotonicity places restrictions on the circuit that the circuit should only be realized using AND and OR gates (but not NOT) gates. Valiant [63] first proved that a polynomial size *monotone* circuit is realizable for majority circuit and provided a construction of size $O(n^{5.3})$. Subsequent works like one by Hoory [46] discuss majority circuits and realize threshold structures using majority circuit. Boppanna [30] showed that $O(t^{4.3}n)$ is the optimal upper bound on the majority circuit over n variables for a threshold t. Hooray [46] further improved the size of the circuite to $O(n^{1+\sqrt{2}})$ while keeping the circuit depth at $O(\log n)$. Goldreich [41] provided an exposition of Valiant's approach to the majority circuit construction, a probabilistic proof while using a different probability amplifier (majority-3) than the one used by Valiant.

We briefly explain the construction provided in [41]:
Let the n variables be $x_i \in \{0,1\}$, $i \in [n]$. Generate m random variables $y_j, j \in [m]$ by uniform randomly sampling an index among $[n]$ and assigning the correspding x_i value to each y_j sequentially. When $\Pr(z_i = 1) = p$ for each $i \in [3]$, the probability that the majority function is 1 is given by $\Pr(MAJ_3(z_1, z_2, z_3)) = 1$ is $3(1-p)p^2+p^3$. If $p = 0.5+\epsilon, \epsilon \leq \epsilon_0 < 0.5$, then $p' \geq 0.5+(1.5-2\epsilon_0^2)\epsilon$. Thus the bias of ϵ is increased by the factor $(1.5 - 2\epsilon_0^2)$ for each level of the tree. When the number of ones in the initial set of variables x_i is $\frac{n}{2} + 1$, the bias of the variables y_i at the lowest level of the tree would be $\frac{1}{n}$. This bias is increased in three steps: First the bias is brought to a constant ($< \frac{1}{2}$) using ℓ_1 layers of the tree, then that constant is increased further to be close to 1 using ℓ_2 layers, finally the probability of majority function being 1 when there is majority in the initial value is taken arbitrarily close to 1, in other words, the probability of function returning 0 when there is majority is made negligibly small $< 2^{-n}$ in another ℓ_3 layers of the circuit. When using majority circuit, using $p = 0.5$ for a given n, when MAJ_3 nodes are used as probability amplifiers, this would result in a circuit depth of $\ell_1 + \ell_2 + \ell_3 \sim 2.71 \log n$. When MAJ_3 is expanded using fan-in 2 gates, we have a circuit implemented using only gates with fan-in 2. This would result in a total circuit size of $O(n^{5.3})$.

C BOOLEAN FORMULA AND DISTRIBUTION MATRIX

The circuit is represented as a boolean formula by expanding MAJ_3 (z_1, z_2, z_3) as $(z_1 \wedge z_2) \vee (z_2 \wedge z_3) \wedge (z_1 \vee z_3)$, resulting in a monotone boolean formula computing majority/threshold function. This formula is then used to compute the distribution matrix of the linear integer secret sharing scheme (LISS). The Benolah-Leichter (BL) [24] construction of converting a monotone boolean formula is briefly recollected here.

Consider Boolean functions $f_{OR} = f_1 \vee f_2$ and $f_{AND} = f_1 \wedge f_2$ where f_1, f_2 are either Boolean functions or literals. Let M_a and M_b are share distribution matrices of f_1 and f_2 respectively. The share distribution matrices of f_{OR}, f_{AND} are computed as M_{OR}, M_{AND} as shown in Figure 9, where C_a is the first column of matrix M_a and R_a is the rest of the matrix except the first column of matrix M_a. Similarly C_b, R_b are the first column of matrix M_b and the rest of the matrix except the first column of matrix M_b respectively. If the function contains only one literal, it is taken just as column i.e., for any literal $f_1 = x_i$, the matrix is just $[1]$ with $C_a = 1$ and no R_a.

$$\boldsymbol{M_{AND}} = \begin{bmatrix} C_a & C_a & R_a & 0 \\ 0 & C_b & 0 & R_b \end{bmatrix}$$

$$\boldsymbol{M_{OR}} = \begin{bmatrix} C_a & R_a & 0 \\ C_b & 0 & R_b \end{bmatrix}$$

Figure 9: Share distribution matrix for OR and AND functions

Table 5: m values when using majority and threshold circuits for different n values for p $= 0.5, 0.66$, e $= 2^{-\frac{n}{4}}$

n	Majority Circuit		Threshold Circuit	
	p = 0.5	p = 0.66	p = 0.5	p = 0.66
5	9	81	9	9
10	81	2187	81	27
20	2187	59049	2187	27
30	19683	531441	19683	81

D REPLICATED SECRET SHARING

Replicated secret sharing [34] for a monotone access structure Γ and its maximal unqualified sets $\mathcal{T}$, the shares of secret $s \in \mathbb{Z}_q$ are generated as follows: the dealer first generates $|\mathcal{T}|$ number of additive shares of s, each labelled by a unique set in $\mathcal{T}$. Let the shares be $\{r_T \in \mathbb{Z}_q, T \in \mathcal{T}\}$, each player P_i is given the vector of shares r_T such that $i \notin T$. Parties of every maximal unqualified set $T \in \mathcal{T}$ jointly do not have access to exactly one share element r_T. Parties of every qualified set jointly own all the share elements and thus additively reconstruct the secret s. For a (n, t) threshold access structure, each party is given $\binom{n}{t}$ share elements.

E ZERO-KNOWLEDGE PROOF OF EQUALITY OF COMMITTED VALUE

The distributed key generation protocol in the Figure 4 involves a zero knowledge proof of equality of values committed by Pedersen commitment and discrete log commitment. Here we reproduce the non-interactive zero knowledge proof of knowledge NIZKPoK [20]: given a discrete log commitment (DLog) commitment of value s as $C_1 = g^s$ and a Pedersen commitment of the same value s as $C_2 = g^s h^r$ for $g, h \in \mathbb{G}$ and $s, r \in \mathbb{Z}_p$, the prover proves the knowledge of (s, r) for the given (C_1, C_2) using the proof we denote by π. It is generated using the following steps: The prover $\mathcal{P}$ does the following: (i) Picks values $v_1, v_2 \xleftarrow{\$} \mathbb{Z}_p$ and computes $(V_1, V_2) = (g^{v_1}, h^{v_2})$ (ii) Computes the hash $c = H(g, h, C_1, C_2, V_1, V_2)$ where $(C_1, C_2) = (g^s, g^s h^r)$ and $H : \mathbb{G} \to \mathbb{Z}_p$ (iii) Computes values $(u_1, u_2) = (v_1 - cs, v_2 - cr)$ (iv) Sends (c, u_1, u_2) as proof π along with (C_1, C_2)

The verifier $\mathcal{V}$ with the values $(g, h, C_1, C_2, c, u_1, u_2)$ performs the following check: (i) Computes: $(V_1', V_2') = \left(g^{u_1} C_1^c, h^{u_2} \left(\frac{C_2}{C_1}\right)^c\right)$

(ii) Computes $c' = H(g, h, C_1, C_2, V'_1, V'_2)$. (iii) Accepts the proof if $c = c'$ else rejects.

E.1 Equality of exponent with different bases

To prove equality of exponent in discrete logarithm commitment with different bases $g \in \mathbb{G}, \mathrm{g} \in \mathrm{G}$, given $C_1 = g^s$ and $C_2 = \mathrm{g}^s$, the prover $\mathcal{P}$ does the following: (i) Picks values $v \xleftarrow{\$} \mathbb{Z}_p$ and computes $(V_1, V_2) = (g^v, \mathrm{g}^v)$ (ii) Computes the hash $c = H(g, \mathrm{g}, C_1, C_2, V_1, V_2)$ (iii) Computes $u = v - cs$ (iv) Sends (c, u) as proof along π_{Eq} with (C_1, C_2)

The verifier $\mathcal{V}$ takes the values $(g, \mathrm{g}, C_1, C_2, c, u)$ and computes the following (i) $(V'_1, V'_2) = (g^u C_1^c, \mathrm{g}^u C_2^c)$ (ii) $c' = H(g, \mathrm{g}, C_1, C_2, V'_1, V'_2)$. (iii) Accepts the proof if $c = c'$ else rejects.

F VERIFYING THE EVALUATION OF THE PRF

While the clients obtain shares as the PRF evaluations presented in Section 6.1, it is imperative for the clients to verify if the values received were generated correctly. The servers after evaluating the PRF, forward a commitment and a zero-knowledge proof proving that the values have been computed according to the protocol. For ease of exposition, we present here the verifiability for *one* PRF evaluation.

The PRF function employed by D-KODE protocol is $F(X, \mathbf{k}) = \left\lfloor H(X) \cdot \mathbf{k} \right\rfloor_p \in \mathbb{Z}_p$ with $H : \mathcal{X} \to \mathbb{Z}_q^u$, $\mathbf{k} \in \mathbb{Z}_q^u$, $F : \mathcal{X} \times \mathbb{Z}_q^u \to \mathbb{Z}_p$ and $p < q$. Let $\mathbf{k} = \{\alpha_1, \alpha_2, \cdots, \alpha_u\}$.

Verification of the private key evaluation. Let $z = F(X, \mathbf{k})$ for $\mathbf{k}$ defined as above. To compute z, the servers compute the inner product $w = (H(X) \cdot \mathbf{k}) \in \mathbb{Z}_q$ and perform the operation $z = \lfloor w \rfloor_p \in \mathbb{Z}_p$. Hence we have, $z = \left\lfloor w \cdot \frac{p}{q} \right\rfloor \implies pw = zq + r$ where the value $r < q$. To provide verifiability, it is enough for the server to prove that the above equation has been evaluated correctly and that the value $r < q$. The server uses commitments and zero-knowledge range proof to do the same.

Server computation. For a key $\mathbf{k} = \{\alpha_1, \alpha_2, \cdots, \alpha_u\}$, and a random $\mathbf{k}' = \{\beta_1, \beta_2, \cdots, \beta_n\}$, the server initially publishes the commitments $c_i = \mathrm{g}^{\alpha_i} \mathrm{h}^{\beta_i}, i \in [u]$, $\mathrm{g}, \mathrm{h} \in \mathrm{G}$ are generators of multiplicative group of order $\tau > pq$.

For proving the correct evaluation of $z = F(X, \mathbf{k})$, the server computes $z' = F(X, \mathbf{k}')$ and $r = pw - qz \bmod \tau$, $r' = pw' - qz' \bmod \tau$; forwards the values $c = \mathrm{g}^r \mathrm{h}^{r'}$ and $z' = F(X, \mathbf{k}')$ The server also computes and forwards zero-knowledge range proof [31] $\pi_r, \pi_{\mathfrak{k}}$ proving that $r < q, \mathfrak{k} < u \cdot q$ such that $w + \mathfrak{k}q = \sum_{i=1}^{u} \alpha_i h_i$. Similarly, he computes $\mathfrak{k}'$.

Thus when evaluating the PRF for an input X, the server replies with the following: $\{z, z', \mathrm{g}^r \mathrm{h}^{r'}, \mathrm{g}^{\mathfrak{k}} \mathrm{h}^{\mathfrak{k}'}, \pi_r, \pi_{\mathfrak{k}}\}$. Note that c_i values are available to the client before the PRF evaluation.

Client side computation. Using the received values and the initially published $c_i = \mathrm{g}^{\alpha_i} \mathrm{h}^{\beta_i}$ values, the client computes

$$\mathrm{g}^w \mathrm{h}^{w'} = \mathrm{g}^{-q\mathfrak{k}} \mathrm{h}^{-q\mathfrak{k}'} \prod_{i=1}^{u} (\mathrm{g}^{\alpha_i} \mathrm{h}^{\beta_i})^{h_i}$$

To verify the PRF value z, after verifying the range proof π_r, the client verifies

$$\mathrm{g}^{pw} \mathrm{h}^{pw'} = \mathrm{g}^{qz} \mathrm{h}^{qz'} \cdot \mathrm{g}^r \mathrm{h}^{r'}$$

F.1 Verification of the public key evaluation

Previously for the secret key evaluation corresponding to identity X, the server computed and forwarded the value $z = F(X, \mathbf{k})$. However, for public key evaluation, the server forwards g^z, for $g \in \mathbb{G}$ a generator of a multiplicative group of order p.

Similar to the procedure for PRF verification above, the server forwards $\mathrm{g}^r \mathrm{h}^{r'}$ such that $pw = zq + r; pw' = z'q + r'$ and $\pi_r, \pi_{\mathfrak{k}}$ proving that $r < q$ and $\mathfrak{k} < u$ such that $w + \mathfrak{k}q = \sum_{i=1}^{u} \alpha_i h_i \bmod \tau$. Similarly, he also computes $\mathfrak{k}'$. However, instead of values z, z', the server forwards g^z and $\mathrm{g}^z \mathrm{h}^{z'}$ where $\mathrm{g}, \mathrm{h} \in \mathrm{G}$ are generators of multiplicative group of order $\tau > pq$.

Additionally, the server sends a zero-knowledge proof of equality of exponents $\pi_{\mathsf{Equ}}(g^z, \mathrm{g}^z \mathrm{h}^{z'})$ proving that the value z in both the exponents $(g^z, \mathrm{g}^z \mathrm{h}^{z'})$ is equal (Refer Section E for the zero knowledge proof used). Thus the server forwards the values

$$\{g^z, \mathrm{g}^z \mathrm{h}^{z'}, \mathrm{g}^r \mathrm{h}^{r'}, \mathrm{g}^{\mathfrak{k}} \mathrm{h}^{\mathfrak{k}'}, \pi_r, \pi_{\mathsf{Equ}}(g^z, \mathrm{g}^z \mathrm{h}^{z'})\}$$

After verifying the zero knowledge proofs, the client computes $\mathrm{g}^w \mathrm{h}^{w'}$ as before and verifies

$$\mathrm{g}^{pw} \mathrm{h}^{pw'} = \mathrm{g}^{qz} \mathrm{h}^{qz'} \cdot \mathrm{g}^r \mathrm{h}^{r'}$$

G SECURITY ANALYSIS

G.1 Correctness and secrecy of BBSS-DKG

Theorem 7.1. *Given a correct and secure (n, t)-verifiable BBSS scheme, the Proactive BBSS protocol of Figure 5 satisfies correctness and secrecy properties under the discrete logarithm assumption.*

Proof. **Correctness.** In Phase 1 of the BBSS-DKG protocol from Figure 4, all honest parties compute the same qualified set Q as the complaint and disqualification information is broadcast to all parties. Any party $P_i \in Q$, which shared its value z_i successfully and any set $\mathcal{T}$ of $t + 1$ or more honest parties can reconstruct the secret key value, owing to the threshold structure of the BBSS performed. Let $\mathcal{R} = \bigcup_i T_i, i \in \mathcal{T}$ be the set of all row indices of $\boldsymbol{M}$ held by the parties of $\mathcal{T}$. Each $z_i = \sum_{k \in \mathcal{R}} s_{ik} \cdot \lambda_k$ $\boldsymbol{\lambda}_{\mathcal{T}} = \{\lambda_k, k \in \mathcal{R}\}$ such that $\mathbf{M}_{\mathcal{T}}^\top \cdot \boldsymbol{\lambda}_{\mathcal{T}} = \boldsymbol{\varepsilon}$ and $z_i = \mathbf{s}_{\mathcal{T}}^\top \cdot \boldsymbol{\lambda}_{\mathcal{T}}$, where $\mathbf{s}_{\mathcal{T}}$ is the vector of all share elements held by all the parties in $\mathcal{T}$. Every honest party computes its share vector $\mathbf{sk}_j = \{\hat{s}_k | \hat{s}_k = \sum_{i \in Q} s_{ik}, k \in T_j\}$ element-wise for each k. Thus we have,

$$sk = \sum_{i \in Q} z_i = \sum_{i \in Q} \Big(\sum_{k \in \mathcal{R}} s_{ik} \cdot \lambda_k \Big)$$

$$\implies sk = \sum_{k \in \mathcal{R}} \lambda_k \cdot \Big(\sum_{i \in Q} s_{ik} \Big) = \sum_{k \in \mathcal{R}} \lambda_k \cdot \hat{s}_k$$

This holds for any set qualified set $\mathcal{T}$ (and hence the corresponding set of rows $\mathcal{R}$), thus giving a unique sk for all such sets with $t + 1$ or more parties.

Simulator $\mathcal{S}$

Let $C = \{P_i, i \in \{1, \cdots, t'\}\}$ denote the parties controlled by the adversary and $\mathcal{H} = \{P_j, j \in \{t'+1, \cdots, n\}\}$ denote the set of honest parties in the protocol. $t' \leq t$. $\mathcal{S}$takes the public key y as input.

(1) The simulator $\mathcal{S}$ performs all the steps in the Phase 1 of the BBSS-DKG on behalf of the parties of set $\mathcal{H}$ including generating and forwarding shares and commitments, verifications of the received shares and handling all communications with the corrupted parties such that the following hold:
  (a) The values $\boldsymbol{\rho}_i, \boldsymbol{\rho}'_i$ for $P_i \in \mathcal{H}$ are chosen at random by $\mathcal{S}$.
  (b) The set Q is well defined with $\mathcal{H} \subset Q$
  (c) The adversary's view consists of $(\boldsymbol{\rho}_j, \boldsymbol{\rho}'_j)$ for $P_j \in C$, shares $(\mathbf{s}_{i,j}, \mathbf{s}'_{i,j})$ for $P_i \in Q$ and $P_j \in C$ and commitments $C_{ik}, P_i \in Q, k \in [t]$
  (d) $\mathcal{S}$ has all shares and commitments of the parties in Q. For $j \in Q \backslash \mathcal{H}$, $\mathcal{S}$ has enough valid shares to reconstruct the vector $\boldsymbol{\rho}_j, \boldsymbol{\rho}'_j$.

(2) Perform:
  (a) Compute $A_{il}, l \in [e] = g^{\rho_{il}}$ for $i \in Q \backslash n, l \in [e]$
  (b) Set $A^*_{n0} = y \prod_{i \in Q \backslash n} (A_{i0})^{-1}$ and $\mathbf{s}^*_{nk} = \mathbf{s}_{nk} = \{s_{nk}, k \in T_n\}$ where $s_{nl}, l \in [e]$ is an element of the vector $M \cdot \boldsymbol{\rho}_n$ item Broadcast the values A_{il} for $i \in \mathcal{H} \backslash n$ and A^*_{nl} with $l \in [e]$ along with the corresponding NIZKPoK $\boldsymbol{\pi}_i$

Figure 10: Simulator for BBSS-DKG

Also, each share element $\hat{s}_k, k \in T_j$ of a party P_j, can be computed and verified from the publicly available values $g^{s_{ik}}$.

$$g^{\hat{s}_k} = g^{\sum_{i \in Q} s_{ik}} = \prod_{i \in Q} g^{s_{ik}} = \prod_{i \in Q} \Big(\prod_{l=1}^{e} A_{il}^{m_{kl}} \Big)$$

which is available from Phase 2 of the protocol of Figure 4. Thus each share (and share element) can be verified for correctness at the time of reconstruction.

The public key $pk = \prod_{i \in Q} g^{\rho_{i1}}$ is computed from values broadcast in the protocol, hence the value can be obtained by all the honest parties. It remains to be shown that $pk = g^{sk}$ such that $sk = \sum_{i \in Q} z_i$. For the parties against whom a complaint is generated, the value z_i is reconstructed publicly. For the other parties against whom there was no complaint, all their values $A_{il}, l \in [e]$ have been verified using the verification step in Phase 2 of the protocol. Since all such parties constitute the qualified set Q which is computed by all the honest parties, the value $A_{i1} = g^{\rho_{il}} = g^{z_i}$. The value pk is computed by honest parties as $pk = \prod_{i \in Q} g^{z_i} = g^{\sum_{i \in Q} z_i} = g^{sk}$. Hence all the honest parties compute the same public key pk corresponding to sk. Also since the qualified set of parties Q computed in the phase 1 of the protocol consists of at least one honest party who shares the value z_i which is chosen randomly, the secret key $sk = \sum_{i \in Q} z_i$ is uniformly random.

Secrecy. We provide a simulator $\mathcal{S}$ in Figure 10 on the lines of [20, 40] which simulates the adversary view of the BBSS-DKG protocol of Figure 4. With out loss of generality we assume that the set of parties $C = \{P_1, \cdots, P_{t'}\}$ are corrupted and set of rest of the parties $\mathcal{H} = \{P_{t'+1}, \cdots, P_n\}$ are honest. The simulator controls all the honest parties $\mathcal{H}$ and performs all computations and communications with the corrupt parties on behalf of them.

The simulator follows the Phase 1 of the protocol as shown in Figure 4 and generates share vectors $\mathbf{s}_{i,j}$ using random $\boldsymbol{\rho}_i$ for $P_i \in \mathcal{H}, P_j \in C$. Similarly it generates and forwards the vectors $\mathbf{s}'_{i,j}$ using random $\boldsymbol{\rho}'_i$. $\mathcal{S}$follows the protocol including the computation of qualified set Q. However, in the second phase of the protocol, it computes and broadcasts all the $A_{i,l}$ for all the honest parties except one party P_n. For the party P_n it sets the secret value $A_{i,0}$ such that the public key obtained as $\prod_{i \in Q} A_{il}, l \in [e]$ is the desired value y. The simulator $\mathcal{S}$ will be able to reconstruct the vector $\boldsymbol{\rho}_k$ for any party P_k which is present in the qualified set Q but not in the set $\mathcal{H}$.

Whenever a valid complaint is broadcast from any party controlled by adversary, $\mathcal{S}$ constructs the secret value and opens it.

□

G.2 Correctness and Secrecy of Proactive secret sharing

Correctness. Let $(n, t), (n', t')$ be access structures in the epochs e and $e+1$. Without loss of generality let $\boldsymbol{sk}_i, i \in [n]$ be shares of secret key sk of the n parties in epoch e and $\boldsymbol{sk}'_i, i \in [n']$ be shares of the n' parties in epoch $e+1$. We need to show that any set of $t'+1$ or more parties in epoch $e+1$ reconstruct the secret key sk.

For epoch e, the share elements held by parties in qualified set Q are $\hat{s}_k, k \in \mathcal{R} = \{\bigcup_i T_i, P_i \in Q\}$. $\mathcal{R}$ is the set of all rows held by the parties in Q. We know, $sk = \sum_{k \in \mathcal{R}} \lambda_k \hat{s}_k$

However, each share element $\hat{s}_k$ is verifiable secret shared in the next epoch $e+1$. Thus any qualified set Q' of $t'+1$ parties can construct the share element $\hat{s}_k$. Let $\mathcal{R}'$ be the rows held by the parties in Q'. Then,

$$\begin{aligned} sk &= \sum_{i \in \mathcal{R}} \lambda_i \hat{s}_i = \sum_{i \in \mathcal{R}} \lambda_i \Big(\sum_{j \in \mathcal{R}'} \lambda_j s_{ij} \Big) \\ &= \sum_{j \in \mathcal{R}'} \lambda_j \Big(\sum_{i \in \mathcal{R}} \lambda_i s_{ij} \Big) = \sum_{j \in \mathcal{R}'} \lambda_j s_j = sk \end{aligned}$$

Secrecy. The secrecy of the secret in each phase follows from the security properties of Verifiable BBSS scheme. Let $\mathcal{B}, \mathcal{B}', |\mathcal{B}|, \mathcal{B}'| < t$ be the set of servers corrupted in an epoch e and $e+1$. W.l.o.g let $\mathcal{B} \cap \mathcal{B}' = \phi$, from the correctness principle above, we know that any $t'+1$ or more parties can construct the secret key in the epoch $e+1$. From the security of the BBSS scheme we know what no set of t' or less number of parties has any information about the secret, hence maintaining the secrecy property.

G.3 Security of PRF evaluations

Here we argue the security of the ParSecretKeyEval and ParPubKeyEval by providing a reduction to LWR problem instance.

Theorem 7.2. *If the $LWR_{(q,m,n)}$ assumption holds, the function ParSecretKeyEval$(X, \mathbf{E}, pp)$ is pseudo-random.*

Proof. Let ParSecretKeyEval$(X, \mathbf{E}, \text{pp})$ be $f_{\mathbf{E}}(X)$, we show that $f_{\mathbf{E}}$ is a family of pseudo-random functions.

Let $\mathcal{D}$ be an efficient algorithm that gets the value of $f_{\mathbf{E}}$ on $\ell - 1$ uniformly chosen inputs $X_1, X_2, \cdots, X_{\ell-1}$ and distinguishes $f_{\mathbf{E}}(X_\ell)$ from random with a non-negligible advantage ϵ. We construct an algorithm $\mathcal{A}$ that breaks the LWR assumption:

On input $(\boldsymbol{A}, \lfloor \boldsymbol{As} \rfloor_p)$ where $\boldsymbol{A} \sim U(\mathbb{Z}_q^{\mathsf{m}\times\mathsf{n}}), \boldsymbol{s} \sim U(\mathbb{Z}_q^{\mathsf{n}})$. $\mathcal{A}$ parses the matrix $\boldsymbol{A}$ as rows $\boldsymbol{a}_1, \boldsymbol{a}_2, \cdots, \boldsymbol{a}_\mathsf{m}$ and vector $\lfloor \boldsymbol{As} \rfloor_p$ as $\boldsymbol{z'}_1, \boldsymbol{z'}_2, \cdots, \boldsymbol{z'}_\mathsf{m}$. For each $\boldsymbol{z'}_i, i \leq \mathsf{m}$, sample $d-1$ uniformly random values $s_{i,2}, s_{i,3}, \cdots s_{i,d} \in \mathbb{Z}_p$. Let $z_{i,j} = \boldsymbol{a}_i \cdot s_{i,j}$ for $i \leq \mathsf{m}; 2 \leq j \leq d$. Now $\mathcal{A}$ invokes m instances of algorithm $\mathcal{D}_i$ each with the $\ell - 1$ pairs of values $\{\langle H(X_j), f_{\mathbf{E}}(X_j)\rangle\}_{j=1}^{\ell-1}$ and a pair $\langle a_i, [\boldsymbol{z'}_i, z_{i,2}, z_{i,3}, \cdots, z_{i,d}]\rangle$ for $i \leq \mathsf{m}$. $\mathcal{D}_i$ distinguishes $[\boldsymbol{z'}_i, z_{i,2}, z_{i,3}, \cdots, z_{i,d}]$ from a uniformly random vector with advantage ϵ. Algorithm $\mathcal{A}$ distinguishes the LWR instance from a uniformly random vector $U(\mathbb{Z}_q^d)$ with an advantage at-least ϵ. □

Theorem 7.3. *If the $LWR_{q,m,n}$ assumption holds, CombSecKey is a (n, t)-threshold evaluation of a pseudo-random function.*

Proof. Let $\mathcal{D}'$ be an efficient algorithm that differentiates an evaluation of CombSecKey from a uniformly random vector with a non-negligible advantage ϵ after $\ell - 1$ queries. It takes the vectors $[z_1, z_2, \cdots, z_n]$, computes $\lambda_i \cdot z_i$ such that the elements of the vector $\lambda_i \in \{-1, 0, 1\}$ and differentiates the resultant vector $\boldsymbol{sk}$ from the uniform vector $U(\mathbb{Z}_q^n)$ with an advantage ϵ.

We first consider the case when all the n servers are honest and then consider the case when t of them are corrupt. We build an algorithm $\mathcal{A}'$ with uses $\mathcal{D}'$ to solve the LWR instance. On input $(\boldsymbol{A}, \lfloor \boldsymbol{As} \rfloor_p)$ where $\boldsymbol{A} \sim U(\mathbb{Z}_q^{\mathsf{m}\times\mathsf{n}}), \boldsymbol{s} \sim U(\mathbb{Z}_q^n)$. $\mathcal{A}$ parses the matrix $\boldsymbol{A}$ as rows $\boldsymbol{a}_1, \boldsymbol{a}_2, \cdots, \boldsymbol{a}_\mathsf{m}$ and vector $\lfloor \boldsymbol{As} \rfloor_p$ as $z'_1, z'_2, \cdots, z'_\mathsf{m}$. For each $z'_i, i \leq \mathsf{m}$, sample $d-1$ uniformly random values $s_{i,2}, s_{i,3}, \cdots s_{i,d} \in \mathbb{Z}_p$. Let $z_{i,j} = \boldsymbol{a}_i \cdot s_{i,j}$ for $i \leq \mathsf{m}; 2 \leq j \leq d_i, Z_i = [z'_i, z_{i,2}, z_{i,3}, \cdots, z_{i,d_i}]$. Now $\mathcal{A}'$ invokes j instances of algorithm $\mathcal{D}'$ each with $\ell - 1$ vectors $\hat{Z}_{i,j}, i \leq \ell - 1$ and an additional input a vector $Z'_j = [Z_j, Z_{j+1}, \cdots Z_{j+n}]$ for $1 \leq j \leq \lceil \frac{\mathsf{m}}{n} \rceil$. Each instance of $\mathcal{D}'$ distinguishes the input vector from uniformly random vector $U(\mathbb{Z}_p^n)$ with an advantage ϵ, thus algorithm $\mathcal{A}'$ distinguishes an LWR instance from a random vector with an advantage at-least ϵ.

In the case where t' servers are corrupt, the adversary has access to the secret key shares of the t' servers. In such a case, the algorithm $\mathcal{A}'$ supplies only $n - t$ element vectors to each instance of the algorithm $\mathcal{D}'$ through the vector $[Z_j, Z_{j+1}, \cdots Z_{j+n-t}]$. Each $\mathcal{D}'$ simulates the t servers by sampling t values $Z_{j+n-t}, \cdots, Z_{j+n} \in \mathbb{Z}_p^{d_i}$. It constructs the vector $Z'_j = [Z_j, Z_{j+1}, \cdots, Z_{j+n}]$, computes $\boldsymbol{sk}_j = \lambda_i \cdot Z_j$ for each element of $\lambda_i \in \{-1, 0, 1\}$. The algorithm $\mathcal{D}'$ differentiates the vector from uniform random vector with an advantage ϵ. The algorithm $\mathcal{A}'$ differentiates the LWR instance from random vector with an advantage of at-least ϵ. □