QUILIBRIUM: A PEER-TO-PEER MPC PLATFORM AS A SERVICE

A PREPRINT

Cassandra A. Heart *
Quilibrium, Inc.
cassandra@quilibrium.com

December 25, 2022

ABSTRACT

A decentralized platform as a service would enable applications to live in a permanent, uncensorable state, without the ability to be deplatformed by a service provider. Running an application on a tertiary device is implicitly a key component, however the reason cloud computing has worked is a sale of trust – trust that the provider will keep the application and the data being processed private. Verifiable computation allows applications to run on untrusted environments and ensure that the application did behave as expected, bringing the data and its computation closer to the requestor, however does not explicitly safeguard the data, nor does it make the application uncensorable. Fully homomorphic encryption processes solve for both the data privacy and verifiability aspects, but does not necessarily prevent all forms of censorship, nor does it explicitly provide repudiability or nescience to the application being served or the requestor. We propose a new protocol which addresses these deficits, solving the trilemma of privacy, verifiability, and censorship resistance using a peer-to-peer network. The network functionally serves as an oblivious sharded hypergraph database, in which nodes are blind to whether their participation enabled a query to succeed, what data was queried, the contents thereof, or the requestor which initiated the query. We further extend the capability of this database with an operating system running on top of it which provides familiar services upon which applications may be built.

Keywords MPC · P2P · E2EE · PFS

*https://www.quilibrium.com

Contents

1 Introduction

Quilibrium is a new decentralized network model which leverages techniques that differ from common block chain constructions. Directly, it does not build consensus around a block chain, but indirectly, a block chain or any other data structure may be maintained within the shards of the hypergraph.

Figure 1: A simplified depiction of a block chain

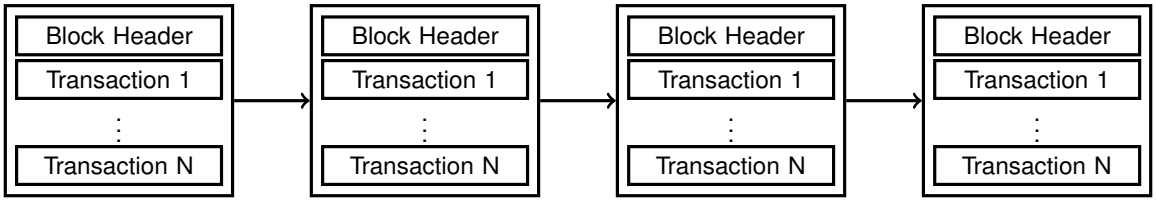
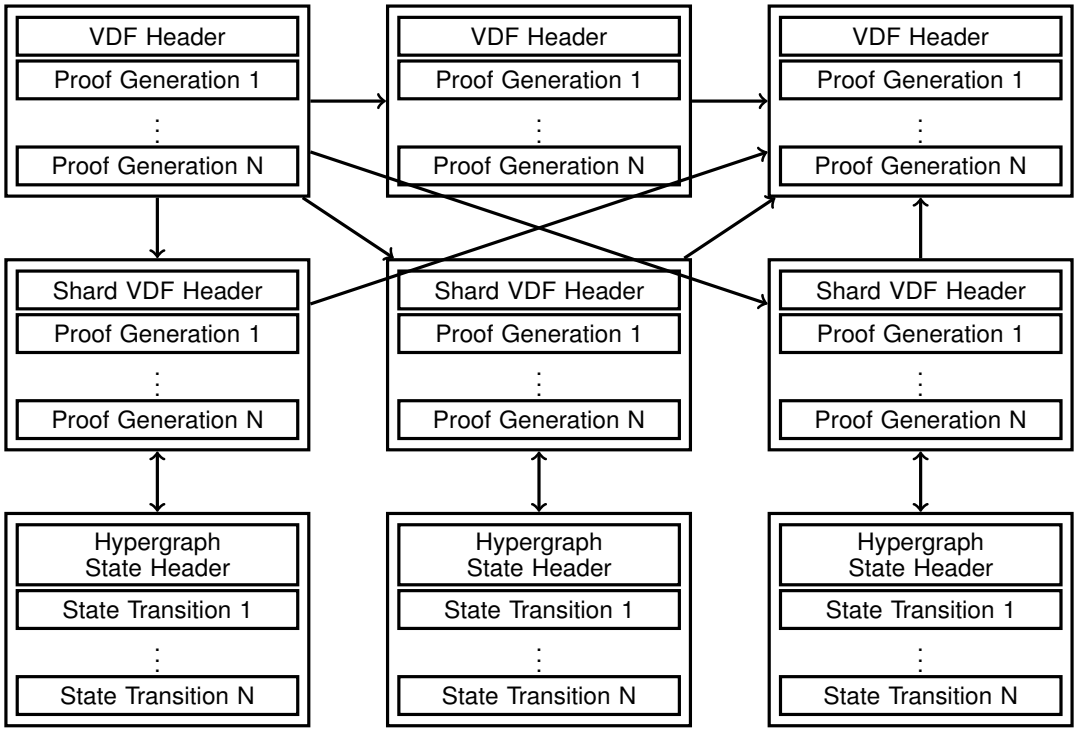


Figure 2: A simplified depiction of a sharded hypergraph network



A major distinction beyond the core data structure also includes the interactivity model preserving complete privacy and anonymity, at the communication level, the query planner level, and the storage level.

To revisit the network behaviors of many block chain-based constructions, the typical behavior of nodes is to be responsible for verifying and validating transactions, and for adding new blocks of transactions to the block chain to maintain the structure:

- Nodes receive new transactions from users on the network and verify that the transactions are valid according to the rules of the protocol. This typically involves validating a signature and that the transaction does not violate any rules or constraints, such as double-spending or invalid execution of code.
- Once a node has verified a transaction, it broadcasts the transaction to other nodes on the network, who also verify the transaction and add it to their own local copy of the block being created.

- Depending on the rules of the protocol, either based on meeting a sufficient difficulty bar in calculating a hash for the block, or being the leader node for the network for that block interval, it creates a new block of transactions and adds it to the block chain. Regardless of consensus method, it typically involves calculating a cryptographic hash for the new block an incorporates it into the header, which serves as a unique identifier for the block and links it to the subsequent block in the chain.
- Once a new block has been added to the block chain, the node broadcasts the block to other nodes on the network, who also verify the block and add it to their own local copy of the block chain.
- In the event there are valid but conflicting blocks, the network follows a fork-choice methodology, typically the pattern of the longest continuing series of blocks in the fork wins. The fork-choice behavior is needfully constrained to be bounded to a probabilistic outcome, in that the likelihood of two viable forks continuing to propagate in length is infeasible beyond a low number of blocks, such that the network may consider all transactions prior to the probabilistic height to be considered finalized.

This approach requires the attribution of significant resources, whether it is computational power, disk space, memory, or wealth incentives in order to assure the security of the protocol, and by consequence is subject to manipulation in the event any of those can be introduced in outsized influence. In the former, this would typically require an outsized influence greater than fifty percent of the network. In the latter-most, this typically is bounded by the wealth being denominated in the token the network issues in reward for following the protocol, with a requirement that of the wealth put at stake, at least two-thirds have agreed to this finalized state.

The central reason of why these networks converge on a block chain under the efforts of this consensus model is specifically to avoid invalid conflicting state, which under data structures with many forward propagating edges to subsequent nodes is otherwise possible due to the transaction and/or state management model not having an unforgeable basis with which it can prevent conflicting state transitions. This ultimately has the downside of bandwidth being constrained to the capacity of block size, whether it is a convergence of state transitions by commitment or the raw transaction data being conveyed.

In summary, the nature of block chain networks produces inherent limitations on scalability due to a reliance on probabilistic consensus schemes to prevent malicious or otherwise adverse behaviors.

1.1 Approach

Quilibrium's network design utilizes cryptographic approaches to provide unforgeably valid transitions of state, such that adverse conditions are structurally impossible, and malicious behaviors are implicitly punishable by removal and reveal.

This article describes the full scope of the network, including full detail of the foundational principles and protocols used, so that while the source material is referenced, this article may be understood without needing to refer to any additional material.

To begin, we will review the communication layer, wherein the privacy-preserving account addressing scheme, the multi-party group broadcast channel construction with perfect forward secrecy, and anonymous routing protocol is described. For storage of blocks of data, first, we review the use of verifiable delay functions, threshold polynomial commitments to enable sharding of the network's storage, the inter-shard gossip protocol, ultimately building the hypergraph network. This hypergraph network state furthermore grants computational privacy through the conversion into an oblivious data structure. First, we describe the process of oblivious transfer as the base primitive, moving into correlated oblivious transfer and its extensions as the underlying principle of how the query planner and query evaluator operates, finally realizing Turing completeness from these operations.

This provides the core of the Quilibrium network, however this article will further elaborate on its utility through the enumeration of the components atop this network which produces a Database Operating System, realizing universal resources, one of which is the fungible unit of reward that provides the means of incentivization for the network to persist and operate. Through this operating system, this article defines

common applications and their design which can be run that demonstrate the flexibility and capability of the network.

2 Communication

The network's communication can be separated out into four components: the Planted Clique Addressing Scheme, Triple-Ratchet Protocol, Shuffled Lattice Routing Protocol, and Gossip Layer. These components are explained in this section in completion to the extent they are isolated, however they are combined in Section 3 to fulfill the construction of a data store for a hypergraph. In Section 4, we further elaborate over use of the PCAS and SLRP sections to realize the oblivious hypergraph structure, completing the base network.

Some degree of familiarity with learning parity with noise, matrix arithmetic, elliptic curve cryptography is required for this section. Additionally, familiarity with computational hardness assumptions is also valuable.

2.1 Planted Clique Addressing Scheme

Undirected graphs are a common structure in discrete mathematics which describes a basic relationship between elements. Graphs are enumerated as a pair of two sets, vertices and edges, where edges are connections between pairs of vertices. A random graph is merely an undirected graph which possesses some number of vertices and edges, which have some probabilistic basis in the likelihood of any two vertices being connected.

Within a graph, a clique is a subset of vertices which are connected to every other vertex in the subset by an edge. Cliques have an interesting property where given the right conditions, it can be extremely difficult to find them. For illustration, review the following graph with a 5-clique: on the left, the vertices in the clique are unmarked. On the right, the vertices in the clique are revealed.

In the above example, it may be easier to determine due to a few obvious factors: some vertices have fewer than four edges and the proportion of clique size to total vertices. If a clique is formed by amending a random graph in a particular way, that clique is said to be a *planted clique*. The planted clique problem is a computational hardness assumption which relates to the inability to distinguish between a random graph from one which has a planted clique in polynomial time.

The computational hardness of finding a planted clique can be leveraged as a type of asymmetric cryptosystem. This is not unheard of, there are a few, albeit uncommon examples of this in existing literature[Kuč91][JP00][Hud16].

In [Hud16], Péter Hudoba describes a variation in which the planted clique problem is paired with the Learning Parity with Noise problem to produce a cryptosystem which is, due to the NP-complete nature of the planted clique problem, believed to be quantum-resistant.

Adopting some common terminology for the sake of further elaboration:

- Ber_{ε}^{n} – Bernoulli distribution, in this case as binary 0 – 1 vectors of length n where ε denotes the probability where the entry is 1.
- U_n – Uniform distribution over binary vectors of length n.
- $G(n, p)$ – Erdös-Rényi random graphs.

The Learning Parity with Noise (LPN) problem is, succinctly:

Given $M \in \mathbb{F}_q^{m \times n}$ and $s \in \mathbb{F}_q^n$, it is computationally infeasible to determine s from $Ms+e$ where $e \leftarrow Ber_{\varepsilon}^{m}$

The encryption algorithm is provided from [Hud16]:

Algorithm 1 Encrypt

1: Generate a private key S, public key G. From G we get the adjacency matrix, M.
2: Choose a random vector $x \leftarrow U_n$ and a random noise vector $e \leftarrow Ber_{\varepsilon}^{n}$, let $b = Mx + e$.
   - To encrypt 0, send the vector b.
   - To encrypt 1, send the vector b with its last bit flipped.

Algorithm 2 Decrypt

1: To decrypt $y \in \{0, 1\}^n$, output $\sum_{i \in S} y_i \pmod 2$.

Algorithm 3 Key Generation

1: Choose a random $G \leftarrow G(n, p)$ graph.
2: Choose a random k sized subset from the nodes of the graph containing the last row. Denote it with $S \subset [n]$.
3: Remove all edges between nodes contained in S: replace E by $(E \setminus \{(u, v) | u, v \leftarrow S\})$ (plant an independent set to the positions corresponding to S).
4: Iterate through $\{u \in V \setminus S \| \Pi_G(u) \cap S| \equiv 1 \pmod 2\}$ with u in random order (a) with p_{add} probability add (u, v) for $v \leftarrow S \setminus \Pi_G(u)$ to E, (b) else remove (u, v) for $v \leftarrow \Pi_G(u)_{|S}$ from E.

The resulting public key is the graph G's adjacency matrix M, and the private key, S.

Hudoba further notes the encryption algorithm can be extended to more than single bit encryption, and indeed – Ring-LPN constructions exist [HKL+12], which we will use to realize a decryption mechanism. The addressing scheme follows:

Algorithm 4 Address Derivation

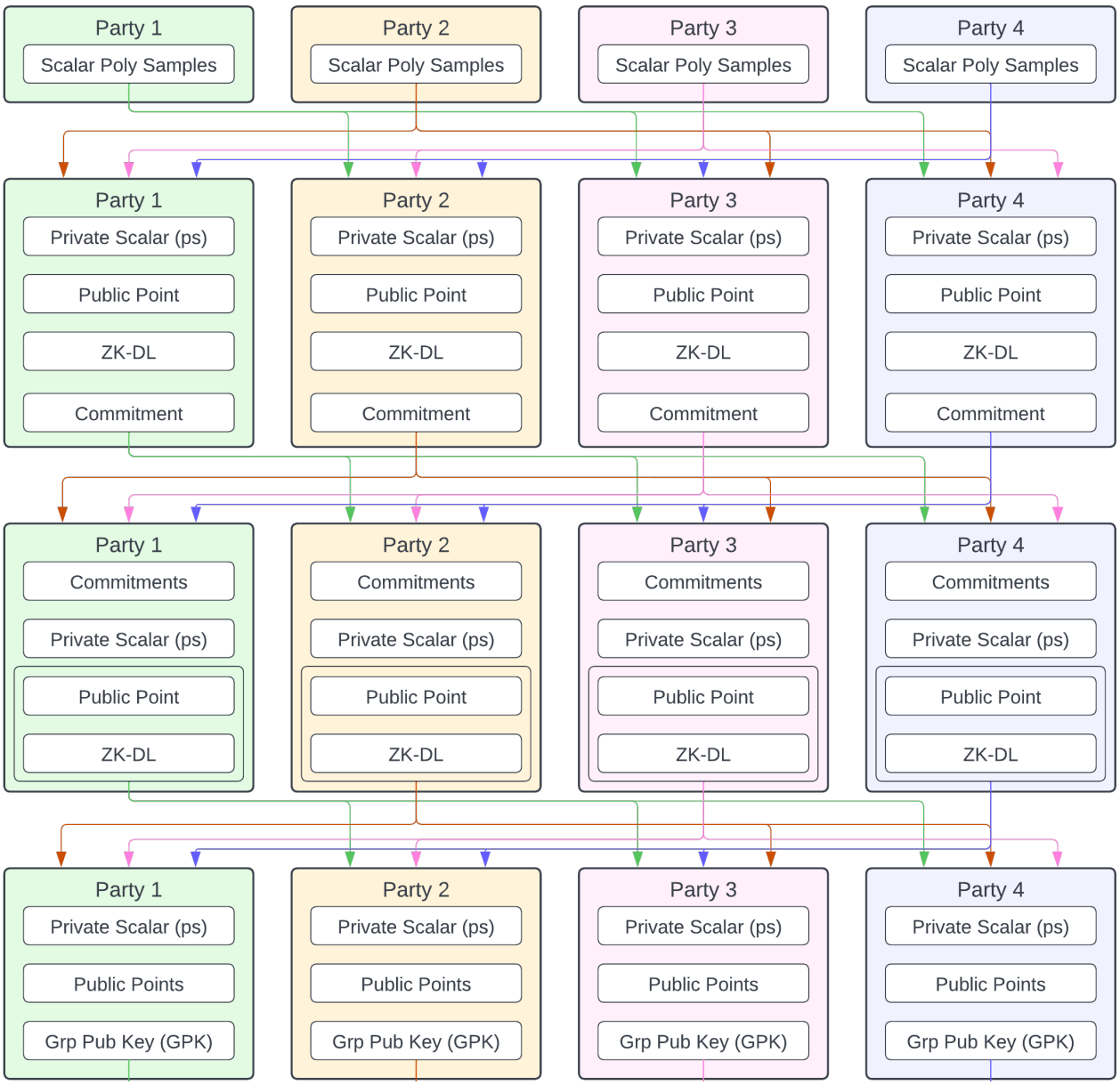
1: Given M, serialized as a binary string m
2: Assume $H(m)$, where H is the hash function cSHAKE with the ASCII customization string "Quilibrium Address" as the domain separator, with output length of 256 bits.
3: Derive the address A as $A = H(m)$.

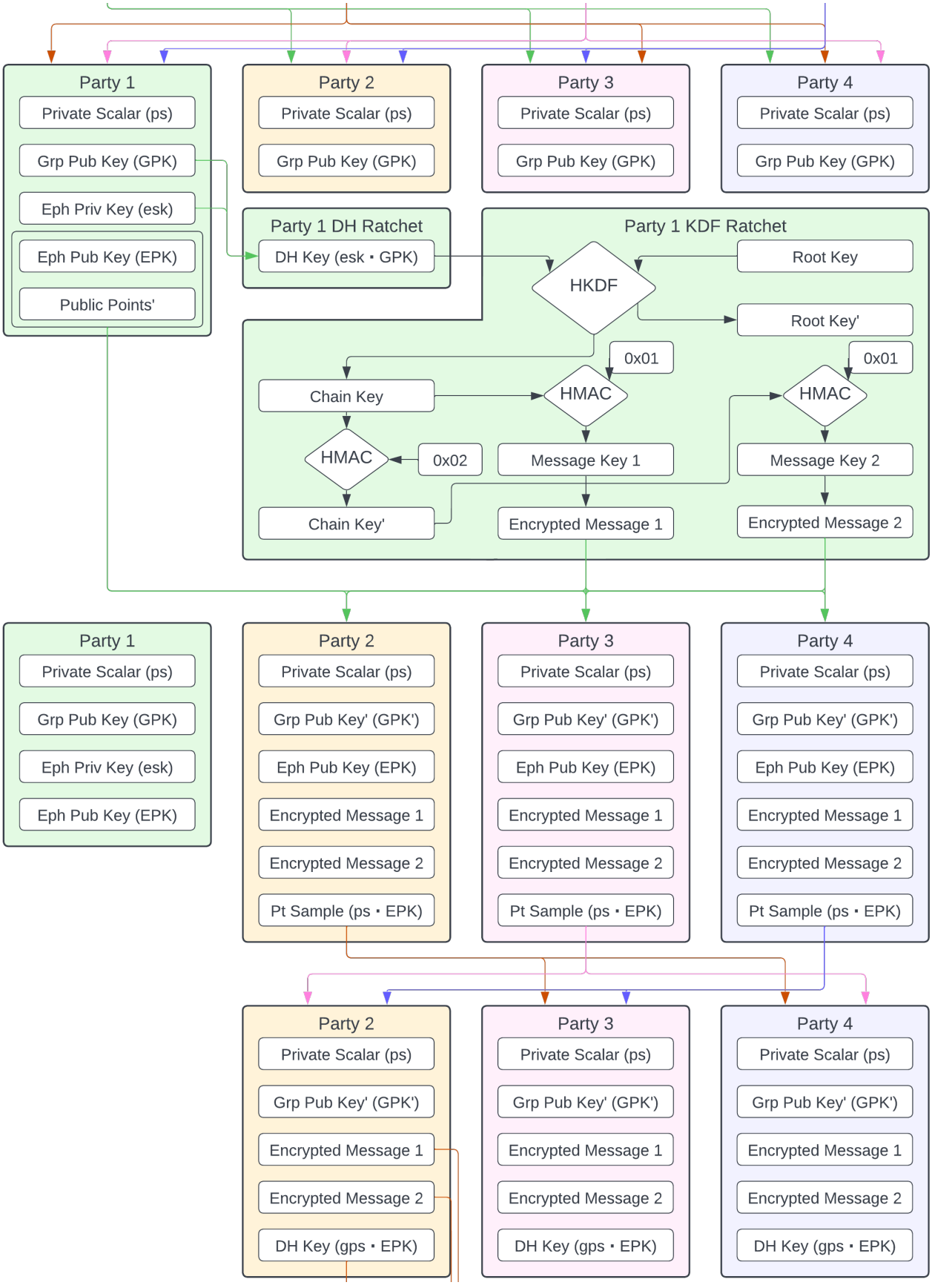
For visual distinction of this resulting output, the preferred serialization is the ASCII encoding of the output bytes prepended with an address identifier: $'Q' \| 'x' \| encode(A)$. To provide convenience in ensuring proper address entry, a checksumming algorithm may be used, but it is strongly encouraged that public user interfaces which interact with addresses do so by indirection via name resources, described in Section 5.7.

2.2 Triple-Ratchet Protocol

The Triple-Ratchet protocol is an extension to the Double-Ratchet protocol[Mar16], utilizing an asynchronous DKG ratchet to provide a group key as the counterparty receiver key plugged into the Double-Ratchet algorithm's Diffie-Hellman process. Future extensions to transition to fully quantum-resistant schemes will include migrating the DKG and DH ratchets to threshold Kyber, however this is not planned at this time.

As a 3-of-4 threshold scheme, the protocol flow is depicted as follows:




Party 1
Private Scalar (ps)
Grp Pub Key (GPK)
Eph Priv Key (esk)
Eph Pub Key (EPK)
Public Points'
Party 2
Private Scalar (ps)
Grp Pub Key (GPK)
Party 3
Private Scalar (ps)
Grp Pub Key (GPK)
Party 4
Private Scalar (ps)
Grp Pub Key (GPK)
Party 1 DH Ratchet
DH Key (esk • GPK)
Party 1 KDF Ratchet
HKDF
Root Key
Root Key'
0x01
0x01
Chain Key
HMAC
HMAC
HMAC
0x02
Message Key 1
Message Key 2
Chain Key'
Encrypted Message 1
Encrypted Message 2
Party 1
Private Scalar (ps)
Grp Pub Key (GPK)
Eph Priv Key (esk)
Eph Pub Key (EPK)
Party 2
Private Scalar (ps)
Grp Pub Key' (GPK')
Eph Pub Key (EPK)
Encrypted Message 1
Encrypted Message 2
Pt Sample (ps • EPK)
Party 3
Private Scalar (ps)
Grp Pub Key' (GPK')
Eph Pub Key (EPK)
Encrypted Message 1
Encrypted Message 2
Pt Sample (ps • EPK)
Party 4
Private Scalar (ps)
Grp Pub Key' (GPK')
Eph Pub Key (EPK)
Encrypted Message 1
Encrypted Message 2
Pt Sample (ps • EPK)
Party 2
Private Scalar (ps)
Grp Pub Key' (GPK')
Encrypted Message 1
Encrypted Message 2
DH Key (gps • EPK)
Party 3
Private Scalar (ps)
Grp Pub Key' (GPK')
Encrypted Message 1
Encrypted Message 2
DH Key (gps • EPK)
Party 4
Private Scalar (ps)
Grp Pub Key' (GPK')
Encrypted Message 1
Encrypted Message 2
DH Key (gps • EPK)

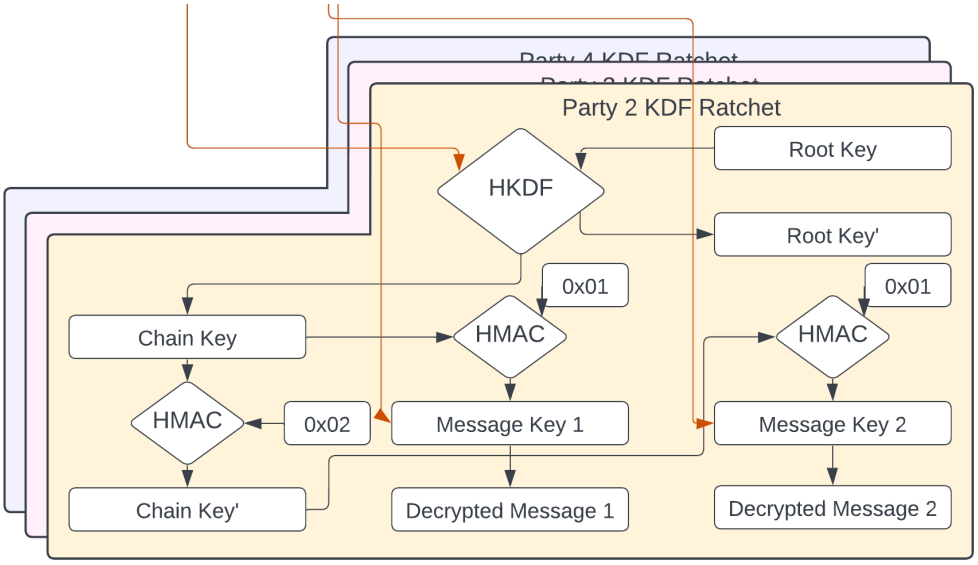


Figure 3: A Simplified Depiction of 3-of-4 Triple Ratchet

To explain this protocol we will use the following terms:

- n – Party Count - The number of parties involved in a multiparty scheme.
- t – Threshold - The number of parties required to reach cryptographic quorum.
- $\mathbb{Z}_q$ – Prime Field - A field where the order is prime.
- $\{a, b, \dots\} \in \mathbb{Z}_q$ – Field Element - Member elements of a field, such as elliptic curve points.
- $\mathbb{G}$ – Group - A cyclic group of order q.
- G – Generator - A chosen field element under a cyclic group which forms a subgroup.
- DLA – Discrete Logarithm Assumption - The assumption that computing discrete logarithms in a group is computationally infeasible. This assumption only applies in certain conditions, such as special classes of elliptic curves over finite fields.
- DDH – Decisional Diffie-Hellman - The computational hardness assumption that the discrete logarithm assumption remains applicable when two randomly sampled scalar factors, multiplied together and raised exponentially to the generator of the group is indistinguishable from a singular randomly sampled field element.
- i, j, k – Node Identifier - The identifier of a given party (i) or the identifier of a party relative to the current party (j, k).
- IDK_i, $sidk_i$ – Identity Key - The unique permanent key pair associated with a given party, with the public key denoted as IDK_i, secret key as $sidk_i$.
- SPK_i, $sspk_i$ – Signed Pre-Key - The unique short-lived key pair associated with a given party that is signed by the party's identity key, with the public key denoted as SPK_i, secret key as $sspk_i$.
- EPK_i, sek_i – Ephemeral Key - The unique ephemeral key pair associated with a given party that is used in a DH ratcheting scheme, with the public key denoted as EPK_i, secret key as sek_i.
- sk, sk_i – DKG Secret - The logical secret key of the distributed key generation process (sk) and the individual shares of that secret held by each party sk_i.
- PK, PK_i – DKG Public Key - The public key output of the DKG process (PK) and the individual sharings of points (PK_i) that collectively, when interpolating p at the y-intercept (y=0) with any subset of t sharings of PK.

- p_i, $p_{i,j}$ – Polynomial - A randomly sampled polynomial of degree $t - 1$ by each party (p_i), with evaluations for another party relative to the current party ($p_{i,j}$).
- H – Hash function - A cryptographic hash function suitable for use as a random oracle.
- z – Zero Knowledge Proof of Knowledge of the Discrete Logarithm (ZKPoK-DL/ZK-DL) - A proof which allows a verifier to confirm the prover possesses knowledge of a scalar exponent without revealing the knowledge.
- c – Commitment - A binding value which confirms a separate value, yet keeps the value hidden until a later phase in which that value is revealed. This is useful when parties need to collectively agree to values which will later be combined, but if any one party revealed their value ahead of others, a malicious party could leverage information from that value and allow them to choose a value that may manipulate the outcome of the combined value in a way to gain an advantage or complete control (Rushing Adversary model).

2.2.1 ZKPoK-DL

In this variant of ZKPoK, we are computing a value relative to a threshold sharing of a logical secret key. Further description of that sharing in of itself is provided in the Secret Sharing and Distributed Key Generation sections, so for this section, simply assume the threshold sharing of the secret and public key (sk_i, PK_i) to be already created.

Algorithm 5 Prove

1: Generate a new random scalar, r, as a private key to an EC keypair (r_i, $R_i = r_i \cdot G$), matching the same curve parameters as the DKG key.
2: To make this process non-interactive, we will apply the Fiat-Shamir heuristic by hashing the serialized threshold sharing's public key concatenated with the random public point: ($ch_i = H(PK_i||R_i)$)
3: To calculate the ZKPoK, we take the threshold secret, multiply it against the integer representation of the challenge, and add the random scalar: ($z_i = sk_i \cdot ch_i + r_i$)
4: Create a commitment to this ZKPoK by taking the hash of the serialized random public point concatenated with the ZKPoK: ($c_i = H(R_i||z_i)$)
5: Broadcast the commitment in the DKG process ahead of the threshold sharing of the public key.

Obtain all commitments, then the ZKPoK, random public points and the threshold sharings of the public key are released. With these values, verify:

Algorithm 6 Verify

1: Reproduce the challenge by hashing the concatenation of the serialized threshold sharing's public key with the random public point: ($ch_j = H(PK_j||R_j)$)
2: Multiply the challenge scalar against the threshold sharing public key, then add the random public point to this point, which should equal the scalar multiplication of the ZKPoK against the generator of the curve: ($Z_j = ch_j \cdot PK_j + R_j$)
3: Multiply the ZKPoK against the generator of the curve and confirm this value equals the previously calculated value, and abort if this does not match: $z_j \cdot G = Z_j$
4: Take the hash of the serialized random public point concatenated with the ZKPoK, and confirm this matches the commitment, and abort if this does not match: $c_j = H(R_j||z_j)$
5: If the values matched, return 1, else 0.

2.2.2 Shamir's Secret Sharing (SSS)

Shamir's Secret Sharing is a technique for encoding a secret in the form of a constant of a randomly sampled polynomial, then distributing evaluations of that polynomial to each participant such that the threshold number of participants in the scheme could perform Lagrange interpolation to reproduce the constant[Sha79]:

Given a threshold of three participants, construct a $t-1$ degree polynomial, randomly sampling coefficients (A, B) from the finite field, setting the constant C as the secret:

$f(x) = Ax^2 + Bx + C$

The dealer of these secret shares then evaluates the polynomial where x is the identifier of the participant (notably, x cannot equal zero as it would simply be handing the participant the secret, and likewise, x cannot be the order of the group either, as $x \pmod q = 0$ where $x = q$).

The dealer distributes these samples to each participant, and when recombining, the participants calculate:

$C = f(0) = \sum_{j=0}^{t-1} y_j \prod_{\substack{(m=0) \\ m!=j}}^{t-1} \frac{x_m}{x_m - x_j}$

2.2.3 Feldman Verifiable Secret Sharing (FVSS)

Feldman Verifiable Secret Sharing is an enhancement to SSS which allows all participants to verify any combination of threshold shares succeed in producing the same value[Fel87]:

When computing the shares from the secret, take the secret as an exponent to the generator G: $s \cdot G = P$, and do the same to all the shares: $s_0 \cdot G = P_0, s_1 \cdot G = P_1, ...$

Distribute the public values with the secrets to each participant, and distribute P. All participants may do Lagrange interpolation of the polynomial with the public values of the shares (Shamir in the Exponent) like before, this time iterating through all participants. The resulting output, if the dealer did not cheat, will equal P for all combinations of threshold participants. If the dealer did cheat, some or all of the combinations will not equal P.

2.2.4 Distributed Key Generation

Leveraging the functions of FVSS and ZK-DL, all parties will engage in the following process:

Algorithm 7 Distributed Key Generation (DKG)

1: Perform SSS locally and send unique evaluations of the polynomial to each party.
2: Sum all received fragments to the local scalar secret key sk_i, and public point ($sk_i \cdot G = PK_i$). Calculate a ZKPoK-DL against these values, and send the commitment to all parties.
3: Once all parties have received the commitments, each party will reveal their public point (PK_i), their random public point from ZKPoK-DL (R_i), and their ZKPoK itself (z).
4: Once all parties have received these values, each party verifies the ZKPoK-DL outputs, if invalid, aborts, and if valid, then performs the FVSS' Shamir in the Exponent recombination of the public points. If any combination produces a differing public key, abort.

2.2.5 Distributed Diffie-Hellman

Recall that in the DKG's second round, we used the generator G to produce the public point sharings PK_i. Assume DKG has been performed. Any threshold number of parties may now perform Distributed Diffie-Hellman:

Algorithm 8 Distributed Diffie-Hellman

1: Given a target ephemeral public key (EK_j) to perform DH against, calculate $sk_i \cdot EK_j = DH_i$. All threshold parties broadcast their sharing (DH_i).
2: Perform the Shamir in the Exponent recombination, which will result in the output agreed key DH.

2.2.6 Asynchronous Considerations

The problem with the above DKG approach is that the setup phase requires all parties to not only be online at time of key generation, but that they must also all perform each round before any may proceed to the next. While it is feasible in some scenarios to allow a small collection of users to conduct DKG to fit into the "counterparty" receiver side of the DH ratchet with the expectation that they will remain online (typically, this would be in a decentralized service in which at least a quorum of node operators leave their nodes always online), this is inviable for highly asynchronous communicators who may only have one party online at a time. This friction can be reduced by simultaneously reducing the security of the protocol through removal of the ZKPoK requirement, and thus participants merely maintain a cache of polynomials/evaluations ahead of time. Losing the security against Rushing Adversaries may not be problematic for a group chat – indeed, Double-Ratchet itself does not care as the DH ratchet itself is quite susceptible to this attack.

We could simply allow users to emit one-time use polynomial fragment bundles in step one into a logical set of queues, each encrypted to the individual peers, and upon an epoch, all parties dequeue the latest polynomial fragments, perform their summations, and distribute public points (omitting the ZKPoK process) so that local Shamir in the exponent can be performed to derive the new public key. This can now be our process as well for the "room key" of the Diffie-Hellman ratchet. This poses two problems, with solutions that break the original strength of Double-Ratchet:

- A sender, upon first encrypted message send has now revealed to a quorum the shared sending chain key for that sender. The receivers can now forge any message during this epoch from that user. Solutions like MLS resolve the message forgery problem through a combination of trust-oriented components (authentication to the server, authentication in the message), but this removes repudiability.
- It is possible that a malicious client could emit one-time use polynomial fragment bundles that produce different public keys to different parties, bifurcating the room key. The protocol could be amended to not allow a new epoch to proceed until all parties have emitted public points and all parties have collectively agreed on a key, but that brings us back into a relatively synchronous pattern, and not all parties are online (and some may even be lost). This would force a re-initiation, which makes the entire experience slow.

Revisiting the initial setup phase of DKG, we already have the following available to us:

- A list of the participants involved
- A secure peer-to-peer channel
- Relevant keys to the participants (IDK_i, SPK_i)

Let's iterate over a theoretical asynchronous DKG protocol which allows for ratcheting of the room key in a way that any party can reasonably verify no bifurcations may occur, nor permit message forgeries, without losing the Forward Secrecy, Post-Compromise Secrecy or Repudiation/Deniability properties of Double-Ratchet.

2.2.7 Polynomial Verifiable Sharing (PVS)

Recall that in FVSS, we had performed Lagrange interpolation over the set of terms raised to the exponent of the generator (Shamir-in-the-exponent) [Fel87]. Performing the same for the fragments produced by any given party can provide a meaningful verifiable sharing mechanism, extending from the simple EC additive homomorphism.

Algorithm 9 Polynomial Verifiable Sharing (PVS)

1: Given a series of fragment ($frag_{i,j}$) scalars to send to each party, we can calculate a publicly verifiable fragment raised to the exponent of the generator $FRAG_{i,j} = frag_{i,j} \cdot G$ and the secret of the polynomial, raised to the exponent of the generator: $S = s \cdot G$. These public fragments are distributed to all parties.
2: Each party confirms it does not resolve anyone's polynomial fragment to the point at infinity with the negation of the previous fragment and the addition of the new and all fragments shared recombine in the exponent to S.

2.2.8 Asynchronous DKG (ADKG) Ratchet

Now that we have a means to ensure no party can produce invalid fragments, we can adopt a new ratcheting scheme for DKG. Each party will (upon their need to ratchet):

Algorithm 10 Asynchronous DKG Ratchet

1: Individually perform a local FVSS with PVS, and enqueue the output bundles to the respective recipients.
2: When a new bundle is needed from a given party, the other participants will dequeue the bundle, and perform the verification process of PVS. Upon confirmation of verification, each party will recalculate the new shared polynomial, substituting the old fragment with the new to obtain a new sk_i, and each party will send their public point (PK_i).
3: Each party then performs FVSS' Shamir-in-the-Exponent recombination of the public points.

2.2.9 Diffie-Hellman Ratchet

The Diffie-Hellman Ratchet of Signal thus remains roughly the same as before, with the sender role as any sender, the ADKG ratchet as the receiver, and agreement remains one round:

Algorithm 11 Amended Diffie-Hellman Ratchet

1: The threshold number of receivers submit their public points produced by the sender Ephemeral Public Key point multiplied by the receiver's private scalar: $PEPK_i = sk_i \cdot EPK_j$. Each receiver will add a new ADKG ratchet output bundle to the output queues.
2: The threshold number of parties perform Shamir-in-the-Exponent recombination to produce the DH agreement key. This value is fed into the KDF ratchet to produce the receiving chain key. The ADKG ratchet gets bumped by the first receiver to have submitted a public point – this can be done deterministically in a context where message ordering cannot be guaranteed.

2.2.10 Key Derivation Function (KDF) Ratchet

The KDF Ratchet also remains roughly the same, except the initial session key (which becomes the initial root key) is derived from the following for each party:

1. $DH1 = DH(idk_i, PK)$

2. $DH2 = DH(spk_i, PK)$

3. $DH3 = DH(sek_i, PK)$

4. $SK = HKDF(salt, (domain||DH1||DH2||DH3), info)$, where $salt$ is a uniquely different $salt$ from the Double-Ratchet $salt$ of 32 '0x00' bytes – instead, 32 '0x01' bytes, to be distinct from a Double-Ratchet session derivation. The $domain$ separator remains the same as Double-Ratchet, and $info$ is the relevant application name for the use of the protocol.

Algorithm 12 Root Key KDF

1: Given the root key RK_i, and output of Distributed Diffie-Hellman DH, use $HKDF(salt, input, info)$ where $salt$ is RK_i, $input$ is DH, and $info$ is the relevant application name for use of the protocol, and HKDF's output size is the size of the sum of the root key size and the chain key size.

Algorithm 13 Chain Key KDF

1: Given the chain key CK_i, use $HMAC(key, input)$ where key is ck_i, $input$ is a single byte identifier given the key being produced:
   - $0x01$ – message key
   - $0x02$ – next chain key
   - $0x03$ – AEAD key

2.3 Shuffled Lattice Routing Protocol

Anonymity is a hard problem, with a long gradient that encompasses the various risk tolerances people will subject themselves to in exchange for less difficulty: trusting a service to not log your requests, trusting an ISP to not record your traffic (or split it into a surveillance funnel on the behalf of another actor), assuming the majority of nodes are not malicious in a distributed proxy, and so on.

Mixnets have been an area of public research since the early 1980s, starting from David Chaum's paper "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms"[Cha81]. The basic principle is that, given an arbitrary intermediary (typically, a collection of many independent servers aiding in this routing function), a sender can communicate anonymously with a receiver by enveloped encryption, where the inner envelope contains a message destined for the receiver, encrypted using the receiver's encryption key, and the outer envelope is encrypted for the mixnet provider using the mixnet's encryption key. This approach is of course simplistic, and there exist many varied attacks against such a scheme, especially with regards to statistical privacy, depending on the capabilities of an attacker:

- External passive attacks, where the attacker monitors traffic to and from the mixnet operators, and the exact timings of communication that occurred.
- External active attacks, where the attacker introduces their own traffic to the mixnet to enhance the analysis of traffic flow, or disrupt operations to better identify individuals.
- Internal passive attacks, where the attacker operates as a normal mixnet node to decrypt one or more hops of the traffic.
- Internal active attacks, where the attacker operates as a malicious mixnet node to actively drop some amounts of traffic to distinctively identify individuals.

As we have seen with the evidence around KAX17's operations on the Tor network[nus21], the idea that an adversary with substantial resources would engage in significant node operation in order to undermine the privacy of a mixnet is not only a hypothetical consideration, it is a present condition of some networks.

In a traditional client-server model with a mixnet proxy inbetween, even when the server is listening only through the mixnet, a malicious majority can undermine privacy in the interaction. To eliminate this condition, an alternative where malicious majorities cannot influence traffic patterns in meaningful ways must be introduced.

2.3.1 Square Lattice Shuffle

In the article "The Square Lattice Shuffle", Johan Håstad proves a particular process involving t permutations of n elements arranged in a $m \times m$ square reaches a statistically negligible difference from uniform distribution such that[Hås05]:

$$\Delta(\Pi_t, U_n) \leq O(n^{1-\lfloor \frac{t}{3} \rfloor \frac{1}{4}} (\log n)^{\lfloor \frac{t}{3} \rfloor})$$

This process is defined formally in the paper as[Hås05]:

Definition 1. *At each time step m permutations of $[m]$, $(\sigma_i)_{i=1}^m$ each on m elements, are picked independently and and uniformly at random. At even time steps, for $0 \leq i < m$, σ_i is applied to the elements in row i while at odd steps it is applied to column i. Repeating this process for t time steps with independent choices at each point in time creates a random permutation from the distribution Π_t.*

2.3.2 Random Permutation Matrices in a Square Network

The square lattice shuffle process was then adopted and extended in the paper "RPM: Robust Anonymity at Scale", where permutation nodes utilize a MPC-driven computation of a permutation matrix P, trading off in computational complexity of the matrix to a square network of nodes. This tradeoff between algorithmic privacy and statistical privacy enables significantly larger number of messages to be mixed by the network[LK22].

The operation of the network follows three phases:

1. Client message collection – the preparation at the client level requires the creation of a Shamir split of the message m, that can be ordered deterministically by the servers. This is achieved by generating Shamir shares of a random value r given out to the client, recombined by the client, and added to the message $m + r$, blinding it. This message can be sent to the servers, who can subtract their share of r and collectively recombine it in the permutation phase to find m.

2. Server permutation – The aforementioned permutation process is performed with the input vector of blinded messages ($Y = P \cdot (X + R) - PR$). This results in an output vector Y of unblinded messages, in completely shuffled order.

3. Broadcast – All messages are broadcast, to be retrieved by their intended recipients.

Calculation of a permutation matrix through a series of secret sharings first requires the calculation of a Beaver multiplication triple to perform multiplication (denoted as ***Mul(x, y)***).

The offline algorithm of RPM's Variant 2 is described as follows[LK22]:

Algorithm 14 Offline Calculation of Permutation Matrix Sharings

1: All parties generate a $k \times k$ permutation matrix M_i and generates secret sharings, distributing to each party
2: All parties verify the columns and rows of each sharing using sketch checks to verify the sharings correspond to a valid permutation matrix, aborting if a check fails.
3: All parties multiply their received matrix shares and their own share of their matrix, $\langle P \rangle = \langle M_1 \rangle \langle M_2 \rangle ... \langle M_n \rangle$
4: All parties generate k random shares, producing the vector $\langle R \rangle = \{\langle r_1 \rangle, \langle r_2 \rangle, ..., \langle r_k \rangle\}$
5: All parties compute $\langle PR \rangle, = Mul(\langle P \rangle, \langle R \rangle)$

Algorithm 15 Online Matrix Permutation Recombination

1: All parties receive blinded messages using chosen random shares, and are slotted into matching positions of the R vector
2: All parties recombine the input vector, yielding $X + R$.
3: All parties calculate $\langle Y \rangle = \langle P \rangle \cdot (X + R) - \langle PR \rangle$, and then recombine to produce Y.

2.3.3 Shuffled Lattice Routing

The above process of random permutation matrix processing in a square network[LK22] is conducted locally within a processing cluster, coordinated through periodic bitmask broadcasts which indicate the collection

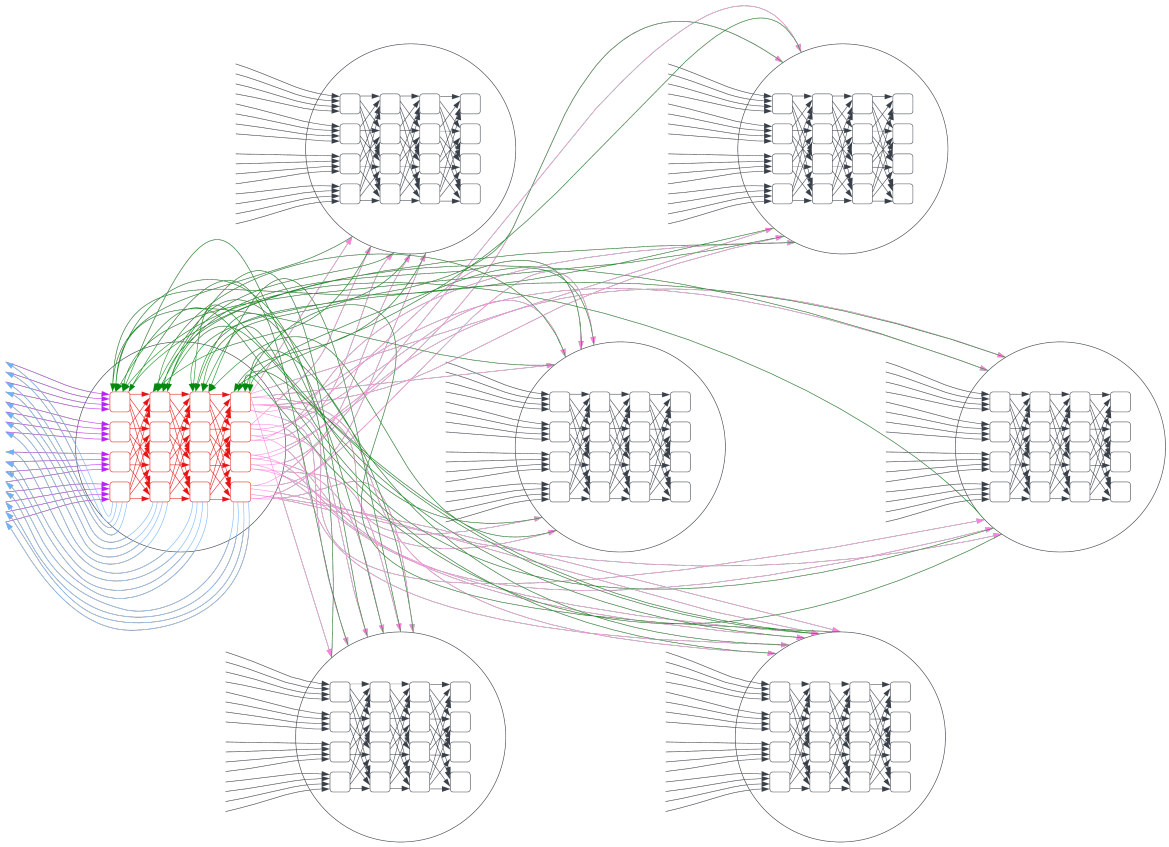


Figure 4: A Depiction of Shuffled Lattice Routing – each circle corresponds to a logical cluster

of addresses the nodes wish to process for. The phases are amended per local cluster to the following, all conducted in parallel, staggered per step, illustrated in Figure 4:

1. Message collection (Purple arrows) – the messages may come from light clients or full nodes, then sorted deterministically.

2. Server permutation (Red boxes) – the Variant 3 with Variant 2 permutation process of RPM is followed, however the permutation matrix shares are then transposed and retained for Phase 4.

3. Broadcast (Pink arrows) – messages are distributed to nodes listening for matching address bitmasks.

4. Processing/Acknowledgement (Green arrows) – messages are confirmed to destination, acknowledgement is returned.

5. Transpose mix (Red boxes/Blue arrows) – The transpose of the permutation matrix shares is followed of the tagged acknowledgements, producing clean acknowledgements in the same order as client requests, allowing anonymous retrieval of confirmation.

Message passing between nodes is communicated following the gossip layer described in Section 2.4.

2.4 Gossip Layer

The final basic element of communication in this network is the message propagation channel. The important characteristics of such a channel are that we need to handle common disruptive attack patterns:

1. Censorship – An attacker attempts to block communication, either between nodes, clients, messages, or destination addresses.

2. Sybil Attacks – An attacker creates many dummy nodes, to perform other types of attacks, or leverage shared information between parties to gain some kind of advantage.

3. Eclipse Attacks – An attacker overwhelms the world state such that it distorts the state accessible by the honest nodes.

2.4.1 GossipSub

In "GossipSub: Attack-Resilient Message Propagation in the Filecoin and ETH2.0 Networks", the authors note the dynamics of various approaches to message propagation and the tradeoffs required to balance high bandwidth utilization of flooding the network with redundant messages, and on the other extreme a gossip driven pub-sub approach. Their contribution brought a new approach that served as a middle ground, with local meshes which engage in local network flooding, meanwhile gossiping announcements of message availability. To balance network utilization, the local meshes would self balance to a small number of connections, using a scoring metric to determine quality of peers[VNM+20]:

$$Score(peer) = TC(\sum_{n=1}^{4} w_{n,t} P_{n,t}) + w_5 P_5 + w_6 P_6$$

The values in this formula are defined as environment-specific weights w_i, and parameter values, P_1 through P_4 are further indexed to the specific topic and capped per TC. The parameters are defined as follows:

1. Time in Mesh for Topic – The time spent as a member of the mesh for the given topic.

2. First Message Deliveries for Topic – The count of times the peer was the first to deliver messages for the topic.

3. Mesh Message Delivery Rate/Failures for Topic – The delivery metric intended to penalize peers engaging in behaviors likely to be explicitly dropping messages.

4. Invalid Messages for Topic – The metric intended to penalize peers delivering application-invalid messages.

5. Application-Specific Score – A deferring score that is application-specific and left to implementation.

6. IP Collocation Factor – The metric penalizing the number of peers connecting from the same IP address, that surpasses an application-controlled threshold.

This approach lends well to a protocol which has transparently validatable messages at time of receipt by a node, however the multi-phase operations of the routing protocol makes this scoring metric impractical, as well as the topic subscription model for a shard model that is not driven by address partitioning. That said, the core structure of local mesh, global mesh, all nodes gossip with some kind of a scoring basis is a pattern worth extending.

2.4.2 BlossomSub

Amending the GossipSub protocol, we replace the topic subscription/broadcast process to not be an exact matching basis, but rather a counting bloom filter based approach, hence the name Blossom, a portmanteau of Bloom-Sum. The scoring system is also amended to take advantage of the message reconstruction protocol ensuring node mesh behaviors are correct, connectivity between nodes is altered to incorporate a circuit construction, and finally time/rate related metrics are instead replaced by generic consumption of unforgeable proofs of valid behavior. These alterations drastically improve privacy (IP collection is useless), remove cheap Sybil attack potential (by virtue of making it very expensive to overwhelm the network), and no longer require message inspection/processing at multiple points, reducing redundant compute costs.

2.4.2.1 Peer Discovery

Peer discovery is achieved through an open peer exchange, where nodes will offer a score-indexed list of peers in terms of ip address and public route keys, or graph address and capabilities.

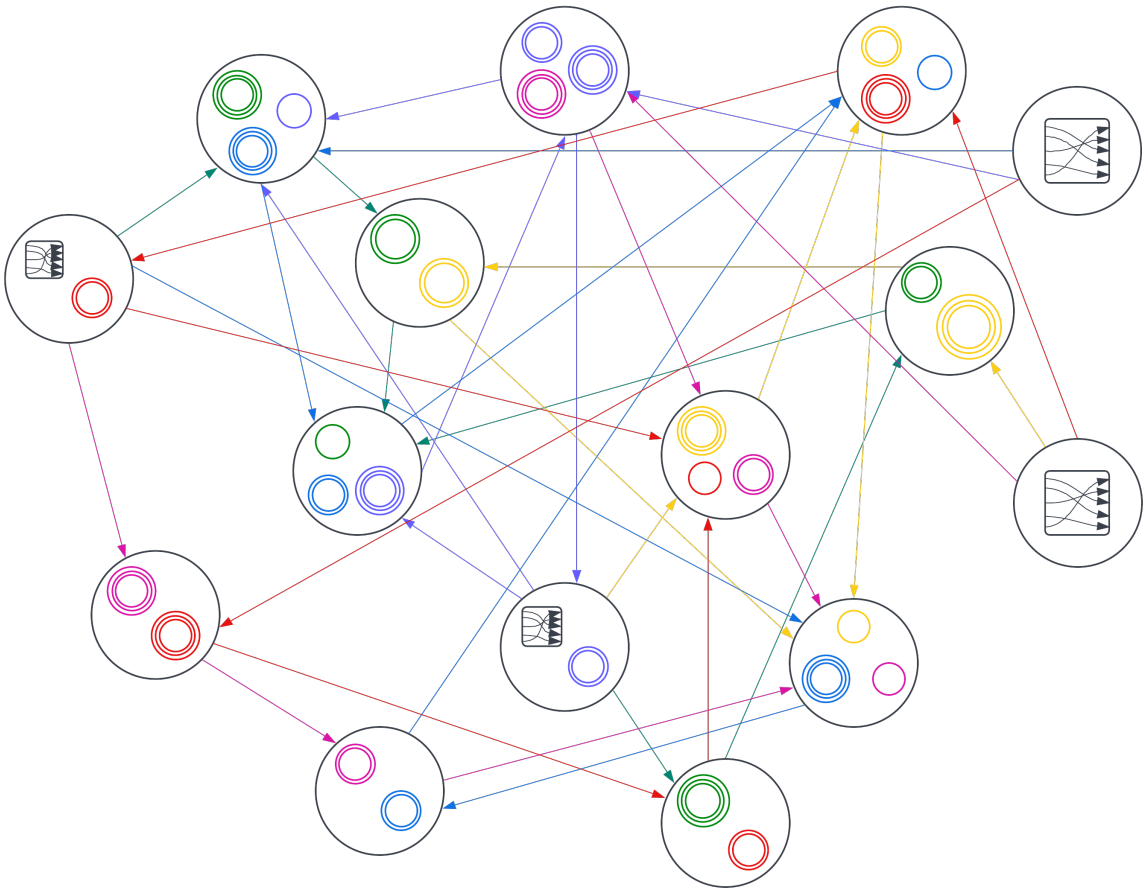


Figure 5: A depiction of a SLRP cluster under BlossomSub – each black circle corresponds to a node, a box with arrows indicates a participant, and 3, 2, and 1 nested circles indicate remaining envelope depth

Node operation capabilities are typically self-adjusting, but in the event a node operator explicitly configures their node to enable/disable certain capabilities (at the impact of their score), only the capabilities the node broadcasts support for will be given.

Algorithm 16 Gossip Channeling Join/Graft

1: A node scans the list for nodes with $ROUTE$ capability.
2: At random, six nodes are picked, and an enveloped 3-depth gossip channel with each is created.
3: The node gossips a bandwidth-adjusted $JOIN[bitmask]$ message (bitmasks are calculated relative to bandwidth, this adjusts periodically) with a fresh random join public key.
4: Nodes processing within that bitmask will gossip a $GRAFT[bitmask]$ message with a fresh random graft public key, exchanged with the gossiped $JOIN$ key. This GRAFT should only occur on one of the gossip channels.
5: Nodes accepting the $GRAFT$ initiate an ad-hoc join of Triple-Ratchet for the cluster, and begin participation in SLRP.

For routing functionality, we can adopt the basic GossipSub scoring mechanism[VNM+20]. To preserve anonymity and unlinkability of application-level functions, the other functions are only associated with a graph address and carry an implicit scoring system.

Algorithm 17 Implicit Node Scoring

1: $V = \sum_{i=1}^{n} r_i v_i$, where V is the aggregate sum of duration of data blocks made available v_i, and the rate at which they were requested r_i. The purpose is to reward long-term retention of data.
2: $A = avg(\{r_1 a_1, r_2 a_2, ..., r_n a_n\})$, where A is the averaged ratio of attested data requests processed:unprocessed, and the inverse rate at which they were requested. The purpose is to reward rare data deliverability.
3: $Score = V + A$

Because these metrics are unforgeable proof broadcasts (See Section 3), nodes can implicitly sweep these into a sorting basis.

3 Block Storage

Block storage in Quilibrium is built as a Verifiable Delay Function (VDF) driven proof of storage, with query processing proof of validity.

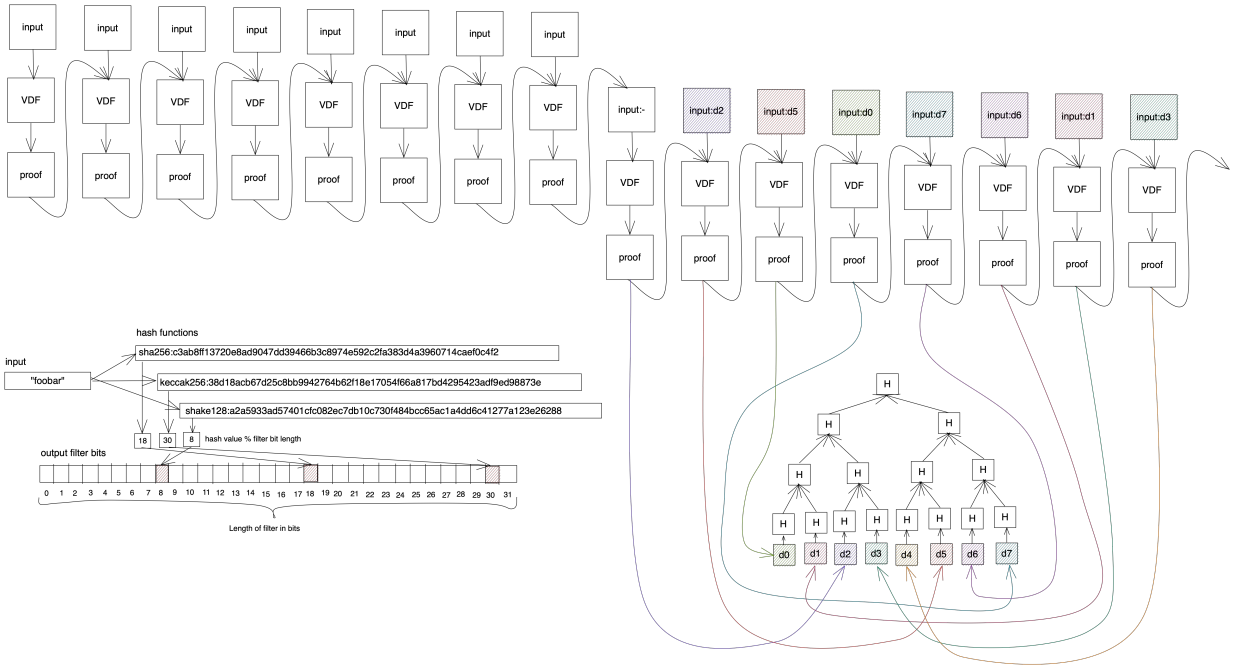


Figure 6: A depiction of a VDF master pulse clock feeding into initial input of a block availability, plus the bloom clock input basis conjoined with Merkle proofs to incorporate in next VDF interval input

3.1 Verifiable Delay Function Timestamping

In "Verifiable Delay Functions", Boneh et. al. introduced a formalization of the behavior of what constitutes a verifiable delay function (VDF), along with some candidate functions. In short, the behavior of a VDF should satisfy three properties[BBBF19]:

1. Sequential – The processing of the function is inherently non-parallelizable.

2. Efficiently Verifiable – Verifying the proof output of evaluation must be sufficiently faster than calculating the proof itself.

3. Unique – It is computationally infeasible to find an output for a given input that collides with another output and proof.

Many constructions of VDFs relate to the problem of squaring a number repeatedly under an algebraic group that is computationally infeasible to shortcut. In a related, earlier construction, Timelock puzzles utilized squaring under an RSA group, however knowledge of the prime factors was sufficient to calculate this efficiently, i.e. a proof of spending this quantity of time is reliant on a trusted setup. Additionally, it is not universally verifiable – the verifier must be aware of the secret state.

3.1.1 Wesolowski VDF

In "Efficient verifiable delay functions", by Benjamin Wesolowski, a construction based around imaginary quadratic cryptography is proposed. Under certain conditions, the class group of an imaginary quadratic field enables a construction of the aforementioned repeated-squaring approach, but is efficiently and universally verifiable[Wes18].

Other networks have adopted the use of this VDF, such as Chia Network, and by consequence of their consensus mechanism being Nakamoto consensus using VDF as a unique proof of holding space over time, the network incentives were strongly aligned to the production of ASICs which implement this VDF as efficiently as possible. We thus have a very well-defined upper bound on the performance of this approach, and node operators wishing to provide as broad a collection of proofs have specialized hardware already meaningfully available to do so. To ensure this compatibility, we have adopted the classgroup arithmetic specified in "Binary quadratic forms" by Lipa Long[Lon20].

Using this VDF, the network initially bootstraps its master pulse clock input over the first hour of network runtime. When the network is first launched, operators will begin mesh construction following the processes of Section 2. Either after an hour is reached, or the network mesh remains sufficiently stabilized, whichever occurs first, the network will collectively perform a global KZG[KZG10] initialization ceremony, and toss the field element, publishing the public parameters. This will serve as the genesis input for the master pulse VDF.

The VDF proofs are gossiped as a bundle every second to the network, serving as the heartbeat for BlossomSub, tagged with the current UTC time of the node creating the proof. Every hour thereafter, the network will evaluate the gossipped proofs against the global delta, as some machines will inherently emit at a slightly faster rate. The iteration count of the VDF is recalibrated to remain aligned to 100ms intervals. Because there is no reward basis attached to computing the master pulse VDF, this recalibration process is not inherently tilted towards the fastest producers, but rather the mean. This is so that VDF proof generation built around ASICs is favored for block storage proofs instead.

To utilize the VDF for block storage, the first frame of the block snapshot will incorporate the VDF proof output from the master pulse clock at time of initiation. Subsequent iterations of the VDF will incorporate the previous iteration's output as a selection modulo block chunk size, choosing a Merkle proof of the block to input as the next iteration.

To form a bond between data pulses and the master pulse clock, at the end of each hour, the polynomial commitments of the current state of the data block are broadcast – the subsequent hour admits time for gossip to be collected from all nodes, and is then merged into the inputs of the master pulse clock. This weaving pattern enables global state reconciliation, however individual clusters will always remain up to date (or participants will lose their reward).

3.2 Bloom Clock Event Capture

As SLRP clusters admit messages which alter the processed blocks structurally, the messages under the capture window are collected as inputs to a bloom clock for the block, and thus the clock filter is joined with the subsequent Merkle proof selection, forming the new input for the data pulse. To attach an intrinsic reward basis, the data pulse clock is additionally run on the same data, for each interval encrypted using a proof reward key exchanged with a public ephemeral key to form a symmetric key, then subsequently the private ephemeral key is revealed as the input of the next pulse.

4 Oblivious Hypergraph

With a network formally constructed that manages arbitrary block storage, we now incorporate the messaging structure that enables arbitrary queryability, expressing this block storage as the data settlement layer of a hypergraph. In the context of oblivious data structures, this particular construction is oblivious in the sense that evaluating queries is unclear to the nodes processing the requests as to whether or not the request evaluated data corresponding to specific edges and vertices of the hypergraph, the requestor is blind to the contents of any additional data served to it, and someone who has a complete ingestion of the network cannot determine any meaningful data about the contents, structure, or relevant processors of the hypergraph. In the context of oblivious transfer-based communication, the oblivious hypergraph is realized utilizing OT based communication.

4.1 Hypergraph Construction

Hypergraphs are a generalization of graphs in which the edges are capable of connecting more than two vertices, and thus are referred to as hyperedges. There are many higher-degree relationships which can be expressed over hypergraphs, such that any variety of database model can be directly expressed over one. This makes a hypergraph a useful tool for representing and querying such data. Performance is not always of course *better* under a hypergraph, but the generalization is valuable in that it lends to a query pattern that is able to be efficiently made oblivious.

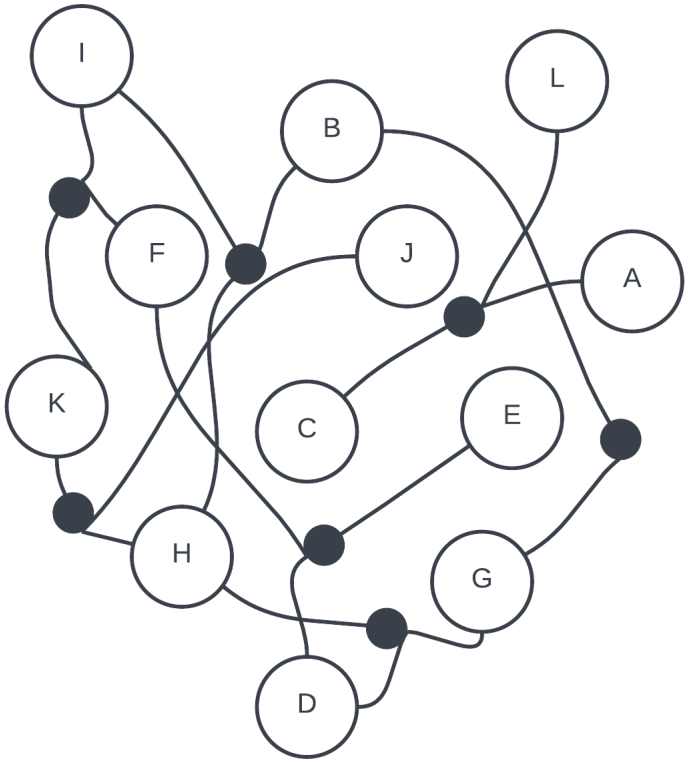


Figure 7: A depiction of a simple hypergraph, where the sets of vertices [F, I, K], [B, H, I], [J, K, H], [D, E, F], [D, G, H], [B, G], and [A, C, L] are all connected by hyperedges.

In the context of Quilibrium, any hypergraph-oriented approach is amenable as the clients can be updated to support them, however to realize most immediately useful functionality in bridging traditional web resources

with decentralized resources, we firstly implement our data mapping strategy as an engine for querying RDF graphs.

4.2 Oblivious Transfer

Oblivious Transfer is a cryptographic technique in which a sender and a receiver engage in a series of messages between one another such that a sender has messages which a receiver can request, the receiver makes a choice, and prepares a response which the sender consumes and applies to the messages being sent but is unaware of the choice made by the receiver, then responds with the choice-applied message data such that upon receipt by the receiver, they are able to consume the intended message, but gain no awareness of the contents of the other messages.

We begin by explaining some base concepts of oblivious transfer protocols such that it becomes easier to follow along in the broader construction.

4.2.1 Simplest OT

In "The Simplest Protocol for Oblivious Transfer", Chou and Orlandi defined a new approach to oblivious transfer in which the construction relied only on the same assumptions as Diffie-Hellman. The approach is summarized below[CO15]:

Algorithm 18 Simplest OT over ECC

1: Given a sender has two messages, m_0, m_1, and a receiver has a choice bit c, both parties sample a random private scalar $x \leftarrow \mathbb{F}_p$ The sender's private scalar will be denoted as a, the receiver's private scalar will be denoted as b.
2: The sender calculates the point $A = a \cdot G$, and sends this to the receiver.
3: If the receiver's choice bit is 0, the receiver replies with $B = b \cdot G$. If 1, the receiver replies with $B = A + (b \cdot G)$.
4: The sender calculates two keys, $k_0 = H(a \cdot B)$, and $k_1 = H(a \cdot (B - A))$, then encrypts m_0 with k_0, m_1 with k_1, and sends both encrypted messages to the receiver.
5: The receiver calculates $k_c = H(b \cdot A)$, and then uses this value to decrypt their chosen message.

Because neither party has the other party's private scalar, provided the discrete logarithm assumption holds, it is impossible for the receiver to calculate the other message's key, and impossible for the sender to calculate the receiver's choice.

Simplest OT is capable of only delivering a singular choice of two messages, and has to be re-evaluated from scratch for each subsequent bit of information learned. Because of this, it is a poor system to directly construct oblivious protocols, however it is exceptionally useful to build on top of as a base OT seed for extension.

4.2.2 Correlated OT

Correlated oblivious transfer is a variant of oblivious transfer where instead of sending a singular choice, the choices themselves are implicitly correlated. The traditional extension approach[IKNP03] involves a Random OT base, where the initial choice is random, and then pseudo-random extensions of the random seeds enable larger correlated constructions with many bits being able to be transferred based on the correlation without the cost of doing a singular OT for every bit. This is utilized in many MPC protocols, and further extended in subsequent papers such as "Actively Secure OT Extension with Optimal Overhead" [KOS15].

At the lower level, correlated OT looks like the following:

Algorithm 19 Correlated OT

1: The receiver obtains a $\Delta \in \mathbb{F}_{2^\kappa}$ from the sender.
2: For every extension:
- Sample $v \leftarrow \mathbb{F}_{2^\kappa}^\ell$. If the sender is corrupted, instead receive $v \leftarrow \mathbb{F}_{2^\kappa}^\ell$ from the adversary.
- Sample $u \leftarrow \mathbb{F}_2^\ell$ and compute $w := v + u \cdot \Delta \in \mathbb{F}_{2^\kappa}^\ell$. If the receiver is corrupted, instead receive $u \leftarrow \mathbb{F}_2^\ell$ and $w \leftarrow \mathbb{F}_{2^\kappa}^\ell$ from the adversary, and recompute $w := v + u \cdot \Delta \in \mathbb{F}_{2^\kappa}^\ell$.

4.2.3 Correlated OT Extension over LPN

In "Ferret: Fast Extension for Correlated OT with Small Communication", the authors contributed improvements to this protocol have sufficiently made COT feasible for general computability at speed. In short, the speed improvements are nearly over 200 times faster per correlation[YWL+20].

For brevity, we list only the functions relevant to Quilibrium's instantiation of Ferret, from the article[YWL+20]:

Algorithm 20 Multi-Point Correlated OT

1: Given a family of efficiently-computable functions $\Phi = \{\Phi_{n,t}\}_{n,t\in\mathbb{N}}$ such that for any $n, t \in \mathbb{N}$ with $t \leq n$, $\Phi_{n,t}$ takes as an input a sorted subset of $[n]$ of size t and outputs another subset of $[m]$ with the same size for some integer $t \leq m \leq n$.
2: The receiver obtains a $\Delta \in \mathbb{F}_{2^\kappa}$ from the sender.
3: For extension, the receiver and sender agree to n, t, and the receiver sends $Q = \{a_0, ...a_{t-1}\}$ where $Q \subseteq [n]$ is a sorted set:
- Sample $v \leftarrow \mathbb{F}_{2^\kappa}^n$. If the sender is corrupted, instead receive $v \leftarrow \mathbb{F}_{2^\kappa}^n$ from the adversary.
- Define an n-sized bit vector $u := \mathcal{I}(n, Q)$, and compute $w := v + u \cdot \Delta \in \mathbb{F}_{2^\kappa}^n$. If the receiver is corrupted, instead receive $u \leftarrow \mathbb{F}_2^n$ and $w \leftarrow \mathbb{F}_{2^\kappa}^n$ from the adversary, and recompute $w := v + u \cdot \Delta \in \mathbb{F}_{2^\kappa}^n$.

4: Compute the set $T = \{\beta_0, ...\beta_{t-1}\} := \Phi_{n,t}(\{\alpha_0, ...\alpha_{t-1}\})$.
5: Wait for the adversary to input m sets $I_0, ...I_{m-1} \subseteq [n] \cup \{-1\}$.
6: Check that $\alpha_i \in I_{\beta_i}$ for all $i \in [t]$ and $-1 \in I_j$ for all $j \in [m] \setminus T$. If the check fails, the process aborts.

Algorithm 21 Deal COT/MPCOT

1: Given a family of efficiently-computable functions $\Phi = \{\Phi_{n,t}\}_{n,t\in\mathbb{N}}$ such that for any $n, t \in \mathbb{N}$ with $t \leq n$, $\Phi_{n,t}$ takes as an input a sorted subset of $[n]$ of size t and outputs another subset of $[m]$ with the same size for some integer $t \leq m \leq n$.
2: To initialize: The receiver obtains a $\Delta \in \mathbb{F}_{2^\kappa}$ from the sender.
3: To COT Extend: Call Correlated OT (Algorithm 19), receive ℓ random COT correlations.
4: To MPCOT Extend: Call MPCOT (Algorithm 20), receive a multi-point COT of length n.

Algorithm 22 Ferret COT Protocol

1: Given LPN parameters (n, k, t) and code generator C such that $C(k, n, \mathbb{F})$ outputs a matrix $A \in \mathbb{F}_2^{k \times n}$.
2: Both parties initialize once, sender samples a uniform $\Delta \in \mathbb{F}_{2^\kappa}$ and both parties invoke the Deal (Algorithm 21) initialization step.
3: Both parties invoke the COT extend functionality of Deal (Algorithm 21), returning $v \leftarrow \mathbb{F}_{2^\kappa}^k$ to sender, $(u, w) \in \mathbb{F}_2^k \times \mathbb{F}_{2^\kappa}^k$ to the receiver such that $w := v + u \cdot \Delta$.
4: To extend:

- The receiver samples $A \leftarrow C(k, n, \mathbb{F}_2)$ and $e \leftarrow \mathcal{HW}_t$, then sends A to the sender. Let $Q = \{a_0, ...a_{t-1}\} \subseteq [n]$ be the sorted indices of non-zero entries in e.
- The sender and receiver invokes the Deal (Algorithm 21) MPCOT functionality, returning $s \in \mathbb{F}_{2^\kappa}^n$ to the sender and $r \in \mathbb{F}_{2^\kappa}^n$ to the receiver, where $r + s = e \cdot \Delta$. If either party aborts, this protocol aborts.
- The sender computes $y := v \cdot A + s \in \mathbb{F}_{2^\kappa}^n$ and the receiver computes $x := u \cdot A + e \in \mathbb{F}_2^n$ and $z := w \cdot A + r \in \mathbb{F}_{2^\kappa}^n$.
- The sender updates vector $v := y[0 : k] \in \mathbb{F}_{2^\kappa}^k$, and outputs a vector $y' := y[k : n] \in \mathbb{F}_{2^\kappa}^l$. The receiver updates vectors $(u, w) := (x[0 : k], z[0 : k]) \in \mathbb{F}_2^k \times \mathbb{F}_{2^\kappa}^k$ and outputs two vectors $(x', z') := (x[k : n], z[k : n]) \in \mathbb{F}_2^l \times \mathbb{F}_{2^\kappa}^l$.

On a 1st gen Intel Xeon Skylake-SP at 3.6GHz with 32GB of RAM, Ferret is capable of generating 60 million COTs per second. AES, for example, requires 2^{19} OTs [ALSZ15], resulting in approximately 120 evaluations of AES within one second, provided sufficient bandwidth (50Mbps).

4.3 RDF to Hypergraph

The construction is described in the paper "Hypergraph Based Query Optimization", which we will use by translating the logic to OT circuits, starting with decryption into the circuit using the extended decryption process of the address content. We adopt their term definitions in this and subsequent two subsections of the paper, their definitions provided here roughly verbatim for convenience[SGDD15]:

1. RDF Graph – $G = (V, E)$ where $V = \{v | v \in S \cup O\}$ and $E = \{e_1, e_2, ...\} \exists e = \{u, v\}$ where $u, v \in V$.

2. Edge Labeling Function – $l_e(S, O) = P$.

3. Node Labeling Function – $l_v(v_t) = t$ where $t \in (S \cup O)$ and $S = Subject(URI \cup BLANKS), P = Predicate(URI), O = Object(URI \cup BLANKS \cup LIT)$.

4. Hypergraph – $H(G) = (V, E)$ where node $V = \{v_1, ..., v_n\}$ and $E = \{e_1, ..., e_n\}$ where $V = \{v | v \in S \cup O \cup P\}$ and each edge E is a non-empty set of V. $\forall P, \exists e | (S_i, O_i) \in H(G)$ where $1 \leq i \leq n$.

5. Overlapping Hyperedge – $(h_i(G) \sqsubseteq h_{i+1}(G))$ where $h_1(G) = (S_1, P_1, O_1)$ and $h_2(G) = (S_2, P_2, O_2)$, $(h_1(G) \sqsubseteq h_2(G))$ iff $\forall s_1 \in S_1 \in h_1(G) \exists s_2 \in S_2 \in h_2(G) \vee \forall o_1 \in O_1 \in h_1(G) \exists o_2 \in O_2 \in h_2(G) \vee \forall p_1 \in P_1 \in h_1(G) \exists p_2 \in P_2 \in h_2(G)$.

6. Predicate-Based Index – $I(G) = (V, E)$ where $V = \{v | v \in P_i \in h_i \wedge \delta\}$ and $E = (v_i, v_j)$ where $v_i, v_j \in V$ and $1 \leq i \leq n - 1, 1 \leq j \leq n$ for $\delta \in V \exists e = (\delta, v)$. δ is the root of the index.

7. SPARQL Query – Q^R contains $< Q^q, Q^s, Q^p >$ where Q^q is the query form and Q^p is the match pattern if $?x \in var(Q^q)$ then $?x \in var(Q^p)$ and Q^s contains constraints like FILTER, OPTIONAL.

8. Query Graph – $Q^G = (V, E)$, $V \leftarrow \{var \in Q_i^p\}$ and $E \leftarrow \{P \in Q_i^p \wedge (var \in Q_i^p, var \in Q_{i+1}^p)\}$ where $1 \leq i \leq n$, n is the number of predicates, P is the predicate.

9. Query Path – $Q^{path} \exists Q^{path}, \delta \rightarrow P_i \in I(G) | P_i.\ size = minsize \wedge (P_i \rightarrow P_{i+1}) \in I(G)$ if $\exists var \in Q_i^p == var \in Q_{i+1}^p$.

10. Data Insertion – Given Q^R and Q^p then check $\exists P_i \in Q^p \in I(G)$, if true then check $P_i \in H(G) \vee createh_i \in P_i \wedge updateH(G)$ with h_i. Check if $\exists var \in Q^p \in H(G)$, if true then overlap h_i with var's $h \vee updateh_i$ with $var \in Q^p$.

11. Data Deletion – Given Q^R and Q^p then check $\exists P_i \in Q^p \in I(G) \wedge P_i \in var$, if true then check $|h_i| == 0$, if true then remove $var \in Q^p \in H(G) \vee$ copy of P_i exists.

Given these definitions, their paper proposes algorithms to perform the whole of the queries for this database[SGDD15]:

Algorithm 23 Create Hypergraph

1: Given an RDF graph G as triple (S, O, P).
2: $V \leftarrow \emptyset, E \leftarrow \emptyset, e_i \leftarrow \emptyset$
3: $\forall (S, O, P) \in G, V \leftarrow V \cup V \in (S \cup O \cup P)$.
4: $\forall P_i | 1 \leq i \leq n, 1 \leq j \leq n, e_i \leftarrow \{P_i, \{S_j, O_j\}\}, E \leftarrow e_i$.
5: $H(G) = (V, E)$.

Algorithm 24 Create Predicate-Based Index

1: Given a hypergraph $H(G)$.
2: Sort hyperedges by size.
3: $\forall h_i | 1 \leq i \leq n-1$, if $MIN(size(h_i))$ then $I(G) = I(G) \cup \delta \downarrow P_i, \forall j | i+1 \leq j \leq n$, if $(h_i(G) \sqsubseteq h_j(G)) \wedge (size(P_i) == size(P_j))$ then $I(G) = I(G) \cup P_i \leftrightarrow P_j$ else $I(G) = I(G) \cup P_i \rightarrow P_j$.

Given this construction, we now split between the roles of query planner and evaluator.

4.4 Query Planner

For query planning, the sender role is conducted by the key holder for the relevant resources. The receiver role is conducted by the cluster(s) responsible for the hypergraph. Again, these algorithms are produced from the source literature[SGDD15]:

Algorithm 25 Create SPARQL Query Graph

1: Given a query match pattern $Q^p \in Q^R$.
2: $V \leftarrow \emptyset, E \leftarrow \emptyset$
3: $\forall Q_i^p | 1 \leq i$
- if $i == 1, V \leftarrow V \cup (var \in Q_i^p.S \cup var \in Q_i^p.O), E \leftarrow (var \in Q_i^p.S, var \in Q_i^p.O)$
- if $i \geq 2$, if $(var \in Q_i^p.S == var \in Q_{i+1}^p.S) \wedge (var \in Q_i^p.O == var \in Q_{i+1}^p.S)$ then $V \leftarrow V \cup var \in Q_i^p.S, E \leftarrow (var \in Q_i^p, var \in Q_{i+1}^p)$ else $V \leftarrow V \cup var \in Q_{i+1}^p.O, E \leftarrow (var \in Q_i^p, var \in Q_{i+1}^p)$
- else, if $(var \in Q_i^p.S == var \in Q_{i+1}^p.O) \wedge (var \in Q_i^p.O == var \in Q_{i+1}^p.O)$ then if $\exists var \in Q_{i+1}^p.S \in V, V \leftarrow V \cup var \in Q_i^p.S, E \leftarrow (var \in Q_i^p, var \in Q_{i+1}^p)$. $MIN(size(h_i))$ then $I(G) = I(G) \cup \delta \downarrow P_i, \forall j | i+1 \leq j \leq n$, if $(h_i(G) \sqsubseteq h_j(G)) \wedge (size(P_i) == size(P_j))$ then $I(G) = I(G) \cup P_i \leftrightarrow P_j$ else $I(G) = I(G) \cup P_i \rightarrow P_j$.
- If not $(var \in Q_i^p.S == var \in Q_{i+1}^p.O) \wedge (var \in Q_i^p.O == var \in Q_{i+1}^p.O)$, then $V \leftarrow V \cup var \in Q_{i+1}^p.S, E \leftarrow (var \in Q_i^p, var \in Q_{i+1}^p)$.

4: $Q^G = (V, E)$

Algorithm 26 Create Query Path

1: Given a set of predicates $P \in Q^G, Q^G$ and $I(G)$.
2: Start from δ.
3: $Q^{path} \leftarrow \delta \rightarrow min(size(P_i)) \in I(G)$
4: $\forall Q_i | 1 \leq i \leq n-1, i+1 \leq j \leq n$, if $\exists var \in Q_i^p \in Q^G == var \in Q_j^p \in Q^G$, then $Q^p ath \leftarrow Q^p ath \cup (P_i \rightarrow P_j) \in I(G)$.

4.5 Query Evaluator

Execution of queries retains the same role responsibility, where the receiver is the cluster(s) responsible for the hypergraph, and the sender is the keyholder. The sender being blind to decisions of the receiver, makes the query being processed indistinguishable from gossip requests for additional data blocks, which will happen by virtue of the parent BlossomSub protocol. Again, the algorithms are produced from the source literature[SGDD15]:

Algorithm 27 Load Node $(Q_i^p(S, O))$

1: $V \leftarrow \emptyset, E \leftarrow \emptyset$
2: Execute query on h_i.
3: $v_i \leftarrow S \wedge w_j \leftarrow O, V \leftarrow \{v_i, w_j\}, E \leftarrow (v_i, w_j), Q_i^{AG} \leftarrow (V, E)$
4: Return Q_i^{AG}.

Algorithm 28 Load Neighbor Node $(Q_i^p, Q_{i+1}^p(S, O))$

1: $V \leftarrow \emptyset, E \leftarrow \emptyset$
2: If $(Q_i^p.S == Q_{i+1}^p.S) \wedge (Q_i^p.O == Q_{i+1}^p.S)$, then if $\exists Q_{i+1}^p.O \in V$ then $V \leftarrow V \cup Q_i^p.S \wedge Q_i^p.O \in Q_{i+1}^p.O, E \leftarrow (V \in Q_i^p, V \in Q_{i+1}^p)$ else $V \leftarrow V \cup Q_{i+1}^p.O, E \leftarrow (V \in Q_i^p, V \in Q_{i+1}^p)$.
3: Else, if $(Q_i^p.S == Q_{i+1}^p.O) \wedge (Q_i^p.O == Q_{i+1}^p.O)$ then if $\exists Q_{i+1}^p.S \in V$ then $V \leftarrow V \cup Q_i^p.S \wedge Q_i^p.O \in Q_{i+1}^p.S, E \leftarrow (V \in Q_i^p, V \in Q_{i+1}^p)$, else $V \leftarrow V \cup Q_{i+1}^p.S, E \leftarrow (V \in Q_i^p, V \in Q_{i+1}^p)$. In either case, $Q^{AG} \leftarrow Q_i^{AG} \cup (V, E)$
4: Return Q^{AG}.

Algorithm 29 Process Query

1: Given Q^G, Q^p and $I(G)$.
2: $\forall Q_i^p | 1 \leq i \leq n-1$, if $i == 1$ then execute $LoadNode$ with $Q_i^p(S, O)$, else execute $LoadNeighborNode$ with $Q_i^p, Q_{i+1}^p(S, O)$.
3: Return Q^{AG}.

Finally, the authors produced a cost function for graph creation[SGDD15]:

$$cost(H(G)) = \sum_{i=1}^{n} cost(V_i.E_i)$$

And query processing[SGDD15]:

$$cost(Q^{proc}) = \sum_{P_i \in Q^{path}} cost(P_i)$$

We adopt these cost metrics in terms of OTs required for evaluating predicates P_i, and storing of vertices per V_i and edges per E_i.

4.6 Turing Completeness

While a hypergraph of course can represent a Turing machine's tape and perform queries which evaluate state transitions of the Turing machine, and thus it can be easily stated that as-is this design is Turing-complete, it would be painfully slow, as building the index is in of itself bounded at a complexity of $O(n^2)$ where n is the number of hyperedges for a given predicate and processing is $O(n)$ where n is the number of match patterns. Instead, we can trade off to utilizing our OT circuit building techniques to loading circuits stored as resources on the global hypergraph. Because loading and storing onto hypergraphs requires a controlling key, and that key is known to be needed at the time of the query, we have an ideal resource control mechanism in the form of asking for relevant control keys before the query is executed, which can flag potential abuse in cross-resource querying.

5 Operating System

We now define the operating system layer of Quilibrium, in which we build a database operating system on the hypergraph database, realizing common OS primitives, such as a file system, scheduler, IPC-like equivalent, message queuing, control key management, then finally a universal resource pattern, in which we globally define resources, and unique instances of those resources, initially with an Account resource, which tracks the balance of individual holders' network reward tokens, unifying the reward structure at the most base level of the protocol all the way up the architectural stack.

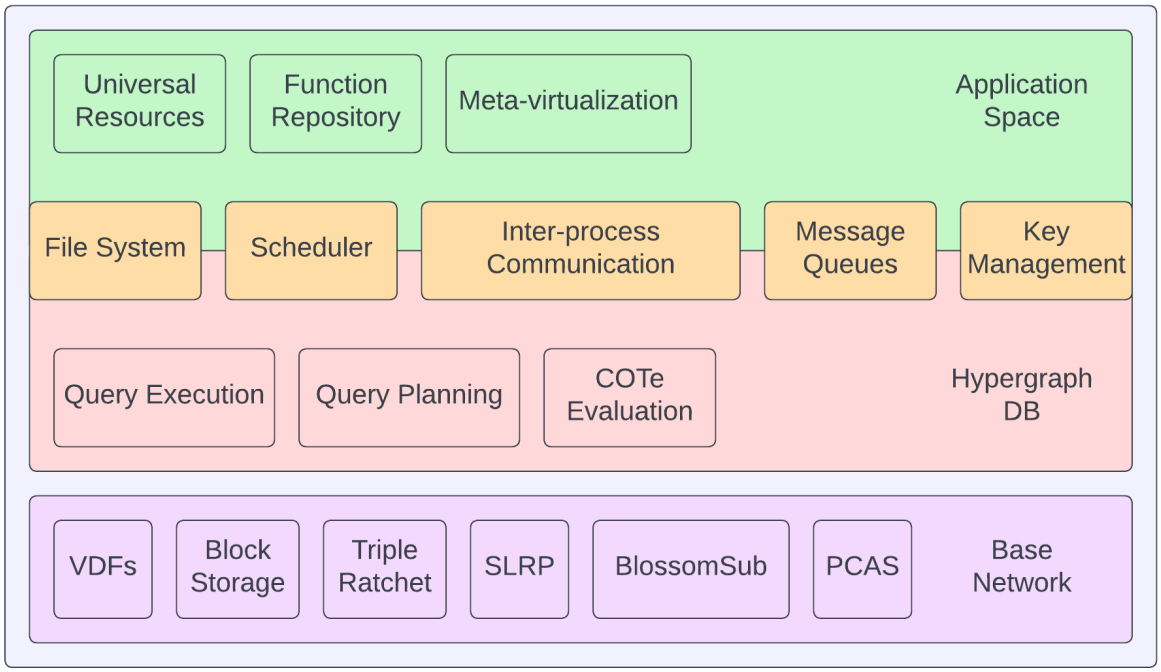


Figure 8: The complete view of Quilibrium's layers

5.1 Database Operating System

The idea of database-oriented operating systems is a newer idea, first formalized in "DBOS: A DBMS-oriented Operating System", where the authors introduce a microkernel-style base layer upon which a distributed

database has raw device access, and then OS-level primitives are realized on top of the database [SLK+22]. In the same way, we will realize OS-level primitives to make decentralized application development simpler, however, it will not involve a base level microkernel necessarily (although this is not to preclude someone from building a rumpkernel host for Quilibrium nodes), nor will these primitives be represented as sharded tables. Instead, we will rely on the hypergraph representation of RDF, and construct these resources through this. For simplicity in representation, we will use RDF syntax to define these things. Through named IRI references to the address of any graph, we can link these concepts together where relevant.

5.2 File System

We begin with a basic representation of loose files – while typical file systems are hierarchical, we adopt an object store more proximate to S3.

```
:File a rdfs:Class;
    rdfs:label "a file object".
:FileSize a rdfs:Property
    rdfs:domain rdfs:Literal;
    rdfs:range :File.
:FileName a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :File.
:FileOctet a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :File.
:Block a rdfs:Class;
    rdfs:label "a block of data";
:FileParent a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :File.
:BlockHash a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :Block.
:BlockData a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :Block.
:BlockNumber a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :BlockData.
:BlockNumber a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :BlockData.
:BlockSize a rdfs:Property
    rdfs:domain rdfs:Literal;
    rdfs:range :Block.
```

This representation can of course be extended to provide additional features, but is sufficient for describing a basic block list associated with a file. With FUSE drivers per RDF2FS, we immediately gain a direct bridge between classical OS and Quilibrium, offering file backup capabilities.

5.3 Scheduler

We define a scheduler based on two possible types, a priority-based scheduler (where tasks can be easily ordered as a max-heap), and a cron-based scheduler (where repeated tasks will get picked up at increments aligned to the schedule).

```
:Task a rdfs:Class;
    rdfs:label "a simple task".
:TaskData a rdfs:Property;
    rdfs:domain :File;
    rdfs:range :Task.
:TaskPriority a rdfs:Property;
    rdfs:label "a one-time execution parameter that indicates the 0-255 priority, in order of pr
    rdfs:domain rdfs:Literal;
    rdfs:range :Task.
:TaskSchedule a rdfs:Property;
    rdfs:label "A cron string that describes the frequency to evaluate the task";
    rdfs:domain rdfs:Literal;
    rdfs:range :Task.
:TaskResult a rdfs:Property;
    rdfs:domain rdfs:File;
    rdfs:range :Task.
```

5.4 Inter-Process Communication

Defining IPC in RDF is easily achieved, using a named graph based structure:

```
:IPCMessage a rdfs:Class;
    rdfs:label "a message".
:MessageData a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :IPCMessage.
:ToAddress a rdfs:Property;
    rdfs:label "the address of the target graph";
    rdfs:domain rdfs:Literal;
    rdfs:range :IPCMessage.
```

5.5 Message Queue

Construction of a basic message queue can be achieved by a linked list, with a parent reference.

```
:Queue a rdfs:Class;
    rdfs:label "a FIFO queue".
:QueueNode a rdfs:Class;
    rdfs:label "a node in a queue".
:HeadNode a rdfs:Property;
    rdfs:domain :QueueNode;
    rdfs:range :Queue.
:NextNode a rdfs:Property;
    rdfs:domain :QueueNode;
    rdfs:range :QueueNode.
:QueueMessage a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :QueueNode.
```

5.6 Key Management

Key management is a necessary component of the protocol, as to allow any member of a cluster to participate on their relevant side of an OT circuit. This is additionally important in the context of non-interactive processing – where a client is not directly initiating the computation, but rather the protocol has prompted it, by virtue of task management or other functions.

```
:Key a rdfs:Class;
    rdfs:label "a key object".
:KeyShare a rdfs:Class;
    rdfs:label "a share corresponding to a key".
:OfKey a rdfs:Property;
    rdfs:domain :Key;
    rdfs:range :KeyShare.
:Format a rdfs:Property;
    rdfs:domain :literal;
    rdfs:range :Key.
:PublicData a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :Key.
:Protocol a rdfs:Property;
    rdfs:domain :literal;
    rdfs:range :Key.
:KeyData a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :KeyShare.
```

Because the distinct sections of data are effectively controlled and encrypted by the relevant keyholders, provided keyshare owners are not one and the same, the key never will exist combined on a single device, but further, even if keyshare owners are one and the same, their meaningful online use would still be reflected in the global hypergraph mutation and thus cannot be used to forge state.

The Protocol reference property is multi-purpose – because it is a Literal, it may refer to a known protocol that is baked into the node software, or, if all parties are inclined to engage, can refer to an executable File reference which can contain an OT circuit, so as to enable additional MPC protocols not inherent to network function.

5.7 Accounts

Accounts are the aspect that allows one to assert both identity and balances. The network reward token will be a Coin, which in a simple form would be merely an exchangeable, splittable unit of balance, but in the historic context of cryptocurrencies, there have been problems discovered in both public block chains and private block chains. The Quilibrium network is a private computation network, and by virtue there exists an ethical dilemma: rigid financial institutions cannot accept a coin without explicit proof of legitimacy, but people deserve a right to financial privacy. To counteract this problem, we adopt a bloom filter property, which on the transfer of a coin, the circuit will apply the holding account's public address. Because a user may wish to check this coin against a public registry of known bad actors, they may reference a public list provided by a financial institution, wherein they can choose to accept this coin, or reject it, which will result in its completed transfer to the designated refund address. Coins may be joined together, with the caveat that the bloom filter will also be unioned, but the choice of joining is at the behest of the owner. Similarly, Coins may be split, but will inherit the bloom filter.

```
:Account a rdfs:Class;
    rdfs:label "an account object".
:Coin a rdfs:Class;
    rdfs:label "an object containing a numeric balance and historical lineage".
:CoinBalance a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :Coin.
:OwnerAccount a rdfs:Property;
    rdfs:domain rdfs:Account;
    rdfs:range :Coin.
```

```
:LineageFilter a rdfs:Property;
    rdfs:domain rdfs:Literal;
    rdfs:range :Coin.
:PendingTransaction a rdfs:Class;
    rdfs:label "a pending transaction".
:ToAccount a rdfs:Property;
    rdfs:domain rdfs:Account;
    rdfs:range :PendingTransaction.
:RefundAccount a rdfs:Property;
    rdfs:domain rdfs:Account;
    rdfs:range :PendingTransaction.
:OfCoin a rdfs:Property;
    rdfs:domain :Coin;
    rdfs:range :PendingTransaction.
```

Notably, the presence of a pending transaction does not reveal a source account, retaining sender privacy outside of a bloom filter entry, and so if the sender wishes to retain that privacy they may designate an alternative account for refund, then consolidate funds afterwards.

5.8 Universal Resources

The definitions of these RDF schema will be deployed to the network on initialization by the Quilibrium team, with an access key so that any member of the network may read them and reference them. This model of schema deployment is intended such that anyone may deploy a new data schema and instantiate a universal resource. We further intend to migrate ownership of these schema to a separate foundation, which can be governed independently such that the improvements remain in the pure interest of the network's continued function and evolution.

References

[ALSZ15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. 2015.

[BBBF19] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. 2019.

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. 1981.

[CO15] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. 2015.

[Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. 1987.

[Hås05] Johan Håstad. The square lattice shuffle. 2005.

[HKL+12] Stefan Heyse, Eike Kiltz, Vadim Lyubashesvky, Christof Paar, and Krzysztof Pietrzak. Lapin: An efficient authentication protocol based on ring-lpn. 2012.

[Hud16] Péter Hudoba. Public key cryptography based on the clique and learning parity with noise problems for post-quantum cryptography. In *Proceedings of the 11th Joint Conference on Mathematics and Computer Science*, 2016.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. 2003.

[JP00] A Juels and M Peinado. Hiding cliques for cryptographic security. In *Designs, Codes and Cryptography*, pages 269–280. Springer, 2000.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure ot extension with optimal overhead. 2015.

[Kuč91] Lukáš Kučera. A generalized encryption scheme based on random graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 180–186, 1991.

[KZG10] Aniket Kate, Gregory Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. 2010.

[LK22] Donghang Lu and Aniket Kate. Rpm: Robust anonymity at scale. 2022.

[Lon20] Lipa Long. Binary quadratic forms. 2020.

[Mar16] Moxie Marlinspike. The double ratchet algorithm. 2016.

[nus21] nusenu. Is "kax17" performing de-anonymization attacks against tor users? 2021.

[SGDD15] Sangeeta Sen, Moumita Ghosh, Animesh Dutta, and Biswanath Dutta. Hypergraph based query optimization. 2015.

[Sha79] Adi Shamir. How to share a secret. 1979.

[SLK+22] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. Dbos: A dbms-oriented operating system. 2022.

[VNM+20] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias, and Yiannis Psaras. Gossipsub: Attack-resilient message propagation in the filecoin and eth2.0 networks. 2020.

[Wes18] Benjamin Wesolowski. Efficient verifiable delay functions. 2018.

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated ot with small communication. 2020.