Hidden in Plain Sight: Storing and Managing Secrets on a Public Ledger

Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Sandra Deepthy Siby,
Nicolas Gailly, Philipp Jovanovic, Linus Gasser, and Bryan Ford
{*firstname.lastname@epfl.ch*}
EPFL

Abstract

Current blockchain systems are incapable of holding sensitive data securely on their public ledger while supporting accountability of data access requests and revocability of data access rights. Instead, they either keep the sensitive data off-chain as a semi-centralized solution or they just publish the data on the ledger ignoring the problem altogether. In this work, we introduce SCARAB the first secure decentralized access control mechanism for blockchain systems that addresses the challenges of accountability, by publicly logging each request before granting data access, and of revocability, by introducing collectively managed data access policies. SCARAB introduces, therefore, *on-chain secrets*, which utilize verifiable secret sharing to enable collectively managed secrets under a Byzantine adversary, and *identity skipchains*, which enable the dynamic management of identities and of access control policies. The evaluation of our SCARAB implementation shows that the latency of a single read/write request scales linearly with the number of access-securing trustees and is in the range of 200 ms to 8 seconds for 16 to 128 trustees.

1 Introduction

Blockchain technology is considered an important building block for many next-generation systems with applications in sectors like finance [44], healthcare [40], e-democracy [31] and insurance [37]. These distributed applications commonly assume shared access to sensitive data between independent and often mutually distrustful parties. However, mechanisms to enforce access control are either not present at all or realized rather naively.

To share sensitive data, an application could, for example, encrypt the information towards the public keys of the intended recipients and then publish the ciphertexts on a distributed ledger, such as Bitcoin's blockchain [26]. Once the data is public, however, the application has no longer any control over who accesses the data at which point in time and can also not change access rights retrospectively since the access control is enforced statically during encryption. To prevent these issues, many current systems [17, 45, 46] fall back to semi-centralized solutions that record access requests and access permissions on-chain but keep the data itself and the decryption keys off-chain in a centralized or distributed storage service [6]. This approach makes the storage provider a single point of failure, as it can simply deny access, even for legitimate requests or decrypt data undetected.

A blockchain-based system supporting shared access to sensitive on-chain data therefore has to address the following challenges: (1) Ensure accountability of all accesses to the sensitive data that is secured on-chain. (2) Provide a mechanism for adding and removing access rights since, *e.g.,* not all eligible recipients of a given data record might be known initially when the record is stored. (3) Ensure that changes to the access rights and execution of data access requests are done atomically, *e.g.,* to avoid the participants from exploiting race conditions to gain access to sensitive data while not having the necessary permissions (anymore). (4) Prevent all single points of failure, *e.g.,* prevent a single data storage provider from decrypting secrets without permission.

To address these challenges we introduce the Secret-Caretaking Blockchain SCARAB, the first decentralized data-sharing system that avoids any single point of failure and enables users to encrypt secrets "care-of" the blockchain itself. SCARAB therefore implements an access control mechanism that enforces *accountability*, supports dynamic access rights management that enables *revocability*, and ensures *atomicity* of changes to access policies and execution of access requests. In particular, we make the following contributions:

First, SCARAB introduces *on-chain secrets (OCS)*, which combine threshold cryptography [10, 16, 32, 34, 35] and blockchain technology [20, 26] to enable a client to put encrypted data under the control of a set of trustees who collectively enforce data access as stated by a client-specified policy file. If an eligible reader requests access

to an on-chain secret by providing a cryptographic proof, the validators atomically log the proof on the blockchain to guarantee accountability and then hand out the protected data. The key technical idea of OCS is to decouple the per-object state, *i.e.,* the access request, from the secret state. To realize OCS we propose two approaches called *one-time secrets (OTS)*, which allows to verifiably encrypt a secret under the control of an ad-hoc set of trustees, and *long-term secrets (LTS)*, which allows to verifiably encrypt a secret to a well-defined set of trustees. OTS is a simple implementation of OCS and does not require the trustees to store protocol state. However, this simplicity causes higher encryption and decryption cost. LTS utilizes distributed key generation [10] and verifiable secret sharing [35] for bootstrapping and requires the trustees to store protocol state. LTS has, in comparison to OTS, less client-side overhead with respect to en-/decryption and support additional features, such as trustee group reconfiguration.

The second contribution of SCARAB is the use of skipchains [28] for making the access control more dynamic, without adding a single point of failure. Although there are proposals to use blockchains for identity management [5, 41] or access control [46], they use the blockchain to enforce transparency on a centralized identity manager. Instead we enable the users to remain self-sovereign over their identity and use the blockchain as a serialization tape to guarantee atomicity over updates. In SCARAB we use skipchains to manage identities and to enable dynamic access control so that anyone can follow the identity of a user or organization over time, without trusting centralized look-up services. This *personal* blockchain creates a digital identity for the user who can now use it to include multiple aspects of their digital presence, such as their SSH or PGP keys, and to enable external clients to reliably and securely track the authoritative set of keys of the user. The IdS is also used to create a *federated* identity of an organization. Such an IdS points to other personal IdSs that can have variable access levels and can be managed autonomously. At the same time, the users are free to evolve their personal identity, adding and removing keys, without gaining any more privileges. Revoking an individual's access is as easy as excluding their personal IdS from the federated IdS. Furthermore, we use the same federated IdS for defining dynamically evolving access-control policies that are expressed in a JSON-based language.

We implemented both versions of OCS and show that they have a moderate overhead of hundreds to thousands msec, which scales linearly to the provided decentralization, thus making SCARAB a realistic system for real-world applications. Furthermore, we implemented personal IdSs and dynamic policies, and we show that the overhead of using them is in the order of usec. SCARAB is already used for multiple PoCs. We describe specific applications in Appendix C.

Our main contributions are as follows:

- We introduce SCARAB, a secure and decentralized data-sharing platform that provides dynamic access-control, auditability and atomic handling of decryption requests.
- We introduce on-chain secrets that use threshold encryption and consensus to enable a blockchain system to hold secret state without introducing single points of failure.
- We introduce IdS, a blockchain data-structure that enables both persons and federated groups to remain self-sovereign over their identities and dynamically evolve them. We also use IdSs to express dynamically evolving access-control policies.

2 Background

2.1 Blockchain systems

A blockchain is a a distributed append-only log used in decentralized applications [42, 26]. Blockchain takes its name because it is composed of *blocks* that are chained together via hashes. As each block includes in its header a hash of the parental block(s) leading to the block depending on the entire prior history, thus forming a tamper-evident log.

SCARAB is based on a specific blockchain system called collective authority or *cothority* [39]. SCARAB implements its blockchain using ByzCoin [20] that builds on top of CoSi [39] to create a scalable and secure blockchain that supports a cryptocurrency. Chainiac [28] is a system for transparent software-updates. One of Chainiac's contributions is the extension of ByzCoin's blockchain with multi-hop forward and backward *skiplinks* that enable efficient timeline traversal. This authenticated data structure is called *skipchain* as it is a combination of skiplists and blockchains that provides trust delegation via digitally signed forward links. In SCARAB, we employ skipchains to manage identities and access policies. SCARAB can be seen a special use case of sharded blockchains [7, 21] that chooses subsets of nodes to perform a subset of total system functionality; except in SCARAB we are choosing subsets for two different heterogeneous roles (blockchain management and consensus versus secret caretaking) rather than homogeneous, symmetric shards.

2.2 Threshold Cryptosystems

Secret sharing was introduced independently by Blakely [3] and Shamir [34] in 1979. A (t,n)-secret sharing scheme, with $1 \leq t \leq n$, enables a dealer to share a secret a among n trustees such that any subset of t honest trustees can reconstruct a, whereas smaller subsets cannot. In other words, the sharing scheme

can withstand up to $t-1$ malicious participants. The downside of these simple secret sharing schemes is that they assume an honest dealer which might not be realistic. Verifiable secret sharing (VSS) [9] adds verifiability to those schemes: it enables the trustees to verify that the shares distributed by the dealer are consistent. VSS has a wide range of applications such as threshold signature and threshold encryption, which we describe below. Finally, publicly verifiable secret sharing (PVSS) [32] is a variation of VSS that enables not only the trustees but also any external third-party to verify the secret shares distributed by the dealer.

Once we are able to share a secret, we can construct more complicated systems out of it. In SCARAB, we use a distributed key generation (DKG) protocol [10], and a threshold encryption protocol [35]. The idea behind DKG is to remove the trusted dealer from the secret sharing scheme by having one secret sharing round per trustee. Afterwards, the validity of the secret sharing protocol is verified through acknowledgments of reception of valid shares. Once consensus is reached on the correctly shared secrets, each trustee can combine all her shares to a new share, which corresponds to the distributed key. The upside of this protocol is that nobody knows the distributed secret key and, as a result, the key can only be used if and only if a threshold of servers decide that using it is the correct thing to do.

After generating the key, a client can encrypt data under the publicly shared key. One option is to use ElGamal encryption, but it is susceptible to chosen ciphertext or malleability attacks. In order to provide a secure threshold encryption scheme, Shoup et al. [35] show how to bind the access-control with the ciphertext via NIZK proofs, and we use this technique in SCARAB (see Appendix B).

3 SCARAB

In SCARAB, we solve the following problem: Alice wants to share some sensitive data with Bob. She wants to preserve the capability of withholding the data later and she wants to log the fact that Bob accessed the data. Concurrently, Bob wants to be able to dynamically change his identity (*i.e.,* his public keys), even after the data has been encrypted, but still be able to decrypt. Bob does not trust Alice to deliver the data, as she might try to log an access request and not serve the data. Alice does not trust Bob, as he might want to secretly obtain the data (*e.g.,* PGP encryption) and claim that he never accessed.

3.1 Strawman Solution

This section introduces *StrAcc*, a strawman system that we use to outline SCARAB's design. Below we describe one encryption/decryption round of *StrAcc*, where Alice encrypts some data for Bob. *StrAcc* is a combination of PGP with Bitcoin's [26] blockchain. First, Alice encrypts the data using the PGP key of Bob and, in order to guarantee that the data will be available when she goes offline, she posts the encrypted message on Bitcoin's blockchain. When Bob wants to read the data, he can download the correct block from Bitcoin's blockchain to retrieve the ciphertext and use his private PGP key to decrypt and read the data.

StrAcc already provides a similar functionality to SCARAB, but it has two significant security restrictions. First, upon the release of the encrypted data, Alice loses ownership of the data-access policy. Even if she wants to revoke access and writes a revocation record to the blockchain, the ciphertext is already available and encrypted to Bob's private key, and Bob can still decrypt the document. Therefore, she is no longer able to withhold this data from Bob. Second, Alice is unaware whether Bob tried to access and decrypt the data. Bob has plausible deniability in case of a privacy breach. *StrAcc*'s design also falls short in terms of usability. First, if the identity of Bob corresponds to multiple keys (per device key or Bob is an organization with multiple employees), Alice will have to encrypt the message to each individual key. Second, if Bob wants to rotate his keys for security reasons, he no longer has access to the data, unless he asks Alice to redo the protocol.

To address those issues, we introduce two new capabilities in *StrAcc* that results into SCARAB:

1. To enable auditability, revocability and atomicity, we introduce on-chain secrets (Section 4).
2. To remove the need for static identities and access-control policies, we introduce IdS (Section 5) that enables Bob to manage his identity while preserving a provable trust delegating path to his new keys.

A high-level overview of how Alice uses SCARAB to encrypt data to Bob is illustrated in Figure 1. In the remainder of this section, we define the goals of SCARAB and the environment it is deployed into.

3.2 System Overview

System Model. The actors and the workflow are depicted in Figure 1. We assume that the writer (Alice) always symmetrically encrypts the data under a key k, which is the shared secret. Alice can store the encrypted data either on-chain or off-chain, but in either case, the reader (Bob) has to retrieve it and verify its integrity before making a read transaction. SCARAB implements two logically separate cothorities.

The first one is called the *Access-control Cothority (AC)* that runs Byzantine consensus [20] and maintains the blockchain (*i.e.,* the per-object state) where transactions for reading/writing from/to the chain are logged, and encrypted data is stored. The access log serializes all transactions and transparently maintain proofs.

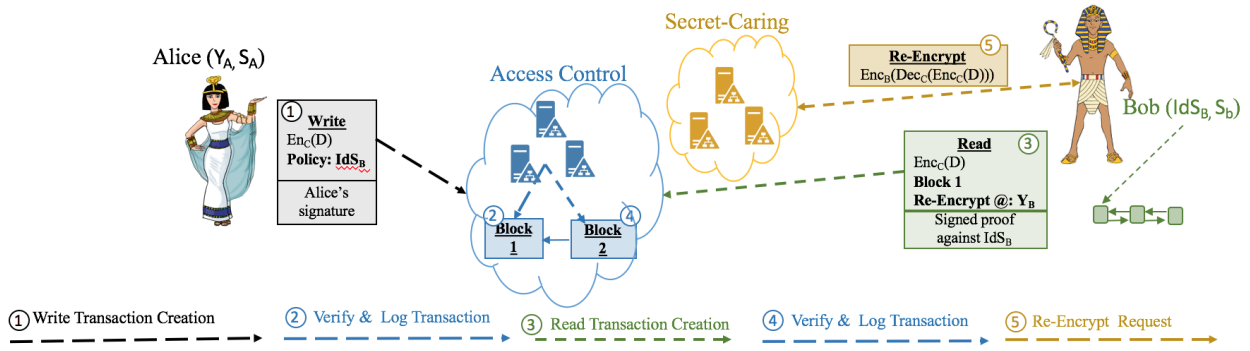

Figure 1: Alice encrypts the data "care-of" the SC, adding Bob's IdS in the access control label. AC verifies and logs the transaction. Bob can prove ownership of the identity to AC which logs the proof. Finally Bob sends the logged proofs to the SC, which re-encrypts the data to the public key provided by Bob.

The second cothority is the *Secret-caretaking Cothority (SC)* that manages the secrets. There can be a per-secret (OTS) or dedicated (LTS) SC. The SC trustees need to maintain secret state, which is their private keys (OTS and LTS) and the shared key shares (LTS). They do not run consensus for every transaction nor maintain a blockchain.

The same set of nodes can physically run both the AC and the SC, as in the evaluation of Section 8. We separate the roles logically both for architectural cleanness and flexibility, e.g., in case its worthwhile to provision the nodes for one differently from those for the other (e.g., bulk storage only on AC nodes, stronger security lockdown for the long-term private keys for the SC nodes).

Threat Model. Readers (*e.g.,* Bob) do not trust writers (*e.g.,* Alice) to deliver the data as writers might try to log a read transaction and not serve the data, framing the readers in case of a privacy breach. We assume that writers encrypt the correct data and symmetric key, as readers can realease a protocol transcript and blame in case of misbehavior. Writers do not trust readers as readers might want to get the data in secret (*e.g.,* as in *StrAcc*) and claim that they never tried to decrypt. We do not address the case of multiple mutually distrustfull readers, as we cannot pinpoint the exact source of leakage. The security assumption of the AC is the standard blockchain threat model (Byzantine), adapted for cothorities as prior work [20, 28] namely, that at most $1/3$ of the active cothority servers are malicious. Finally, the threshold of the VSS schemes on the SC is a design trade-off between availability and security. We set it to $1/3$ for simplicity as we use the same servers to implement the cothorities.

Network Model. As everything is done over the Internet, we borrow the network assumption from other cothority protocols [38, 28], specifically a weakly synchronous network.

3.3 System Goals

With SCARAB, we have the following primary goals with respect to security and usability

3.3.1 Security Goals

G1: Decentralization No single point of failure.

G2: Privacy of Secrets Secrets encrypted on-chain remain secret to non-authorized parties.

G3: Auditable Decryption Before an authorized user accesses data, there is an auditable proof logged.

G4: Revocability The data owner does not lose ownership over the data's access policy (as long as nobody performs an auditable decryption of the data) and is able to revoke access rights.

G5: Atomic Decryption If and only if a read transaction is logged does the reader receive the secret data.

G6: Transparency All accesses to data are provable against a public log (blockchain), and clients can verify that the proofs they see have been disclosed.

3.3.2 Usability Goals

G7: User-Sovereign Identity A user or organization is sovereign over their identity. The identity provider does not hold the keys of the user (unless requested), hence is unable to impersonate the user.

G8: Dynamic Identity Management A user or organization is able to dynamically evolve (e.g. rotate keys for security) their online identity without losing access to any data.

4 On-Chain Secrets (OCS)

This section introduces the OCS protocols that provide accountability of accesses on secret data, revocability of published data and atomicity between provably requesting data and securely accessing them.

4.1 OCS Overview

In this section, we give an abstract overview of OCS. For now, we assume that the access policy is a simple encrypt-to-public-key, where Alice statically binds the secret with Bob's public key. Bob signs the read transaction to request access and the AC verifies the signature before accepting the read. In Section 5, we remove this constraint and define dynamic identities and policies.

As shown in Figure 1, when the writer puts a secret on-chain, she sends a write transaction to the AC; whereas the reader sends a read transaction to request access. Figure 1 displays the transactions in LTS; OTS write transactions additionally include the encrypted secret shares and the corresponding NIZK encryption consistency proofs [32].

We describe two approaches to implementing OCS that enable us to consider different tradeoffs. One-Time Secrets (OTS) uses PVSS and employs a per-secret SC. OTS's simplicity enables each write transaction to define a fresh, ad hoc group of SC servers that do not require any prior knowledge of, or coordination with, each other. However, simplicity comes at a cost: both the write transaction size and the encryption/decryption overhead linearly increase with the size of the SC. This is because (1) encrypted shares are stored in the write transaction at the AC and (2) the number of shares is equal to the number of servers in the SC.

Long-Term Secrets (LTS) is the second approach to implementing OCS; it uses DKG and VSS. LTS requires a coordinated bootstrapping phase (DKG) of the SC and for the SC servers to remember some minimal state of their VSS secret shares. Unlike OTS, LTS requires a fixed or pre-determined group of servers for the SC. On the other hand, write transaction size and encryption/decryption overhead are constant in LTS. Furthermore, LTS enables additional capabilities such as reconfiguration of the trusted servers and delegation of the "secret-caretaking" to a different SC.

4.2 One-Time Secrets (OTS)

Our first implementation of OCS is OTS, which is based on PVSS [32]. As the PVSS dealer, Alice initializes the protocol by creating an encrypted share of her secret for each server in the SC (SC corresponds to the PVSS trustees in Section 2) by using their public keys. She uses the shared secret to generate the encryption key, symmetrically encrypts the data under the encryption key, and sends a write transaction to the AC to store it on the blockchain. Later, Bob signs a read transaction and sends it to the AC. If the AC verifies that Bob is authorized to access the data, he can show the proof-of-access, together with the encrypted shares, to each server in the SC (PVSS trustee), and obtain the decrypted shares. Once Bob obtains a threshold number of valid decrypted shares, he can reconstruct the symmetric key, hence the original data. Figure 2 demonstrates the protocol steps for OTS.

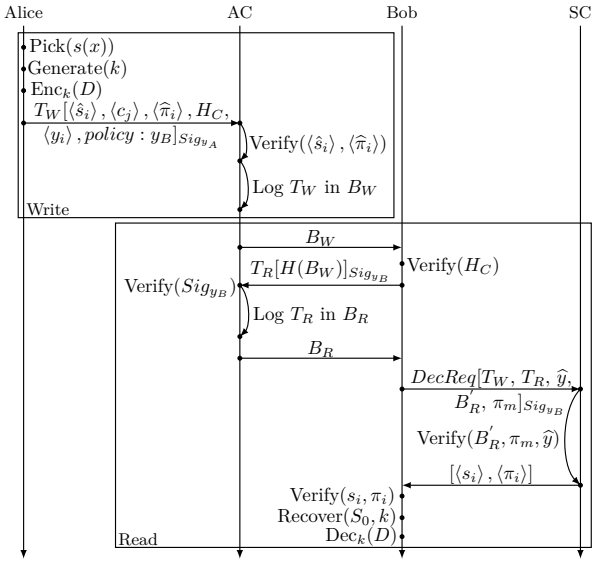

Figure 2: OTS Protocol Communication

4.2.1 Protocol

Let $\mathscr{G}$ be a multiplicatively written cyclic group of large prime of order q with generator G, where the set of non-identity elements in $\mathscr{G}$ is denoted by $\mathscr{G}^*$. We assume that Alice and Bob collectively trust a set of servers that, denoted by $N = \{1,\ldots,n\}$, constitute the SC. We refer to each server i in the SC as an *SC trustee*, and we denote its private and public key by x_i and $y_i = G^{x_i}$, respectively. Similarly, we use x_A, y_A and x_B, y_B to denote Alice and Bob's private and public keys, respectively. We also assume that the AC has an aggregate public key $\hat{y}$. We require that $t = \lfloor |N|/3 \rfloor + 1$, which is the PVSS threshold, as described in Section 2. Finally, we assume that the AC and SC verify the signatures of the protocol messages that they received from Alice and Bob.

Write Transaction Processing (Writer). Alice creates a write transaction as follows:

1. Compute $B = H(y_B)$ to map Bob's public key y_B to a group element $B \in \mathscr{G}^*$, used as the base point for generating the PVSS polynomial commitments.
2. Choose a secret sharing polynomial $s(x) = \sum_{j=0}^{t-1} a_j x^j$ of degree $t-1$. The secret to be shared becomes $S_0 = G^{s(0)}$.
3. For each SC trustee i, compute the encrypted share $\hat{s}_i = y_i^{s(i)}$ of the shared secret S_0 and create the corresponding NIZK encryption consistency proof $\hat{\pi}_i$, where $1 \le i \le n$. The exact NIZK proofs can be found in Appendix A. Create the polynomial com-

mitments $c_j = B^{a_j}$, for $0 \leq j < t$.

4. Compute $H(S_0)$ and use it as the symmetric key k to encrypt the message m. The encrypted message is $C = Enc_k(m)$.
5. Create the write transaction:

$$T_W = [\langle \widehat{s_i} \rangle, \langle c_j \rangle, \langle \widehat{\pi_i} \rangle, H_C, \langle y_i \rangle, policy : y_B]_{Sig_{y_A}}$$

where $\langle \widehat{s_i} \rangle$ is the list of encrypted shares, $\langle c_j \rangle$ is the list of polynomial commitments, $\langle \widehat{\pi_i} \rangle$ is the list of NIZK proofs, H_C is the hash of the ciphertext C, $\langle y_i \rangle$ is the public keys of the SC trustees, and *policy* is the public key of Bob to designate him as the decryptor of the ciphertext. Send it to the AC.

When creating a write transaction, Alice derives the group element B from $H(policy)$ to tie the encrypted PVSS shares to *policy*. Suppose an adversary obtains a write transaction from the blockchain, forges a new transaction where they keep everything from the original one except for *policy*. They change *policy* with their public key, and dispatches it to the AC. As the AC will derive a new B using $policy'$, the PVSS proofs in the forged transaction will not verify against the new B. This way, the AC can immediately detect a malicious write transaction (chosen-ciphertext secure) and discard it.

Logging the Write Transaction (AC). Before logging the write transaction, AC ensures that it contains valid encrypted shares:

1. Derive the group element B from $H(policy)$.
2. Verify each encrypted share $\widehat{s_i}$ against $\widehat{\pi_i}$ using $\langle c_j \rangle$, $\langle y_i \rangle$ and B (Appendix A). This step guarantees the validity of the encrypted shares.
3. If all shares are valid, log T_W.

Read Transaction Processing (Reader). Bob performs the following steps to create a read transaction:

1. Retrieve the ciphertext C either from Alice, the AC or a highly available storage, and the block B_W, which stores write transaction T_W, from the AC.
2. Compute $H^{'} = H(C)$. Compare H' to H_C in T_W to ensure that the ciphertext C is not altered.
3. If the hashes agree, create the read transaction:

$$T_R = [H_W]_{Sig_{y_B}}$$

where H_W is the hash of the write transaction. Send it to the AC.

Logging the Read Transaction (AC). AC stores T_R on the blockchain as follows:

1. Use y_B in T_W to verify the signature of T_R .
2. If the signature is valid, then logs T_R.

Decryption Request (Reader). Bob sends a decryption request to each SC trustee to obtain the decrypted shares:

1. Create a decryption request:

$$DecReq = [T_W, T_R, B_R^{'}, \widehat{y}, \pi_m]_{Sig_{y_B}}$$

where $B_R^{'}$ is the signed header of the block that stores T_R, and π_m is the Merkle path to prove the membership of T_R in the blockchain. Send *DecReq* to the trustees.

Decrypting the Shares (SC Trustee). Upon receiving a decryption request, each SC trustee executes the following steps:

1. Use y_B in T_W to verify the signature of T_R .
2. Verify that T_R is logged on the blockchain using π_m, $\widehat{y}$ and the Merkle root in $B_R^{'}$.
3. Compute the decrypted share $s_i = (\widehat{s_i})^{x_i^{-1}}$ and create the corresponding NIZK decryption consistency proof π_i (Appendix A).
4. Send s_i and π_i back to Bob via a private channel.

Key Reconstruction. (Writer) Bob carries out the following steps to reconstruct S_0, followed by k:

1. Verify each s_i against π_i.
2. If there are at least t valid shares, use Lagrange interpolation to recover S_0.
3. Recover k by computing $H(S_0)$ perform the decryption to obtain $m = Dec_k(C)$.

4.2.2 Mapping to Goals

OTS achieves the following security goals:
Decentralization: Both the logging and the key handling processes can withstand up to t failures.
Privacy of Secrets: OTS's secrets remain private due to the privacy of PVSS, as long as the assumption that less than t trustees collude. AC verifies the encrypted shares without obtaining any information and all decrypted shares are sent to the reader via a private channel.
Auditable Decryption: Any read transaction is validated by the AC and serialized on-chain. The reader has enough proof to request shares only afterwards.
Revocability: Alice can post a revocation transaction (it is logged the same way as a read transaction) and ask the AC to stop accepting new reads.
Atomic Decryption: Given a correct AC and SC (less than t Byzantine) and partially synchronous network, OTS will either execute both (a) log the read transaction and (b) re-encrypt the key or reject (a) and stop.
Transparency: All actions are reads and writes and are provable against the blockchain maintained by the AC.

4.2.3 Advantages and Shortcomings

OTS is the first implementation of OCS and it has several advantages. First, it does not need a bootstrapping phase at the SC to generate a collective private-public key pair. Second, OTS enables the use of a different SC for each secret without requiring the servers to maintain protocol

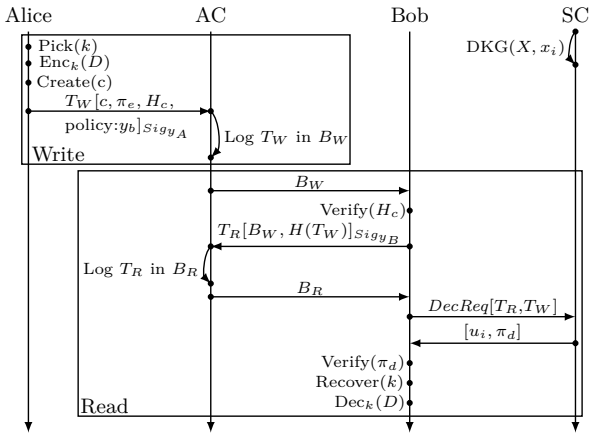

Figure 3: LTS Protocol Communication

state. Although these properties make OTS simple to implement and reason about, it still has shortcomings.

First, OTS has high PVSS setup cost and share reconstruction cost for the client. Alice needs to evaluate the secret sharing polynomial at n points, create n encrypted shares and NIZK proofs, along with t polynomial commitments. As these operations rely heavily on elliptic curve arithmetic, they will be computationally expensive, especially for large numbers of shares. Similarly, Bob has to verify up to n decrypted shares against the NIZK proofs and reconstructing the shared secret on his device.

A second shortcoming is the large transaction size. The SC trustees do not store any protocol state, hence the write transaction contains the encrypted shares, NIZK proofs and the polynomial commitments, making it linearly increasing to the SC size. Finally, Alice cannot use OTS to share an already existing encryption key. She has to either re-encrypt the data or use another key that encrypts the original key.

4.3 Long-Term Secrets

The second OCS design, LTS, addresses the above limitations. The key idea behind LTS is that there is a dedicated SC that is aware of participating in the system and provides the necessary availability and bootstrapping procedures. After the bootstrapping phase of Distributed Key Generation (DKG), the overhead of encrypting secrets is constant as a write transaction simply contains an ElGamal encryption of the secret to a single shared key, eliminating the need to publish individual shares of each secret. Furthermore, the configuration of the SC can change without compromising the availability by re-sharing the same secret shared key or re-encrypting all the secrets to a new SC. The basic workflow of LTS is shown in Figure 3.

4.3.1 Protocols

Distributed Key Generation: Unlike in OTS, SC needs a bootstrapping step called DKG in LTS. A DKG [10] protocol is run by a distributed set of servers to collectively generate a private/public key pair in a way that the private key is not known to any single party, but can be reconstructed by a threshold *t+1* of available *key shares*. Each server holds one such share; and if *t+1* shares collaborate, they can reconstruct this key.

In order to remain consistent with our adversary and network models we could use HybridDKG [18]. However, given the rarity of the event of key generation, we assume a pessimistic synchrony assumption for the DKG (*e.g.*,. $\Delta = 1$ minute) and implement the DKG introduced by Gennaro et al. [10], because of its simplicity and the fact that it produces uniformly random keys.

Let $N = \{1, \ldots, n\}$ denote the list of servers that participate in the SC and that $n = 3f + 1$, where f denotes the number of dishonest servers. Furthermore, let t be the reconstruction threshold such that $t = f + 1$. We make the standard assumptions of using g to denote a generator of a secure subgroup $\mathscr{G}$. After the DKG is executed, the output is a public key $X = g^x$, where x is the private key that is unknown. Additionally, each server i holds a share of the secret share denoted as x_i and all servers know the public counterpart $X_i = g^{x_i}$.

LTS Write Transaction Processing (Writer, AC): When Alice wants to share a secret, she will create a T_W

1. Cothority Key Retrieval: First Alice retrieves X, the public key of the SC that we assume for simplicity to be available in the genesis block of the blockchain. We can provide more dynamic cothority evolution using ideas from Chainiac [28] and is orthogonal to SCARAB.
2. Symmetric Encryption: Alice chooses a key k to symmetrically encrypt the message and then embed k to a point $m \in G$.
3. Ciphertext Generation: To share the secret with the SC, Alice uses ElGamal encryption modified for threshold cryptosystems [35]. More specifically, she picks r at random, computes $c = (X^r m, g^r)$ and also creates the NIZK proofs π_e to guarantee that the system is chosen ciphertext secure and non-malleable. The exact NIZK proofs can be found in Appendix B. Finally, she defines a policy L that includes the public key y_B of Bob.
4. Write Transaction Creation: Alice creates a write transaction that contains c, π_e, and the hash of the symmetrically encrypted data H_c, then she signs the transaction using her private key x_A and dispatches it to the AC for logging.

LTS Read Transaction Processing

1. Encrypted Data Retrieval (Reader): Bob requests

the block with Alice's write transaction and the encrypted data. He verifies Alice's signature and π_e and that $H(data) = H_c$ to ensure data availability.
2. Read Transaction Creation (Reader): Bob creates a read transaction pointing to the write transaction (block and transaction hash) and signs it with his private key x_B as proof of access, then he sends it to the AC for validation.
3. Read Transaction Validation (AC): AC verifies the signature against L and logs the read transaction.
4. Collective Re-encryption (Reader, SC): Bob collects the logged read and write transactions from the AC and either goes to each SC server i and collects the encrypted share $u_i = g^{rx_i}$ or delegates the reconstruction to an exeternal server. We describe both approaches below.

Secret Retrieval (Reader)

1. Share Verification: For each share u_i, Bob verifies the decryption share creation NIZK proof π_d (Appendix B) until he collects t valid shares.
2. Combining Shares: Assuming that Bob has a set of t decryption shares (i, u_i), the recovery algorithm does Lagrange interpolation of the shares:

$$X^r = \prod_{k=0}^{t} u_i^{\lambda_i}$$

, where λ_i is the i^{th} Lagrange element.
3. Decrypt: Bob computes the inverse of X^r and finds $m = cX^{-r}$. From m he can derive k and symmetrically decrypt the original message.

Collective Re-encryption If Bob retrieves the data in a portable or IoT device, then it is preferable that he does not have to do the costly public-key operations. In this section, we devise a protocol that enables any untrusted server, instead of Bob, to do the share reconstruction. We assume that this server is honest-but-curious and will not DoS Bob. If the server delivers an invalid ciphertext, Bob can detect that he cannot decrypt the data and either ask another server or do the protocol himself. Bob selects a server to do the share reconstruction, instead of himself. He signs this delegation request with x_B.

1. ElGamal Re-Encryption. Each SC server that created his decryption share as $g^{rx_i} = u_i$ ElGamal encrypts the share for Bob using x_i as the blinding factor instead of a random r'. The new share is: $g^{rx_i} g^{x_B x_i} = g^{(r+x_B)x_i} = g^{r'x_i} = u'^{x_i} = u'_i$. Then, to show that the share has been correctly generated he computes the NIZK proofs π_d as before (App. B).
2. Share combination. The untrusted server collects t valid shares and reconstructs the re-encrypted secret with Lagrange interpolation as shown above. The end result is $g^{r'x} = g^{(r+x_B)x}$. Note that the server never sees g^{rx}, hence learns nothing.
3. Secret retrieval. Bob obtains $g^{(r+x_B)x}$ and as he knows g^x and x_B he can find $-x_B$ and compute $g^{x^{-x_B}} = g^{-xx_B}$. Finally he computes $g^{rx} = g^{(r+x_B-x_B)x}$ and carries out the decryption as it is explained above.

4.3.2 Evolution of SC

Given that the SC is long lived in LTS, we need to make sure that it remains secure and available. A number of issues can arise during the lifetime of SCARAB. First, secret-holding servers can join and leave SC, which results in churn. Secondly, even if SC memberships are static, private shares of the secret-holding servers should be refreshed/rotated regularly (perhaps every month), so that an attacker cannot collect a threshold of shares over a sufficiently long time. Finally, for security reasons, we also want the shared private key of the SC to be rotated (*e.g.,* once every year or two). This will require re-encrypting the long-lived secrets from one shared keypair to another.

We solve the first two problems by periodically resharing [14] the existing collective public key when a secret-holding server joins or leaves the SC, or when the server wants to rotate its individual private key-shares. For the last challenge, where the SC wants to rotate the shared public/private key pair (X, x), SCARAB needs to collectively translate each individual valid secret under the new public key $Y = g^y$. To achieve this, we generate translation certificates [16].

On a high level, the translation functionality is as follows: Let $c_1 = (mg^{rx}, g^r)$ be an ElGamal encryption of a message m with respect to a public key X, and let x be the corresponding secret key that is shared by the trusted servers by using a threshold scheme as described before. The trusted servers want to compute the ElGamal encryption $c_2 = (mg^{yt}, g^t)$ of m, with respect to the public key Y. However, they do not want to expose m to any set of dishonest servers (or any other set of servers). Furthermore, they do not know the private key y that corresponds to Y. To translate c_1 to c_2, we need to execute the following two steps:

1. Trustee i selects a value t_i uniformly at random $\in \mathbb{Z}_q$, and computes $(a_i, b_i) = (g^{-rx_i} g^{yt_i}, g^{t_i})$. This is the *translation share*.
2. The SC servers (or an untrusted gateway) collect a threshold of shares and compute the translation (c_2, g^t) = $(c_1 \prod_{k=0}^{t} a_k, \prod_{k=0}^{t} b_k)$.

The above algorithm is not robust as is, therefore the trustees distributively generate robust translation certificates, using the protocol of Jakobsson et al. [16] to guarantee that they performed the translation correctly. Afterwards, the translation certificate is appended to the (re)write transaction of the secret on the blockchain for Bob to verify before he issues a read transaction.

Periodically creating fresh secret-holding keys and collectively translating all secrets in the currently-outstanding state from the old (X) to the new (Y) shared public keys is a per-object operation. This means that it can take a long time in a large system holds many secrets, hence it should be done rarely and over a long period of time where both X and Y are considered valid.

4.3.3 Mapping to Goals

LTS achieves the same security goals as OTS the same way except from the *Privacy of Secrets.*:
LTS remains private due to the unpredictability of the DKG algorithm that guarantees that the shared private key is never revealed, as long as less than t of the cothority servers collude. All shares are encrypted under y_B and remain private during reconstruction.

5 Identity-Based Access Control

In the design introduced so far, if Alice encrypts her data and posts a write transaction but later decides to share the data with more decryptors or if Bob decides to rotate his public key, Alice would have to revoke the previous write transaction and post a new one with modified access-control list. In this section, we show how to create decentralized dynamic identities and policies to gracefully handle the identity evolution and access control of SCARAB. Furthermore, we show how to protect against an adversary taking advantage of this new dynamic feature and trying to gain access using stale (*i.e.,* revoked) access rights.

5.1 Identity Skipchains

This section introduces the identity skipchain. The identity skipchain is inspired by CHAINIAC [28], where the authors introduce a double-linked blockchain that provides efficient timeline-traversal in order to enable the rotation of the configuration of a cothority without requiring the clients to trust a third-party look-up service after their initial bootstrap. In SCARAB we adopt the skipchain data structure, because of the additional trust-delegation property that it provides compared to normal blockchains, and use it to create and manage decentralized personal and federated identities that we call *Identity Skipchains (IdSs)*. IdSs enable SCARAB to decouple OCS from static identities, but it can also be used in more general identity management applications.

5.1.1 Personal Identity Skipchain

The personal identity blockchain was first introduced in a worksop talks [19]. In this section we reintroduce the design and extend it for design completeness. We define the per-user IdS that enables the user to access services (*e.g.,* OCS service) from multiple devices without the need to manually configure the remote locations every time a key changes. IdS is made of blocks storing two types of keys. The first type consists of the cold keys of the user (ideally stored offline in multiple devices): they are used only when the user evolves his identity. The second type consists of the warm keys that are used for the applications (*e.g.,* SSH, PGP key). The cold keys are used for signing forward links in order to delegate trust [28] between the digital identity representations of a user. To provide security against an adversary that uses an old block as the active one, we use the AC from SCARAB to periodically timestamp all IdSs and to enforce a single-inheritance rule (no-forks). This way, the freshness attack window is minimized to be between two timestamp events.

After applying IdSs on SCARAB, Alice is now able to encrypt data for the personal IdS of Bob and Bob can rotate his keys (*e.g.,* for security) without losing access.

5.1.2 Federated Identity Skipchain

Enabling people to manage their identities is one step towards our goal. However, defining the identity of an organization by adding all the keys and updates of employees to one shared IdS will quickly lead to a large number of conflicts, as multiple people concurrently try to change the IdS. For this reason, we scale by creating recursive federated IdSs.

As illustrated in Figure 4, the federated IdS includes not only the public keys, but also the personal IdSs of the employees (*e.g.,* Bob). This enables a user to manage his identity independently and re-use it at multiple locations. Also, revocation becomes cleaner, as removing an employee's IdS from the federated IdS automatically removes the employee's ability to create inclusion proofs against the federations' IdS.

After applying federated IdSs in SCARAB, Alice can now encrypt data to the full sales department of her company, and anyone able to provide a provable path from the federated IdS to his personal IdS can access. Furthermore, if the federated IdS includes personal IdSs, people can easily evolve their identities without conflicts.

5.2 Access Control Policies

One final issue remaining in SCARAB is dynamic access control. Even with IdSs, Alice cannot change the fact that she shared some data with the sales department. A simplistic solution would be to define a special "revoke transaction" and then a new write transaction with the new access-control list. In this section, we describe how to provide the same functionality without the need for new write transactions by tuning federated IdSs upside down and using them as *policy skipchains PolSs*

The intuition is that the same way federated identities evolve (to include / exclude authorized users) is used to evolve access-control rules. We use federated identities (or policy identities) to enable Alice to dynamically change the access writes of a specific document. Instead

of using the IdS of an individual or an organization in the access control policy, Alice creates a new PolS specific for the document. She then dynamically manages this PolS as any other federated identity. For example, she can create a PolS for her medical records and then share it dynamically with her new doctor. This enables Alice to always be sovereign over her data. Finally, we enhance the expressiveness of PolSs by defining access control rules that enable Alice to have multiple rules or rules with more complex conditions attached to her writes.

A policy is a set of rules that regulates access control to a resource. We use a simple JSON-based access-control language to express policies. In the following sections, we define our policy structure and describe how requests for access to resources are created and verified.

5.2.1 Policy Structure

In this section, we describe the structure of policies and demonstrate how they can be used for access control. A policy consists of a random ID, a version number and a list of rules that outline the access control. A rule comprises three components: *action*, *subjects* and *expression*. *Action* refers to the type of activity that can be performed on the resource (*e.g.,* encrypted document). It is an application-specific string indicating the activity. *Subject* is the list of users that are permitted to perform an *action*. We permit individuals and federations to be *subjects* similar to federated IdSs, hence a *subject* can be a public key or an IdS.

To build more sophisticated rules, we introduce *expressions*. An *expression* is a string of the form: $\{operator : [operands]\}$. Operator can be a logical operator (such as AND/OR) and operands can be a list of Subjects. An example is: $\{AND : [ID_{Group1}, ID_{Bob}]\}$. In the context of signatures, this means that both ID_{Group1} and ID_{Bob}'s signatures are required by a particular rule. We can combine expressions to express complex conditions for rules. For example: $\{`OR' : [\{`AND' : [S1, S2]\}, \{`AND' : [S3, S4]\}]\}$ evaluates to ((S1 AND S2) OR (S3 AND S4)). In Appendix D, we describe single and multi-signature access requests against policies and outline how they are created and verified.

5.2.2 Evolving Policies

Figure 4 shows an example of the full deployment of SCARAB, where Report X has a PolS, group A is a federation with its own federated IdS and Bob is part of A but also manages his personal IdS. The latest version of the PolS grants read access to Group A, of which Bob is a member. Bob can access Report X as he is a member of the group that has access. If a new version of the policy (v3) is created where Group A does not have access, its members will no longer be able to read the document. In the case of an access request from Bob's Key1 to Report

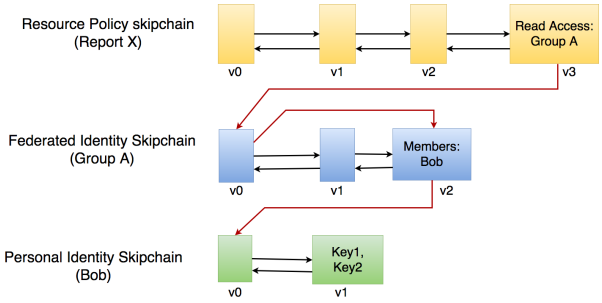

Figure 4: Evolution of policies by using skipchains.

X's policy, the path is shown in red. The rules in the latest policy block indicate which IdS to access. The skiplinks enable fast traversal through the policy's versions.

5.2.3 Transparent Logging against Races

One key idea on SCARAB's design is using the AC's blockchain to timestamp the latest versions of IdSs and PolSs. This guarantees atomicity of events such as dynamically changing an identity (e.g. to exclude someone) and soon after granting it more access rights.

Specifically, in a naive system that lets IdSs to evolve off-line without any transparent announcement, access policy structures will not be atomic with the structures being access controlled. For example, administrator Sally of the sales group, decides that Bob should be fired because he is performing industrial espionage, hence she removed the IdS of Bob from the federated IdS of the sales group. Afterwards Sally grants the rest of her employees access to the new corporate strategy plan. In a naive asynchronous access control system in which policy changes can take varying amounts of time to propagate and take effect (e.g., OAuth [12]), there is significant accidental time window in which Bob can still convince someone that he is part of the sales group, as he might be able to still prove membership to the controller of the the sensitive object (*i.e.,* to a threshold of SC trustees).

In SCARAB, all the changes of the IdSs are serialized together with the read and write transaction on the blockchain maintained by the AC. Hence the exclusion of Bob will be strictly after the granting of access. This means that Bob will be unable to provide a correctly timestamped (by the AC) proof to the SC trustees and as a result be unable to read the sensitive document.

5.3 Mapping to Goals

We inherit the achievment of the security goals from OCS, other than the revocability. Furthermore we also achieve the usability goals as follows:
Revocability: The writer is able to revoke access to the data by extending the PolS with an empty block, after the new block is timestamped by the AC no reader can

access any more.

User-Sovereign Identity: Even if the lookup of an identity is delegated to an identity provider (*e.g.,* using CONIKS [24]), the identity creator holds the private keys and is both sovereign over the identity and responsible for each evolution.

Dynamic Identity Management: The legitimate owner can dynamically change their identity's representation using their cold keys and re-use it to many federated IdSs.

6 Anonymous Access Control

In this section, we explore a second approach to access-control in OCS. Instead of looking into dynamic identities, we look into relationship privacy. More specifically, once Alice indicates that Bob should be able to decrypt her write transaction, it becomes evident and logged on the chain that Alice and Bob have some kind of relationship. However, if for example Alice is a whistle-blower and Bob is Wikileaks, this leakage is a security breach.

To solve this problem in SCARAB, we introduce an "on-chain private-key exchange" protocol. The basic idea of the protocol is that Alice hides Bob's identity in the write transaction, and generates a temporary public key for Bob to use in his read transaction. The corresponding private key can only be calculated by Bob, and a signature from this corresponding private key is enough for Alice to prove that Bob created the read transaction. In order for Bob to know which transaction to search for, Alice needs to notify him on a separate secure channel that she posted a write transaction for him. The protocol works as follows:

1. Key creation: Alice generates a random number r and uses it to generate Bob's temporary public key y'_B as $y'_B = y_B^r = g^{rx_B}$.
2. Write Transaction Creation: Alice uses y'_B instead of y_B in the policy of the write transaction. In order for Bob to be able to calculate $x'_B = x_B r$, he needs r. Alice encrypts r for Bob in the write transaction. She picks random number r' and encrypts r as $e = g^{x_B r'} r$ and $f = g^{r'}$. The final write transaction has y'_B as the allowed reader and e, f as further metadata of the ciphertext.
3. Read Transaction Creation: When Bob wants to read, he will first decrypt e by computing $f' = f^{x_B}$ and $r = \frac{e}{f'}$. Then he will compute $x'_B = r x_B$ and use it to sign the read transaction.
4. Auditing: If Alice wants to prove that Bob generated the transaction, she can simply release r. Then anyone can verify that $y_B = y'_B g^{-r}$ hence be sure that only Bob could know the x'_B.

This solution enables Alice and Bob to hide their relationship without forfeiting any of OCS's guarantees and it has already been deployed for industrial applications. This approach does not support dynamic identity evolution, as we cannot predict the future key of Bob, but is already used in an industrial PoC where relationship privacy is more important than dynamic identity evolution. We leave the extension of this protocol with dynamic identities to future work, and emphasize that if relationship privacy is required, then the identities should be static.

7 Security against Attacks

Our contributions are mainly pragmatic rather than theoretical as we employ already-proven secure cryptographic protocols. We argued on achieving the security goals in the previous sections. In this section, we describe the attacks that could happen in SCARAB and argue on SCARAB's defenses.

Malicious Reader/Writer: SCARAB's functionality resembles a fair-exchange protocol [30]. In such a protocol, a malicious reader wants to retrieve the secret without paying for it and a malicious writer wants payment without revealing the secret. In SCARAB, we protect against malicious readers and writers by employing the AC and SC as decentralized trusted third parties that mediate the interaction.

AC logs a write transaction on the blockchain only after it successfully verifies the encrypted data against the corresponding consistency proof. Similarly, before logging a read transaction, AC verifies that (1) it is for a valid write transaction and (2) it is sent by a party that is listed as an authorized reader in the matching write transaction. As for the SC, it decrypts the data for the reader after confirming that they are the authorized reader, and they have an auditable proof logged on the blockchain (*i.e.,* read transaction). By employing these mechanisms at the AC and SC, we protect SCARAB from malicious readers and writers. Additionally, because of the atomic delivery property, readers and writers are guaranteed that they will not be worse-off after the protocol ends. Writer know that the data will only be decrypted to the reader, if they have a valid read transaction logged in the blockchain and readers know that if there is a valid read transaction logged, then the SC will be convinced to deliver the data.

Chosen Ciphertext Attack SC's functionality resembles a decryption oracle, hence SCARAB needs to be CCA secure. A CCA attack would happen when Eve copies the transaction of Alice from the AC blockchain and dispatches a new write transaction that has the same ciphertext but a modified access-control list.

OTS protects against this attack as it uses the access-control list to derive the base-point that is used to create the NIZK consistency proofs in PVSS. Eve's attack will fail in SCARAB because she cannot generate correct proofs against her access list as she does not know the

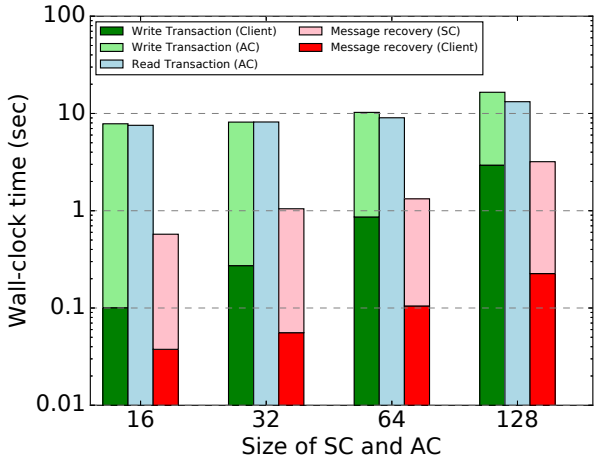

Figure 5: Latency of OTS protocol steps for varying sizes of SC/AC

decrypted shares (they are encrypted inside the write).

LTS protects against such an attack by using the NIZK-proofs introduced in the protocol of Shoup et al. [35], which binds the access-control list with the ciphertext at encryption time. Eve's attack will fail in SCARAB because she cannot generate correct proofs without the knowledge of the blinding factor of the ElGamal encryption.

Individual Malicious Trustees A final component that can misbehave in SCARAB are the AC and SC trustees. If they decide to DoS the system, this does not cause a problem as long as the threat model holds (less than t Byzantine). Similarly, if they try to collude and decrypt the secrets, they still do not hold enough decryption shares as long as the threat model holds.

8 Implementation and Evaluation

8.1 Implementation

In order to evaluate our system, we implemented SCARAB in Go [11] and will make it available on GitHub. The AC is implemented using the current skipchain code written for the cothority project. We implemented both the OTS and LTS verisons of OCS, complete with encryption and decryption of the shared secrets. For the IdSs and PolSs we implemented signing and verifying using AND, OR and NOT expressions. In our code, we used the Ed25519 curves giving 128-bit security. All the code will be made available under an open source license on GitHub. The total SLOC-count of each module is for skipchains 4,000 LOC, for OTS 2,000 LOC, for LTS 4,000 LOC and for IdSs 1,000 LOC.

8.2 Evaluation

In this section, we evaluate the performance of SCARAB. We first evaluate and compare the two implementations of OCS, then we look into the overhead introduced when using dynamic identities and policies. For both implementations of SCARAB we evaluate the time it takes to create a write and a read transaction with different sizes of SC and AC. We run all experiments using Mininet [25] run on 4 servers where each server has 256 GB of memory and 24 cores running at 2.5 GHz. To make realistic assumptions on the network, Mininet introduced 100 ms point-to-point latency between all nodes and a maximum bandwidth of 100 Mbps for each node. The basic questions we want to answer is whether SCARAB has an overhead that can be acceptable for blockchain systems (*e.g.,* Bitcoin validates transactions every 10-minutes), and whether we can scale it to hundreds of validators that provide a high degree of confidence that the thresholds assumed will hold. Similarly for IdSs we wanted to evaluate the overhead of creating and verifying proofs and see if it scales well when multiple levels of federated IdSs need to be searched.

8.3 OCS

In our experiments, we measure the cost of different protocol steps of OTS and LTS for a varying number of nodes, where each node is part of both the AC and SC. In OTS, we represent the cost in terms of wall-clock time, while in LTS, we additionally use the CPU time.

OTS For the OTS experiments, we measure the cost of three protocol steps, which are described in Section 4.2.1. Each step includes the message exchange latency between the parties that are involved in that step. *Write transaction* step consists of two consecutive parts that take place at the client and AC: The first part involves creating a write transaction by setting up the encrypted PVSS shares, polynomial commitments, NIZK proofs, and symmetrically encrypting the data. The second part involves the collective verification of the encrypted shares against the NIZK proofs and logging the write transaction on the blockchain. *Read transaction* step takes place at the AC, where a read transaction is collectively verified and logged on the blockchain. *Message recovery* step consists of two consecutive parts that take place at the SC and client: In the first part, upon receiving a *DecReq*, each SC trustee decrypts its PVSS share, creates a NIZK decryption proof, and sends it back to the client. The second part involves verification of the decrypted shares against the NIZK decryption proofs, reconstruction of the shared secret by performing the Lagrange interpolation, and decryption of the ciphertext.

Figure 5 shows the results of our measurements for

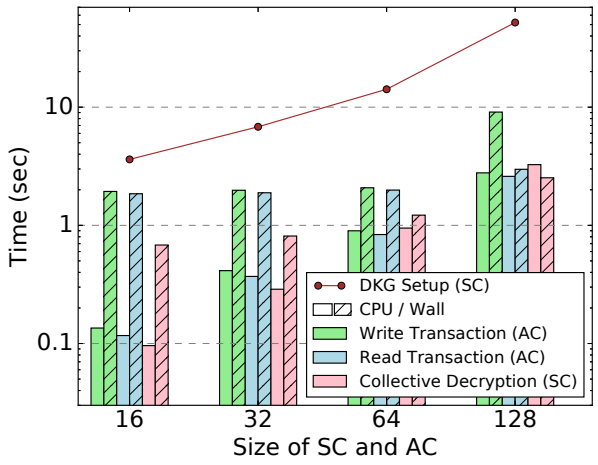

Figure 6: Latency of LTS protocol steps for varying sizes of SC/AC

varying sizes of SC and AC cothorities. Note that in the context of OTS, size of the SC equals the number of trustees involved in PVSS. First, we observe that creation of the write transaction at the client is a costly operation as it takes almost a second when the SC is comprised of 64 nodes. This result conforms with our expectations, as preparing the write transaction involves many elliptic curve arithmetic operations to set up the PVSS shares, commitments and proofs. Second, we observe that the write transaction and read transaction processing times at the AC are comparable, but the former takes ~250 ms longer on average than the latter. This is due to the fact that the AC collectively checks the NIZK proofs to verify the encrypted shares, which brings additional computational overhead. Our experiments also show that verifying the NIZK decryption proofs and reconstructing the shared secret are substantially faster than creating the write transaction: they differ by an order of magnitude for large numbers of shares (*e.g.,* with 128 shares: ~250 ms vs ~3 seconds). Finally, we observe that the time values for the SC part of the message recovery are an order of magnitude greater than their client counterparts. This is expected as the client dispatches a request to each SC trustee and waits until all respond.

LTS Figure 6 illustrates the cost of LTS with a varying number of nodes. First, we measure the time to setup the shared key of the SC. After the DKG, each node has a secret share of the overall secret and SCARAB stores the collective public key. Next, we measure a write transaction, where the client stores the symmetric key encrypted under the shared public key of the DKG, that is verified and logged by the AC. Then, we measure the read transaction that the AC (1) verifies by checking whether its signature belongs to a client that is authorized by the writer and (2) logs the read transaction on the blockchain. Finally, we measure the time it takes for a reader to receive the re-encrypted key from the SC; this is the time it takes for the SC to verify the read transaction and then do the actual re-encryption. As we can see, the LTS protocol scales linearly to the size of the cothority, but even for 128 servers takes less than 8 seconds. The CPU-time is significantly lower than the wall-clock time because of the networking overhead over a WAN that is present on the wall-clock measurements.

Number of servers in SC	Write transaction size (bytes)	
	OTS	LTS
16	4,086	160
32	8,054	160
64	15,990	160
128	31,926	160

Table 1: Write transaction size for varying SC sizes

Even though the DKG setup is quite costly in bigger settings, it is a one-time cost that is only incurred at the start of every epoch. The following read and write transactions are faster and smaller in size than their counterparts in OTS, because the data is encrypted to the shared public key of the DKG, which is constant regardless of the SC size. We can see that the costs of read and write transactions are almost equal as they are dominated by adding a block to the AC blockchain. Finally, the collective decryption overhead scales linearly to the SC as a threshold t of shares needs to be validated and interpolated, where t increases linearly to the size of the SC.

Write transaction size Table 1 shows the write transaction sizes in OTS and LTS for different SC sizes. In OTS, a write transaction stores three pieces of PVSS-related information, namely encrypted shares, polynomial commitments and NIZK encryption consistency proofs. Since the size of this information is determined by the number of PVSS trustees, write transaction size increases linearly with the size of the SC. In LTS, since SC servers remember a minimal state of their VSS secret shares, write transactions use the shared key of the SC and do not need to store the encrypted shares. As a result, LTS has constant write transaction size.

8.4 Identity Skipchain

For the IdS, we conduct benchmark tests for the access request verification as this operation is the most resource and time intensive. Other functions, such as creation or updates to policies, are less time consuming. We consider two cases for benchmarking: verification of single signature and of multi-signature requests.

Single-signature Request Verification We benchmark the request verification time for single-signature re-

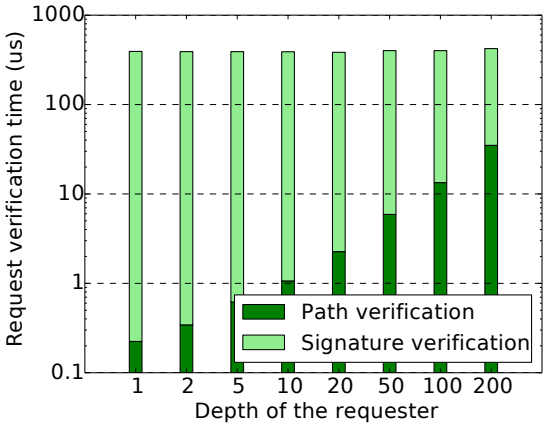

Figure 7: Single-signature request verification.

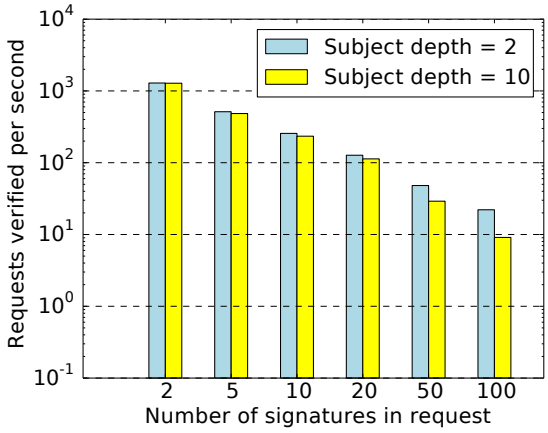

Figure 8: Multi-signature request verification throughput.

quests. The request verification time is the sum of the signature verification and the time to find the path from the target policy to the requester. We vary the *depth* of the requester; this depth refers to the distance between the target policy and the parent policy of the requester. In order to investigate whether access checking has a significant effect on the request verification time. Figure 7 shows the variation in request verification time with depth of the requester. Interestingly, we see that most of the request verification time goes into signature verification. The signature verification takes $\approx 385us$ and accounts for $92.04 - 99.94\%$ of the total time. We observe that even at a depth of 200, a relatively extreme scenario, path finding takes only about 35 *us*.

As signature verification is the major factor contributing to request verification time, it will have an effect on multi-signature requests. To investigate this effect, we create requests with a varying number of signatures and look into the number of request per second we can verify. Figure 8 shows the verification rate for requests with different number of signatures. We show the results for a requester depth of 2 and 10. We see that there is a significant reduction in the number of requests that can be verified when the number of signatures increases. This is expected as the signature verification step is performed for each signature. Finally, the depth of the requester does not play a notable role in the verification rate.

9 Related Work

In the decentralized data sharing topic, the closest work to ours is by Zyskind et al. [45]: they introduce a decentralized personal data management platform that enables users to own and control their data without a trusted third party. Their solution uses a blockchain as the access-control manager for the encrypted user-data that is stored off-chain. The system stores encrypted user-data off-chain, and the access-control policy at the blockchain, which enforces access control before serving data. This is further expanded in Enigma [46], a decentralized computation platform. Enigma provides functionality for users to store and run computations on data with privacy guarantees. MedRec [2], is based on permissionless blockchain implementation and PoW, thus involves transaction fees, and requires involvement into mining and account management processes. On the contrary, Dubovitskaya et al [8] works in a permissioned setting, but uses a cloud-storage breaking our decentralization assumptions. Factom [36] and MedVault [4] also publish medical records on chain, which we assume are privacy-preserving, but have not yet published their blockchain-based design, while SeShare [15] relies on a blockchain based data structure to avoid conflicts in file sharing environments but suffers from performance and usability issues in some of its functionality.

Blockchain-based solutions are also emerging in the IoT ecosystem. Shafagh et al. [33] proposes a blockchain-based solution for providing a distributed and secure data storage system for the IoT domain. It uses a blockchain to provide auditable and decentralized access control for the data that is stored off-chain. Off-chain storage holds the end-to-end encrypted data and when it receives an access request, it checks with the blockchain to enforce the access control. ChainAnchor [13] enables the owner of an IoT device to verify the manufacturing provenance of the device without requiring a trusted third party, and to anonymously register its device by using a permissioned blockchain.

In the identity management area, Maesa et al. [23] outline a blockchain-based access-control system that uses the Bitcoin blockchain and XACML policies. BBDS [43] describes a framework for secure sharing of

medical records where users are verified before they can create requests for data. However, implementation details and experimental data have not yet been published. Another notable system is FairAccess [29], a framework that aims to provide a blockchain-based access-control system for IoT (Internet of Things) devices. Neisse et al. [27] propose an approach where contracts outlining usage policies are deployed on a blockchain. They outline three possible models for the contracts, based on the use cases. Finally, ClaimChain [22] is a decentralized PKI where users maintain repositories of claims about their own and contacts' public keys. However, it allows transfer of access control tokens, which may result in unauthorized access to the claims. Blockstack [1] leverages the Bitcoin blockchain to provide naming and storage functionality, due to which it performs slowly both in throughput and in latency.

References

[1] ALI, M., NELSON, J., SHEA, R., AND FREEDMAN, M. J. Blockstack: A Global Naming and Storage System Secured by Blockchains. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 181–194.

[2] AZARIA, A., EKBLAW, A., VIEIRA, T., AND LIPPMAN, A. Medrec: Using blockchain for medical data access and permission management. In *Open and Big Data (OBD), International Conference on* (2016), IEEE, pp. 25–30.

[3] BLAKLEY, G. R. Safeguarding cryptographic keys. *Managing Requirements Knowledge, International Workshop on 00* (1979), 313.

[4] BLOUGH, D. M., LIU, L., SAINFORT, F., AND AHAMAD, M. Ct-t: Medvault-ensuring security and privacy for electronic medical records. Tech. rep., Georgia Institute of Technology, 2011.

[5] BONNEAU, J. EthIKS: Using Ethereum to Audit a CONIKS Key Transparency Log. In *Financial Cryptography and Data Security 2016* (2016), Springer Berlin Heidelberg.

[6] DABEK, F., BRUNSKILL, E., KAASHOEK, M. F., KARGER, D., MORRIS, R., STOICA, I., AND BALAKRISHNAN, H. Building peer-to-peer systems with Chord, a distributed lookup service. In *8th Workshop on Hot Topics in Operating Systems (HotOS)* (May 2001).

[7] DANEZIS, G., AND MEIKLEJOHN, S. Centrally Banked Cryptocurrencies. *23rd Annual Network & Distributed System Security Symposium (NDSS)* (Feb. 2016).

[8] DUBOVITSKAYA, A., XU, Z., RYU, S., SCHUMACHER, M., AND WANG, F. Secure and trustable electronic medical records sharing using blockchain. *arXiv preprint arXiv:1709.06528* (2017).

[9] FELDMAN, P. A practical scheme for non-interactive verifiable secret sharing. In *Foundations of Computer Science* (1987).

[10] GENNARO, R., JARECKI, S., KRAWCZYK, H., AND RABIN, T. Secure distributed key generation for discrete-log based cryptosystems. In *Eurocrypt* (1999), vol. 99, Springer, pp. 295–310.

[11] The Go Programming Language, Feb. 2018.

[12] HAMMER-LAHAV, E. The OAuth 1.0 protocol, Apr. 2010. RFC 5849.

[13] HARDJONO, T., AND SMITH, N. Cloud-based commissioning of constrained devices using permissioned blockchains. In *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security* (2016), ACM, pp. 29–36.

[14] HERZBERG, A., JARECKI, S., KRAWCZYK, H., AND YUNG, M. Proactive secret sharing or: How to cope with perpetual leakage. *Advances in CryptologyCRYPT095* (1995), 339–352.

[15] HUANG, L., ZHANG, G., YU, S., FU, A., AND YEARWOOD, J. Seshare: Secure cloud data sharing based on blockchain and public auditing. *Concurrency and Computation: Practice and Experience*.

[16] JAKOBSSON, M. On quorum controlled asymmetric proxy re-encryption. In *Public key cryptography* (1999), Springer, pp. 632–632.

[17] JOURNAL, H. The Benefits of Using Blockchain for Medical Records, Sept. 2017.

[18] KATE, A., HUANG, Y., AND GOLDBERG, I. Distributed key generation in the wild. In *29th International Conference on Distributed Computing Systems* (2009).

[19] KOKORIS-KOGIAS, E., GASSER, L., KHOFFI, I., JOVANOVIC, P., GAILLY, N., AND FORD, B. Managing Identities Using Blockchains and CoSi. Tech. rep., 9th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2016), 2016.

[20] KOKORIS-KOGIAS, E., JOVANOVIC, P., GAILLY, N., KHOFFI, I., GASSER, L., AND FORD, B. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Conference on Security Symposium* (2016).

[21] KOKORIS-KOGIAS, E., JOVANOVIC, P., GASSER, L., GAILLY, N., SYTA, E., AND FORD, B. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. Cryptology ePrint Archive, Report 2017/406, 2017.

[22] KULYNYCH, B., ISAAKIDIS, M., TRONCOSO, C., AND DANEZIS, G. ClaimChain: Decentralized Public Key Infrastructure. *CoRR abs/1707.06279* (2017).

[23] MAESA, D. D. F., MORI, P., AND RICCI, L. Blockchain based access control. In *IFIP International Conference on Distributed Applications and Interoperable Systems* (2017), Springer, pp. 206–220.

[24] MELARA, M. S., BLANKSTEIN, A., BONNEAU, J., FELTEN, E. W., AND FREEDMAN, M. J. CONIKS: Bringing Key Transparency to End Users. In *Proceedings of the 24th USENIX Conference on Security Symposium* (2015), USENIX Association, pp. 383–398.

[25] Mininet – An Instant Virtual Network on your Laptop (or other PC), Feb. 2018.

[26] NAKAMOTO, S. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[27] NEISSE, R., STERI, G., AND NAI-FOVINO, I. A blockchain-based approach for data accountability and provenance tracking. In *Proceedings of the 12th International Conference on Availability, Reliability and Security* (2017), ACM, p. 14.

[28] NIKITIN, K., KOKORIS-KOGIAS, E., JOVANOVIC, P., GAILLY, N., GASSER, L., KHOFFI, I., CAPPOS, J., AND FORD, B. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *26th USENIX Security Symposium (USENIX Security 17)* (2017), USENIX Association, pp. 1271–1287.

[29] OUADDAH, A., ABOU ELKALAM, A., AND AIT OUAHMAN, A. Fairaccess: a new blockchain-based access control framework for the internet of things. *Security and Communication Networks 9*, 18 (2016), 5943–5964.

[30] PAGNIA, H., AND GÄRTNER, F. C. On the impossibility of fair exchange without a trusted third party. Tech. rep., Technical Report TUD-BS-1999-02, Darmstadt University of Technology, Department of Computer Science, Darmstadt, Germany, 1999.

[31] PILKINGTON, M. Blockchain technology: principles and applications. *Browser Download This Paper* (2015).

[32] SCHOENMAKERS, B. A simple publicly verifiable secret sharing scheme and its application to electronic voting. *Lecture Notes in Computer Science* (1999), 148–164.

[33] SHAFAGH, H., BURKHALTER, L., HITHNAWI, A., AND DUQUENNOY, S. Towards blockchain-based auditable storage and sharing of iot data. In *Proceedings of the 2017 on Cloud Computing Security Workshop* (2017), ACM, pp. 45–50.

[34] SHAMIR, A. How to Share a Secret. *Communications of the ACM 22*, 11 (1979), 612–613.

[35] SHOUP, V., AND GENNARO, R. Securing threshold cryptosystems against chosen ciphertext attack. *Advances in CryptologyEUROCRYPT'98* (1998), 1–16.

[36] SNOW, P., DEERY, B., LU, J., JOHNSTON, D., AND KIRBY, P. Factom business processes secured by immutable audit trails on the blockchain. *Whitepaper, Factom, November* (2014).

[37] SWAN, M. *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.

[38] SYTA, E., JOVANOVIC, P., KOKORIS-KOGIAS, E., GAILLY, N., GASSER, L., KHOFFI, I., FISCHER, M. J., AND FORD, B. Scalable Bias-Resistant Distributed Randomness. In *38th IEEE Symposium on Security and Privacy* (May 2017).

[39] SYTA, E., TAMAS, I., VISHER, D., WOLINSKY, D. I., JOVANOVIC, P., GASSER, L., GAILLY, N., KHOFFI, I., AND FORD, B. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *37th IEEE Symposium on Security and Privacy* (May 2016).

[40] THE OFFICE OF THE NATIONAL COORDINATOR FOR HEALTH INFORMATION TECHNOLOGY. Use of blockchain in health it and health-related research challenge, Sep 2016.

[41] TOMESCU, A., AND DEVADAS, S. Catena: Efficient Non-equivocation via Bitcoin. In *38th IEEE Symposium on Security and Privacy* (May 2017), pp. 393–409.

[42] VINCENT DURHAM. Namecoin, 2011.

[43] XIA, Q., SIFAH, E. B., SMAHI, A., AMOFA, S., AND ZHANG, X. Bbds: Blockchain-based data

sharing for electronic medical records in cloud environments. *Information 8*, 2 (2017), 44.

[44] YERMACK, D. Corporate governance and blockchains. *Review of Finance 21*, 1 (2017), 7–31.

[45] ZYSKIND, G., NATHAN, O., ET AL. Decentralizing privacy: Using blockchain to protect personal data. In *Security and Privacy Workshops (SPW), 2015 IEEE* (2015), IEEE, pp. 180–184.

[46] ZYSKIND, G., NATHAN, O., AND PENTLAND, A. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471* (2015).

A PVSS

We follow the protocol in [32]. Let $\mathscr{G}$ be a multiplicatively written cyclic group of large prime of order q with two distinct generators G and g, where the set of non-identity elements in $\mathscr{G}$ is denoted by $\mathscr{G}^*$. Let $N = \{1,\dots,n\}$ denote the set of trustees, where each trustee i has a private key x_i and a corresponding public key $y_i = G^{x_i}$. PVSS runs in four steps:

1) **Dealing the shares:** The dealer chooses a secret sharing polynomial $s(x) = \sum_{j=0}^{t-1} a_j x^j$ of degree $t-1$. The shared secret becomes $S_0 = G^{s(0)}$. For each trustee $i \in \{1,\dots,n\}$, they create an encrypted share $\widehat{s_i} = y_i^{s(i)}$ of the shared secret S_0. Furthermore, they create the polynomial commitments $c_j = g^{a_j}$, for $0 \le j < t$, and a NIZK encryption consistency proof $\widehat{\pi_i}$ for each share s_i, $1 \le i \le n$. The dealer publishes all $\widehat{s_i}$, $\widehat{\pi_i}$, and c_j.

   $\widehat{\pi_i}$ proves that the corresponding encrypted share $\widehat{s_i}$ is consistent. More specifically, it is a proof of knowledge of the unique $s(i)$, which satisfies:

$$A_i = g^{s(i)}, \widehat{s_i} = y_i^{s(i)}$$

   where $A_i = \prod_{j=0}^{t-1} {c_j}^{i^j}$. In order to generate $\widehat{\pi_i}$, the dealer picks at random $w_i \in \mathbb{Z}_q$ and computes:

$$a_{1i} = g^{w_i}, a_{2i} = y_i^{w_i},$$
$$C_i = H(A_i, \widehat{s_i}, a_{1i}, a_{2i}), r_i = w_i - s(i)C_i$$

   where H is a cryptographic hash function, C_i is the challenge, and r_i is the response. Each proof $\widehat{\pi_i}$ consists of C_i and r_i, and it shows that $\log_g A_i = \log_{y_i} \widehat{s_i}$

2) **Verification of the shares:** Each trustee i verifies their encrypted share $\widehat{s_i}$ against the corresponding NIZK encryption consistency proof $\widehat{\pi_i}$ to ensure the validity of the encrypted share. To do so, they first compute $A_i = \prod_{j=0}^{t-1} {c_j}^{i^j}$ using the polynomial commitments c_j, $0 \le j < t$. Then, using $y_i, A_i, \widehat{s_i}$, and r_i, they compute:

$$a'_{1i} = g^{r_i} A_i^{C_i}, a'_{2i} = y_i^{r_i} \widehat{s_i}^{C_i}$$

   Finally, they check that $H(A_i, \widehat{s_i}, a'_{1i}, a'_{2i})$ matches the challenge C_i.

3) **Decryption of the shares:** If their share is valid, each trustee i creates the decrypted share $s_i = (\widehat{s_i})^{x_i^{-1}}$ and the corresponding NIZK decryption consistency proof π_i. π_i proves that s_i is the correct decryption of $\widehat{s_i}$ by showing that $\log_G y_i = \log_{s_i} \widehat{s_i}$.

4) **Reconstructing the shared secret:** If the trustees can pool at least t correctly decrypted shares, they use the Lagrange interpolation to recover the shared secret S_0.

B Full Encryption/Decryption Protocol for LTS

We follow the protocol described in [35]. As before we have a group G of prime order q with generators g and $\bar{g}$. We assume the existence of two hash functions: $H_1 : G^6 \times \{0,1\}^l \to G$ and : $H_2 : G^3 \to \mathbb{Z}_q$

A user that wants to encrypt a message under the collective public key X that can be decrypted by anyone included in label $L \in \{0,1\}^{l}$[1] does the following steps:

1. Choose a key k to symmetrically encrypt the message and then embed k to a point $m \in G$.
2. Choose at random $r, s \in \mathbb{Z}_q$. Compute:

$$c = X^r m, u = g^r, w = g^s, \bar{u} = \bar{g}^r, \bar{w} = \bar{g}^s,$$
$$e = H_1(c, u, \bar{u}, w, \bar{w}, L), f = s + re$$

The ciphertext is $(c, L, u, \bar{u}, e, f)$.

Decryption Share Creation Given a ciphertext $(c, L, u, \bar{u}, e, f)$ and a matching authorization to L, a decryption server does the following:

1. Checks if $e = H_1(c, u, \bar{u}, w, \bar{w}, L)$, where $w = \frac{g^f}{u^e}, \bar{w} = \frac{\bar{g}^f}{\bar{u}^e}$ which is a NIZK proof that $\log_g u = \log_{\bar{g}} \bar{u}$

[1] Later this Label is the identifier of an Identity Blockchain

2. If the checks are valid server i choose $s_i \in \mathbb{Z}_g$ at random, and computes:

$$u_i = u^{x_i}, \hat{u}_i = u^{s_i}, \hat{h}_i = g^{s_i},$$
$$e_i = H_2\left(u_i, \hat{u}_i, \hat{h}_i\right), f_i = s_i + x_i e_i$$

3. Outputs (i, u_i, e_i, f_i).

Note that if the label L has changed e is not computed correctly. Given that an adversary will not know r, he cannot change the e to match his new label.

Share reconstruction at the client

1. Share Verification. First the client runs the decryption share check to make sure that the server is not misbehaving. If the check passes then he checks that (u, u_i, h_i) is a DH triple, by checking if: $e_i = H_2\left(u_i, \hat{u}_i, \hat{h}_i\right)$, where $\hat{u}_i = \frac{u^{f_i}}{u_i^{e_i}}, \hat{h}_i = \frac{g^{f_i}}{h_i^{e_i}}$

2. Combining Shares. Assuming that the client has a set of t decryption shares (i, u_i) the recovery algorithm is doing Lagrange interpolation of the shares:

$$X^r = \prod_{k=0}^{t} u_i^{\lambda_i}$$

, where λ_i is the i^{th} Lagrange element

3. Decrypt. The client computes the inverse of X^r and finds $m = \frac{c}{X^r}$. From m he can derive k and symmetrically decrypt the original message.

Share reconstruction at an untrusted Server The client authenticates himself using his public key g^{x_c} One of the decryption servers is assigned to do the reconstruction for the client.

1. ElGamal Encryption. Each Server that created his decryption share as $g^{rx_i} = u_i$ ElGamal encrypt the share for the client using x_i as the blinding factor instead of a random r' the new share looks like : $g^{rx_i} g^{x_c x_i} = g^{(r+x_c)x_i} = g^{r'x_i} = u'^{x_i} = u'_i$. Then he computes $\hat{h}_i$, as before and $\hat{u}'_i = u'^{s_i}$. Finally $e'_i = H_2\left(u'_i, \hat{u}'_i, \hat{h}_i\right)$ and $f'_i = s_i + x_i e'_i$

2. Share combination. Any untrusted server can pool the shares and reconstruct the secret with Lagrange interpolation as shown above. The end result would be $g^{r'x} = g^{(r+x_c)x}$

3. Secret decryption. The client gets $g^{(r+x_c)x}$ and as he knows g^x and x_c he can find $-x_c$ and compute $g^{x^{-x_c}} = g^{-xx_c}$. Finally he computes $g^{rx} = g^{(r+x_c-x_c)x}$ and decrypts as above.

C SCARAB in Real Applications

C.1 Access Control on Encrypted Documents

The first application is a simple decentralized document-sharing system on SCARAB: it enables organization N to share a document S with organization A.

Problem Definition Organization N wants to share a confidential document S with organization A. Within N and A, there is a "confidential" clearance level and parts of S (if not all of it) are meant to be secure. There exist employees that have clearance and departments whose employees automatically obtain access when hired and lose it when they resign. We want to enable the mutually distrustful N and A to share the document and to enforce dynamic access on parts of the document on a per user basis, while each access is transparently logged and available for auditing.

Solution with SCARAB Organizations N and A agree on a mutually trusted cothority configuration. This cothority can include individual trustees that are controlled by N and A. Each organization establishes a *confidential* federated IdS. The hashes of the IdS's genesis blocks are posted on the AC managed skipchain. Inside each chain, any key is solely used for this chain to prevent the linking of identities.

When Organization N generates document S, each paragraph is labeled either confidential or unprotected. Afterwards N generates a write transaction for the confidential symmetric key, including in the label of the transaction the respective IdS of A. The actual document S is encrypted under the symmetric key paragraph by paragraph, then it is shared with organization A. Any authorized user of organization A can retrieve document S and create a proof-of-inclusion to the confidential IdS. This proof is attached to the read transaction, and the cothority can atomically log it and deliver the encryption key. Organization A is free to evolve the confidential IdS, as well as A's employees are free to evolve their individual IdSs.

C.2 Medical Data Sharing

Problem Definition Patient P has his medical data in multiple hospitals that refuse to share them because of potential privacy breaches. P, however, wants to grant access to a medical research team to use his data for their studies. We want to enable P to remain sovereign over his data and to be able to share. At the same time, we want the hospitals to be able to show that P shared the data and that they are not responsible for a privacy breach.

Solution with SCARAB We designed a simple medical-data sharing application on SCARAB that en-

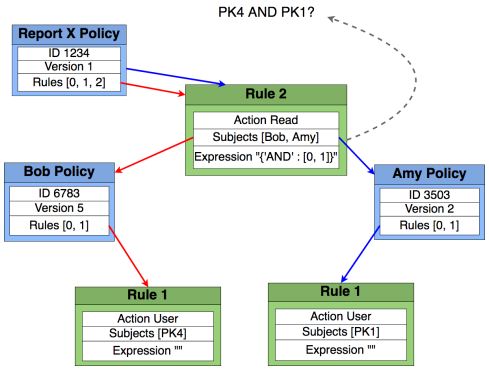

Figure 9: Verifier's path checking for multi-signature requests.

ables P to share his data with multiple potential readers over time. The main difference from the documents-sharing application is that the data generator (hospital) and the data owner (P) are different. For this reason, we use the PolS data-structure as a dynamically evolving indicator for P to the hospitals on how they should enable access on his data.

SCARAB enables P to initialize a PolS when registering with his social security number, which will enable dynamic access control on his medical data. Every time he wants to share data he (1) adds the identity or public key of the decryptor in his dynamic identity and (2) gives a pointer to his medical PolS to the potential reader, hence he can generate the appropriate inclusion proofs.

When new data are generated, the data creator (*e.g.,* hospital) creates a new write transaction where the label is the medical PolS of P, and the hospital stores the data in an encrypted form. But P can authorize decryption requests, thus remaining sovereign over his medical data. P keeps privately a log of all these write transactions. He can point anyone she wants to them in his medical identity, enabling them to access his data by issuing read transactions. This scheme could be further enhanced with payments towards P for sharing his data.

D Access Requests and Verification

In this section, we outline how we create and verify access requests. A request consists of the policy and the rule invoked that permits the requester to perform the action requested. There is also a message field where extra information can be provided *e.g.,* a set of documents is governed by the same policy but the requester accesses one specific document.. A request Req is of the form: $Req = [ID_{Policy}, Index_{Rule}, M]$, where ID_{Policy} is the ID of the target policy outlining the access rules; $Index_{Rule}$ is the index of the rule invoked by the requester; and M is a message describing extra information.

To have accountability and verify that the requester is permitted to access, we use signatures. The requester signs the request and creates a signature consisting of the signed message and the public key used. A request signature's Sig_{Req} form is: $Sig_{Req} = [R_{SK}, PK]$, where R_{SK} is the Req signed by the requester's signing key, SK, and PK is the corresponding public key. An access request consists of the request and the signature, (Req, Sig_{Req}).

On receiving an access request, the verifier checks that the R_{SK} is present and correct. The verifier then checks that there is a valid path from the target policy, ID_{Policy}, to the requester's public key, PK. This could involve multiple levels of checks, if the requester's key is not present directly in the list of *subjects* but included transitively in some federated IdS that is a *subject*. The verifier searches along all paths (looking at the last version timestamped by AC) until the requester's key is found.

Sometimes, an access request requires multiple parties to sign. Conditions for multi-signature approval can be described using the *expression* field in the rules. An access request in this case would be of the form $(Req, [Sig_{Req}])$ where $[Sig_{Req}]$ is a list of signatures from the required-for-access parties. The verification process is similar to the single signature case.

Figure 9 shows an example of the path verification performed by the verifier. Report X has a policy with a Rule granting read access to Bob and Amy. There is an expression stating that both Bob's and Amy's signatures are required to obtain access. Hence, if Bob wants access, he sends a request $(Req, [Sig_{Req,Bob}, Sig_{Req,Amy}])$, where $Req = [1234, 2, \text{“}ReportX\text{”}]$, $Sig_{Req,Bob} = [R_{SK4}, PK4]$ and $Sig_{Req,Amy} = [R_{SK1}, PK1]$. The verifier checks the paths from the policy to Bob's $PK4$ and Amy's $PK1$. Paths are shown in red and blue respectively. Then the expression AND : [0,1] is checked against the signatures. If all checks pass, the request is considered to be verified.

D.1 JSON Access Control Language

A sample policy for a document, expressed in the JSON based language, is shown in Figure 10. The policy states that it has one Admin rule. The admins are S1 and S2 and they are allowed to make changes to the policy. The Expression field indicates that any changes to the policy require both S1 and S2's signatures.

```
{
    "ID" :  2345
    "Version" : 1,
    "Rules" :
    [
        {
            "Action" : "Admin",
            "Subjects" : [S1, S2],
            "Expression" : "{'AND' : [S1, S2]}"
         }
    ]
}
```

Figure 10: Sample Policy in JSON access control language.