Atomic and Fair Data Exchange via Blockchain

최원재

2025. 01. 04

Virtual Machine & Optimization Laboratory

Dept. of Electrical and Computer Engineering

Seoul National University

Paper Info

Title

- Atomic and Fair Data Exchange via Blockchain
- Link

Conference

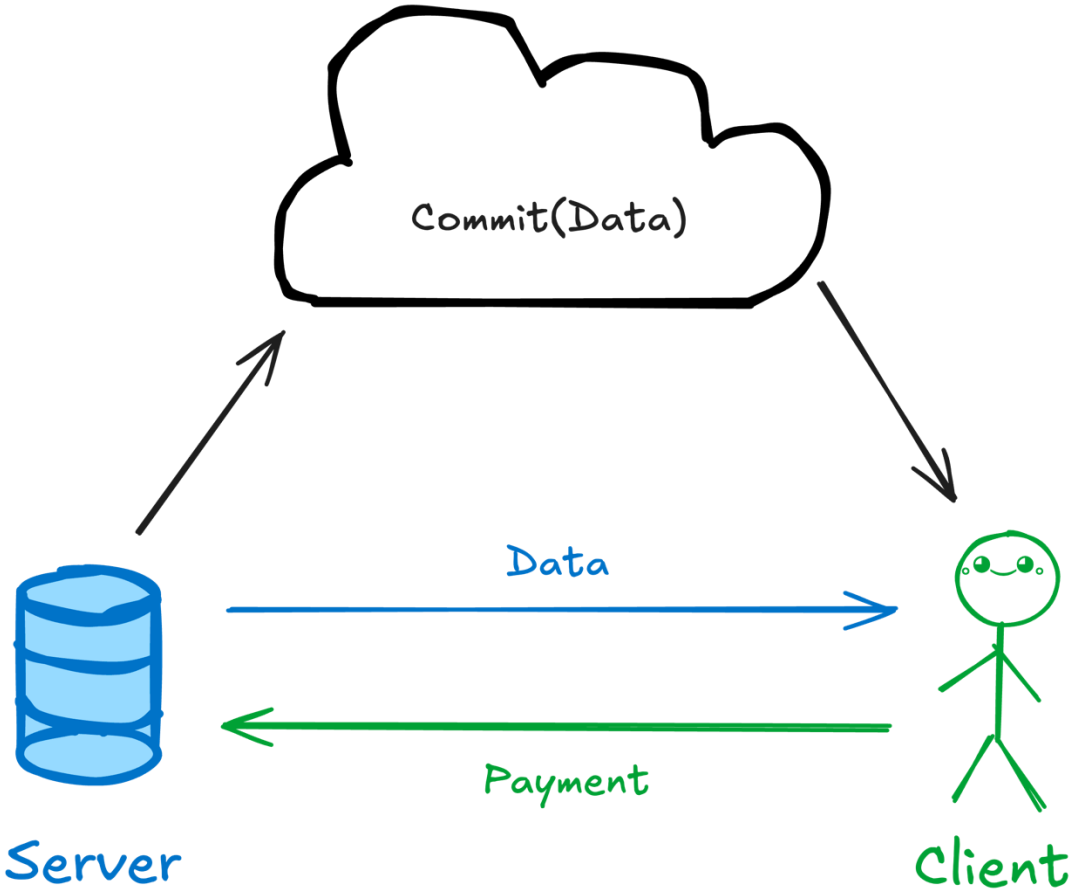
- CCS '24: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security

Authors

- Ertem Nusret Tas
  - Ph.D Candidate at Stanford
  - Also author of Bitcoin-enhanced proof-of-stake security: Possibilities and impossibilities
- A16z

Problem Statement

The Fair Data Exchange (FDE) Problem



Server Fairness

- An adversarial client cannot learn anything about the data without paying the server.

Client Fairness

- An adversarial server cannot receive any payment if the client does not obtain the data.

Correctness

- If the client and the server are honest, the client obtains the data, and the server obtains the payment.

Existing Solutions

Subscription-based

- Client pays the server in advance
- Trust the reputation of the server to deliver the data.
- No Client Fairness

Free Data

- Server provides data for free
- Lack of Incentive for the server.
- No data integrity
- No Server Fairness

Blockchain

- Too Expensive (1MB = $3000)

Proof of Storage / Retreivability

- Only guarantees and incentivize the storage of data
- No guarantee and incentive for the real delivery of the data

FDE without Trusted Third Party is Impossible

- ‘On the Impossibility of Fair Exchange without a Trusted Third Party’, Pagnia and Gartner, 1999

Blockchain as a Trusted Third Party!

Usecases

Private Data Exchange

- Private Key

Data Marketplace

- Movies
- AI Models

Proto-Danksharding (EIP-4844)

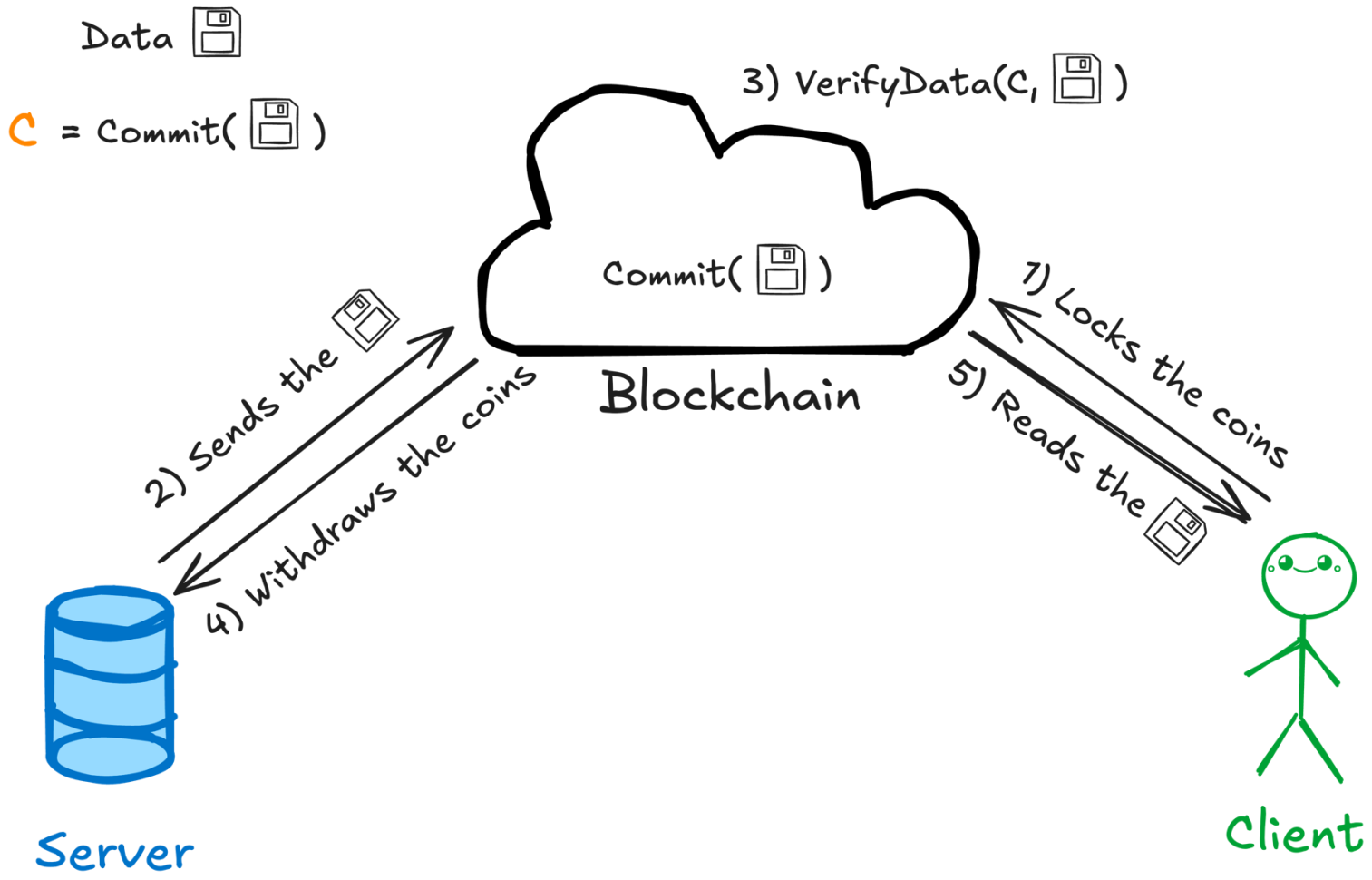
- Blob data is not stored on the Ethereum.
- Only the KZG Commitment is stored.
- Servers(Nodes) voluntarily store the real blob data and provides the the clients.
- FDE can incentivize it

Blockchain Archive Data

- Archive data is also not incentivized but needs big amount of data storage

FDE Protocol

Naïve Solution



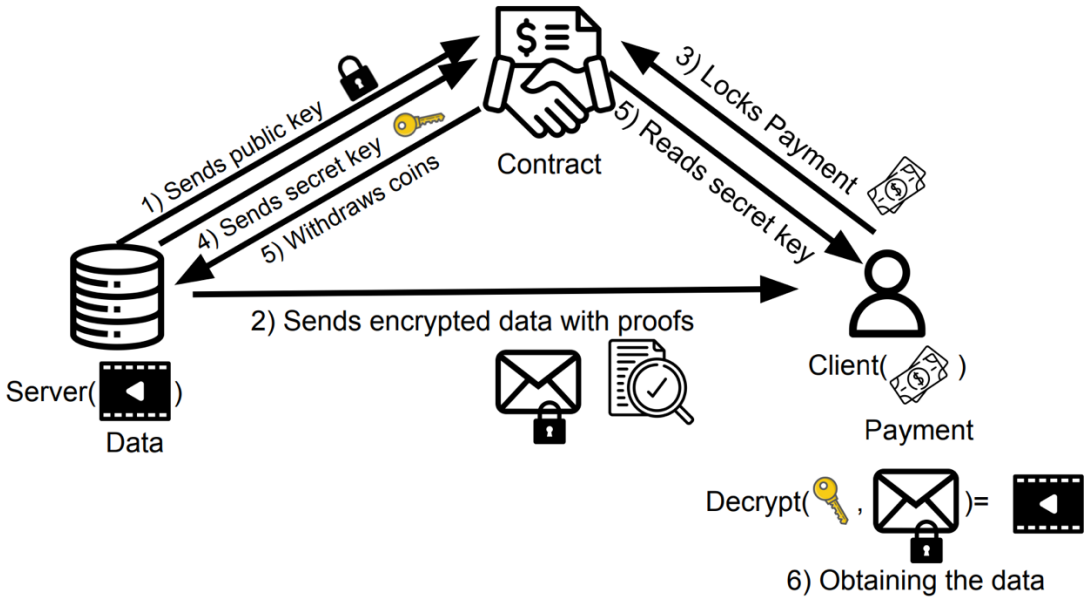
Problems

All data is posted to the blockchain

Two Problems

- Not only the buyer gets the data but everyone can get the data.
- Uploading all the data to the blockchain is impossible(or expensive) due to the limited capacity.

FDE Protocol



Solution

- Server encrypts the data
- Transfers the encrypted data off-chain
- Sells the decryption key on-chain

Must Prove

- The key decrypts the ciphertext to the data under C

Verifiable Encryption under Committed Key (VECK)

Common

- $Gen(crs) \rightarrow pp$

Server

- $Enc(C, w) \rightarrow (vk, sk, ct, \pi)$
  - Proof π proves ct is the encryption of the data committed by C under sk committed by vk.

Blockchain

- $Verify_{key}(vk, sk) \rightarrow 1\ or\ 0$

Client

- $Verify_{ct}(C, vk, ct, \pi) \rightarrow 1\ or\ 0$
- $Decrypt(sk, ct) \rightarrow w\ or\ \perp$

Properties of VECK

Correctness

- Verifications for honestly generated encryption succeed.
- Correctness of FDE

Soundness

- No PPT adversary can generate vk, sk, ct, π such that verification succeeds, yet decryption does not output w.
- Client Fairness of FDE

Computational Zero-Knowledge

- The ciphertext and the proof leak no additional information about the witness.
- Server Fairness of FDE

KZG Commitment

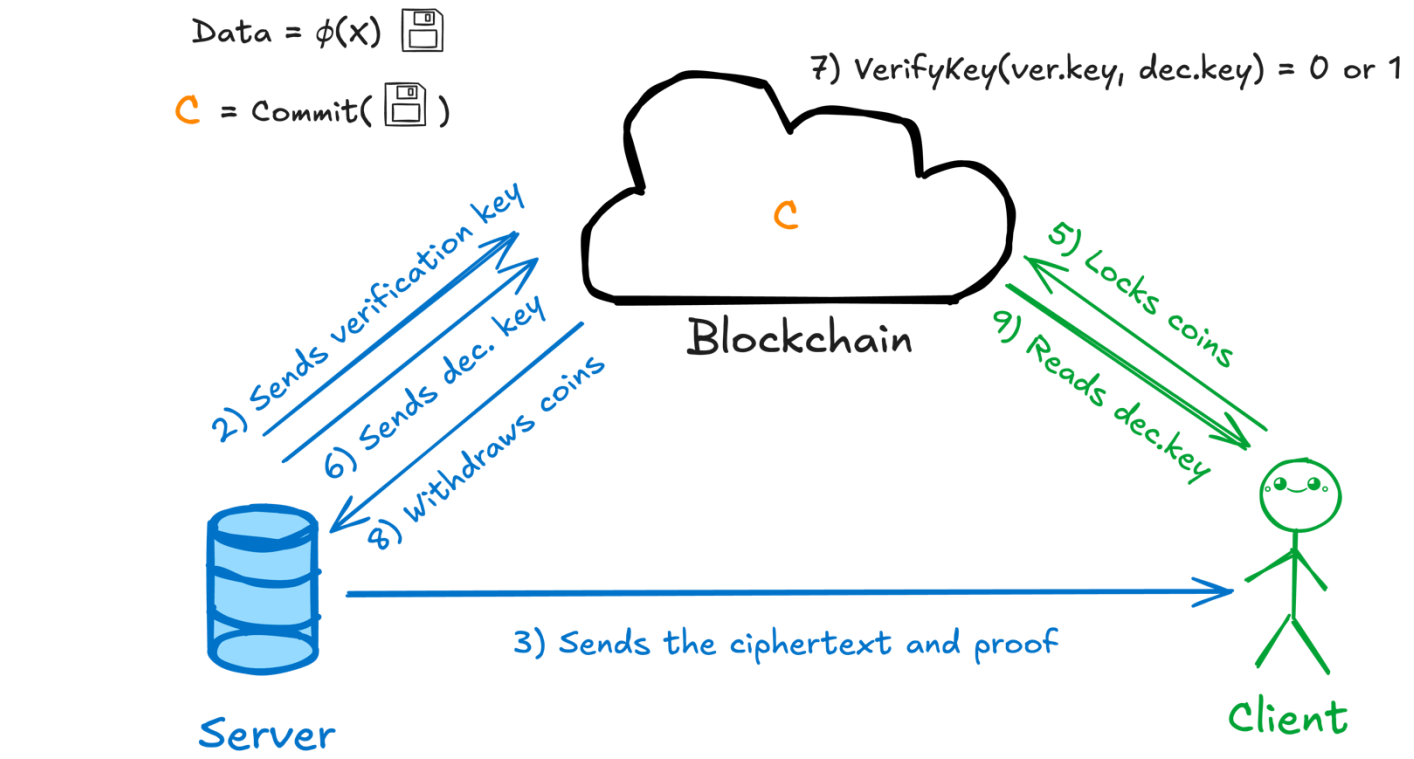
Proto-Danksharding (EIP-4844)

- For the utilization for proto-danksharding, the paper assumes the KZG Commitment scheme.

KZG Commitment

- $w = \phi$
- $data = (\phi(0), \dots, \phi(l))$
- $C = CommitKZG(\phi)$

FDE Protocol for KZG Commitments



1) Enc(C ,) = (ver.key, dec.key, ciphertext, proof)

4) VerifyCt(C, ver.key, ciphertext, proof) = 0 or 1

10) Obtains the data

Decrypt(dec.key, ciphertext) =

Two Encryption Algorithms

ElGamal Encryption

Paillier Encryption

ElGamal Encryption

Key Generation

- p: Big Prime Number
- g: Primitive Root
- x: Random Sampled Secret Key
- y: Public Key
  $y = g^x \bmod p$

Encryption

- Ciphertext $(c1, c2)$
  - $c1 = g^k \bmod p$
  - $c2 = m \cdot y^k \bmod p$

Decryption

- $s = c_1^x \bmod p$
- $m = c_2 \cdot s^{-1} \bmod p$

- Asymmetric Key Encryption Algorithm
- Based on the Diffie-Hellman Key Exchange
- Security depends on Decisional Diffie Hellman Problem
- Discrete Logarithm Problem

ElGamal-based VECK (SNARK version)

Prover (Server)

$Enc(C, \phi(X)) \rightarrow (sk, vk, ct, \pi)$

- Sample $s \leftarrow {}_R\mathbb{F}_P$
- Set $sk := s$ (ElGamal decryption key)
- $vk := h^s$ (corresponding verification key)
- $ct := \{h_i{}^s g_i{}^{\phi(i)}\}_{i=0}^{l}$ (ElGamal ciphertexts)
- Generate $\pi := SNARK.Prove(crs,\ instance = (C, vk, ct),\ witness = (s, \phi(X))$

Verifier (Client)

$Verify_{ct}(C, vk, ct, \pi) \rightarrow 0\ or\ 1$

- Output 1 if π verifies against C, vk, ct

$Decrypt(sk, ct) \rightarrow w\ or\ \perp$

- Decryption algorithm for ElGamal

Blockchain Contract

$Verify_{key}(vk, sk) \rightarrow 0\ or\ 1$

- Check if $h^{sk} == vk$

zkSNARK is Inefficient!

Efficient VECK protocol

- In this paper, the protocol utilizes the shared structure of the ElGamal & KZG

Splitting the Message

- Can support large message by splitting the message into k chunks with range proofs.

Protocol Breakdown

VECK Protocol with ElGamal Encryption and KZG Commitment

$\textbf{Gen}(\text{crs}) \to \text{pp}$: On input $\text{crs} = (\mathcal{G}, \{g_1^{\tau^i}\}_{i=1}^n, \{g_2^{\tau^i}\}_{i=1}^n)$, generate random group elements with unknown discrete logarithms $h_i \leftarrow_R \mathbb{G}_1, h \leftarrow_R \mathbb{G}_1$ for $i \in [n]$. In practice, these can be generated using a Hash-to-Curve [9] function $H_{\text{curv}}: \mathbb{F}_p \to \mathbb{G}_1$, $h_i := H_{\text{curv}}(i)$. Output $\text{pp} = (\text{crs}, \{h_i\}_{i=0}^n, h)$.

$\textbf{Enc}(F_{[\ell]}^{\textbf{full-eval}}, C_\phi, \phi(X)) \to (\text{vk}, \text{sk}, \text{ct}, \pi)$:

(1) Sample $s \leftarrow_R \mathbb{F}_p$, set $\text{sk} := s$, $\text{vk} = h^s$.

(2) Compute the encryptions $\{\text{ct}_i := h_i^s \cdot g_1^{\phi(i)}\}_{i=0}^\ell$.

(3) Generate a Fiat-Shamir pseudo-random challenge $\alpha := \text{H}(C_\phi, \text{ct}_0, \ldots, \text{ct}_\ell, [\ell])$.[a]

(4) Compute $C_\alpha := g_1^{\phi(\alpha)+s(\tau-\alpha)}$.

(5) Compute the opening proof for α: $\pi_\alpha := \textsc{Open}(\text{crs}, \alpha, \phi(X) - s(X-\alpha))$ (note that $\pi_\alpha = g_1^{(\phi(\tau)-\phi(\alpha))/(\tau-\alpha)-s}$).

(6) Compute π_{LIN} to prove that C_α is indeed of the form g_1 raised to a known power, multiplied by $g_1^{\tau-\alpha}$ raised to the secret s of the verification key, *i.e.*, we compute the proof for the relation $\mathcal{R}_{\text{LIN}}$ as described in Appendix A.2, Equation (A.4):

$$\mathcal{R}_{\text{LIN}} = \left\{ \begin{array}{llll} g_{1,1} = g_1 & g_{1,2} = g_1^{\tau-\alpha} & u_1 = C_\alpha & x_1 = \phi(\alpha) \\ g_{2,1} = 1 & g_{2,2} = h & u_2 = \text{vk} \;; & x_2 = s \end{array} \;\middle|\; g_{1,1}^{x_1} g_{1,2}^{x_2} = u_1 \wedge g_{2,1}^{x_1} g_{2,2}^{x_2} = u_2 \right\}.$$

(7) Compute $Q := \prod_{i=0}^\ell h_i^{L_{i,\ell}(\alpha)} \cdot g_1^{-(\tau-\alpha)}$.

(8) Compute discrete logarithm equality π_{DLeq} with respect to (Q, Q^s, h, vk) for the witness s, *i.e.*, we compute the proof for the relation $\mathcal{R}_{\text{DLeq}}$ as described in Appendix A.2, Equation (A.3):

$$\mathcal{R}_{\text{DLeq}} := \{((g = Q, h, a = Q^s, b = \text{vk});\; x = s) \mid a = g^x \wedge b = h^x\}.$$

(9) Output $(\text{sk}, \text{vk}, \text{ct} = [\text{ct}_0, \ldots, \text{ct}_\ell], \pi = (C_\alpha, \pi_\alpha, \pi_{\text{LIN}}, \pi_{\text{DLeq}}))$.

$\textbf{Ver}_{\text{ct}}(F_{[\ell]}^{\textbf{full-eval}}, C_\phi, \text{vk}, \text{ct}, \pi) \to 0/1$:

(1) Parse $\text{ct} = (\text{ct}_0, \ldots, \text{ct}_\ell)$ and the server's proofs $\pi = (C_\alpha, \pi_\alpha, \pi_{\text{LIN}}, \pi_{\text{DLeq}})$.

(2) Verify π_{LIN} for C_α against $g_1, g_1^{\tau-\alpha}, h, C_\alpha, \text{vk}$.

(3) Compute the Fiat-Shamir challenge: $\alpha := \text{H}(C_\phi, \text{ct}_0, \ldots, \text{ct}_\ell)$.

(4) Verify the opening proof π_α: $e(C_\phi / C_\alpha, g_2) = e(\pi_\alpha, g_2^{\tau-\alpha})$.

(5) Compute Q as above for encryption.

(6) Compute $\text{ct} := \prod_{i=0}^\ell \text{ct}_i^{L_{i,\ell}(\alpha)} \in \mathbb{G}_1$, and $Q^* := \text{ct}/C_\alpha$.

(7) Verify π_{DLeq} for the relation (Q, Q^*, h, vk).

(8) If any of the verification checks fail, output 0. Otherwise, output 1.

$\textbf{Ver}_{\text{key}}(\text{vk}, \text{sk}) \to 0/1$: For $\text{sk} = s \in \mathbb{F}_p$, return 1 if and only if $\text{vk} = h^s$.

$\textbf{Dec}(F_{[\ell]}^{\textbf{full-eval}}, \text{sk}, \text{ct}) \to \{\phi(i)\}_{i=0}^\ell$: For $\text{sk} = s \in \mathbb{F}_p$, for each $i \in [\ell]$, recover the value $x = \phi(i)$ by brute-forcing $x \in [\mathcal{B}]$ in $g_1^x = \text{ct}_i / h_i^s$. Pollard's rho algorithm for logarithms [59], Shank's baby-step-giant-step algorithm [65] or brute-force with pre-computed tables [22] could be helpful to find the right trade-off between decryption memory and time.

[a] Generalizing to more general subsets S other than $[\ell]$, the description of the subset S would be part of the input of $\text{H}(\cdot)$ generating the Fiat-Shamir challenge.

VECK Protocol with ElGamal Encryption and KZG Commitment

1. Setup

- Establish Reference
  - Generates a common reference string (CRS) and public parameters.
  - Aligns all participants on the same group-based environment for commitments, encryption and proof verification.

2. Encryption

- Key Generation
  - Server picks a random exponent (secret key) and derives a corresponding verification key.
- Per-Point Ciphertexts
  - Evaluates the polynomial at each point and encrypts these values via group exponentiation.
- Proofs of Correctness
  - Produces short zero-knowledge proofs asserting that the ciphertexts match the committed polynomial under the same secret exponent.

High Level Explanation of the Proof

1. KZG Commitment

- The server has a **committed polynomial** $\phi(X)$ via a KZG commitment, $C\phi$.
- This commitment fixes the polynomial but does not reveal its coefficients.

2. Ciphertexts and a Secret Exponent

- For each point i, the server encrypts the value $\phi(i)$ using a secret exponent ss.
- Each ciphertext ct_i encodes both $\phi(i)$ and the exponent s.

3. Random Challenge α

- A **Fiat–Shamir**-style hash function takes the public data (commitment, ciphertexts) and yields a random challe nge α.
- This binds the proof to the specific commitment and ciphertext set, preventing replay or substitution attacks.

4. Blinded Evaluation at α\alpha

- The server “opens” (proves knowledge of) the polynomial at α\alpha, **but** adds a twist: it mixes in the same exp onent s.
- Concretely, the server shows a short proof that
  - $\phi(\alpha)$ is consistent with the KZG commitment,
  - ss is used in a linear combination with $\phi(\alpha)$, ensuring the exponent matches that used in the ciphertexts.

High Level Explanation of the Proof (Continued)

5. Aggregation via Lagrange Polynomials

- The ciphertexts $\{ct_i\}$ are cleverly combined (using Lagrange coefficients) to simulate an "encryption" of $\phi(\alpha)$.
- If that aggregated encryption matches the server's **blinded** version of $\phi(\alpha)$ from the previous step, it confirms each ct_i was formed correctly.

6. Discrete-Log Equality (DLeq) Check

- Finally, there is a **discrete log equality proof** to confirm the server used **the same secret exponent** s across:
  - The KZG-related portion, and
  - The encryption process.
- If this DLeq check holds, it proves the exponent in the ciphertexts truly matches the exponent that "opens" the polynomial.

By combining these steps, the verifier sees that:

- The polynomial $\phi(X)$ in the KZG commitment is the **exact** polynomial the server used for all evaluations.
- Each ciphertext ct_i **truly** encrypts $\phi(i)$.
- A **single** secret exponent ss ties everything together, so there is no mismatch or cheating between commitmen t and ciphertexts.

VECK Protocol with ElGamal Encryption and KZG Commitment

3. Verification of Ciphertexts

- Challenge Binding
  - The verifier recomputes or checks a challenge (often via a hash) that links the polynomial commitment, the ciphertexts, and t he server's claimed verification key.
- Check Consistency
  - Verifies that each ciphertext matches the supposed polynomial evaluation and that all proofs pass. This involves checking if t he same secret key was consistently used and if the commitments align with the encrypted values.

4. Verification of Key

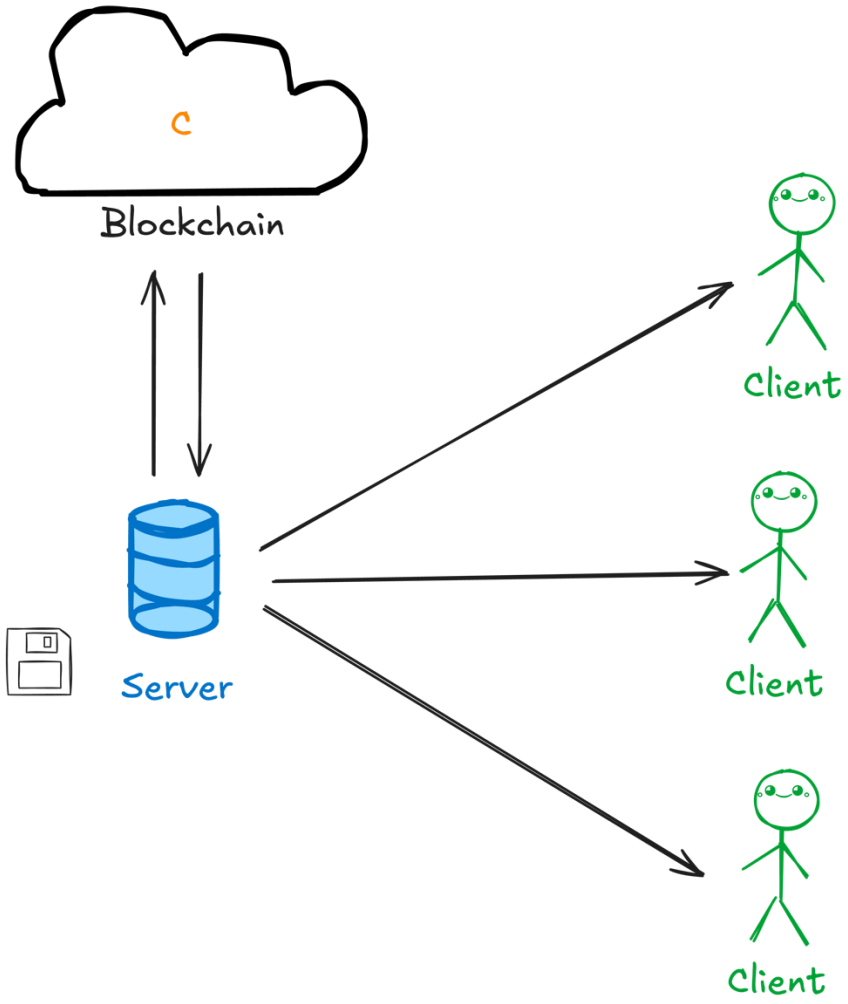
- Server Reveals Secret Exponent
  - At some stage (often to claim payment), the server discloses its secret exponent to a smart contract or directly to the client.
- Key-Value Matching
  - A simple group-based check confirms that the revealed exponent truly corresponds to the previously published verification key. This step prevents a dishonest server from providing a key that does not actually decrypt the ciphertexts.

VECK Protocol with ElGamal Encryption and KZG Commitment

5. Decryption

- Recover Polynomial Evaluations
  - The client (or any holder of the secret exponent) undoes the exponentiation in the ciphertexts to retrieve the original polynomial values.
- Discrete Log Lookup
  - Depending on how the encryption was structured (e.g., exponential ElGamal), the client may perform a discrete-log search (e.g., baby-step–giant-step) to recover the numeric evaluation from the group element.
- Fair Exchange Outcome
  - The client, having paid or locked tokens, now obtains the genuine data—fulfilling the exchange contract on-chain or off-chain with verifiable correctness and fairness.

Multiclient Model



Cannot reuse sk, vk, ct across clients

- If clients share the data, they can know without the payment
- Must rerandomize sk across clients!

Multi-client VECK Protocol

- Prover saves work by moving parts of the proof gen eration to a preprocessing step.
- No need to generate new ciphertexts and range pro ofs per client!

Evaluations

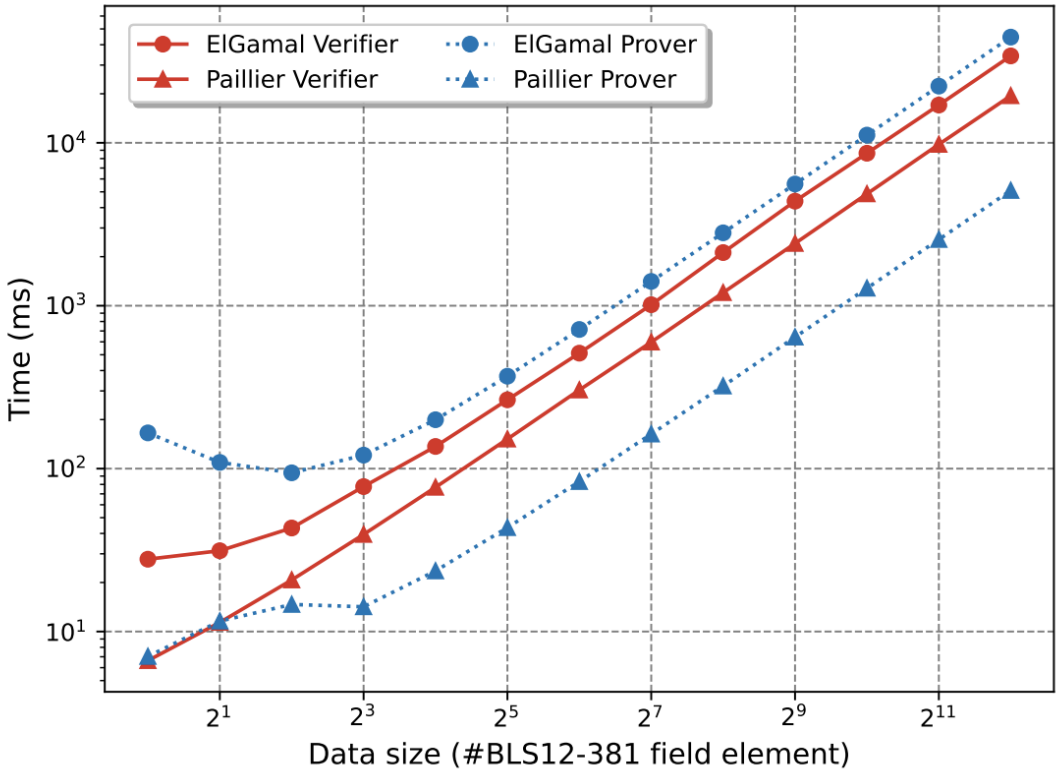
Performance Evaluation

On a Consumer PC Setting

- AMD Ryzen 5 3600 CPU and 8GB RAM

Prover time for 4096 (BLS12-381) field elements (128KB)

- ElGamal Encryption (k=8)
  - Range Proofs + Ciphertexts: 89s
  - Proving the consistency of ciphertexts: < 40ms
  - Ciphertexts & Proofs total size: 1.56MB
  - Proof size constant to the data
- Paillier Encryption
  - 5.09s
  - Ciphertexts & Proofs total size: 6.55MB
  - Proof size linear to the data



On-chain Cost

Transaction	Gas cost		USD cost	
	ElGamal	Paillier	ElG.	Pail.
serverSendsPubKey	158, 449	176, 296	5.11 $	5.68 $
clientLocksPayment	30, 521	30, 521	0.98 $	0.98 $
serverSendsSecKey	73, 692	82, 475	2.37 $	2.65 $
withdrawPayment	43, 836	43, 836	1.41 $	1.41 $

Table 2: EVM gas costs of the smart contract components for $\lambda = 128$ bits of security. Note these costs are constant in the size of the exchanged data, (*cf.* Appendix C.2), in the Exponential Elgamal (*cf.* Figure 2) and Paillier encryption-based FDE schemes (*cf.* Figure 5). For the USD costs, we used the transaction fee 14 GWei and 2, 302.35 USD/ETH exchange rate as of 2024 February 3rd.

Miscellaneous

Can be Implemented on Bitcoin

- No need for turing complete contract

Github

- Link

Thank You