Fair Data Exchange using Homomorphic Encryption

Roxanne Chevalley

School of Computer and Communication Sciences

Semester Project

June 6, 2025

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Prof. Serge Vaudenay
EPFL / LASEC

Abstract

Fair Data Exchange (FDE) protocols address the challenge of ensuring atomic exchange of data between a client and a server. More precisely, the case where a client wants to retrieve data stored on the server. In that case, the client should obtain the data if and only if the server receives payment. A paper, *Atomic and Fair Data Exchange via Blockchain* [1], was recently published and presents a solution to this problem which relies on the blockchain for atomicity and on a new scheme, verifiable encryption under committed keys (VECK), to guarantee data integrity. This new cryptographic scheme has the great properties of having constant sized on-chain communication, being fair and correct. In this project, we propose 5 new FDE protocols that have similar properties and that replace VECK constructions with fully homomorphic encryption (FHE) and hybrid homomorphic encryption (HHE) primitives. Our work demonstrates how homomorphic primitives can yield practical FDE alternatives. We also provide an open-source implementation of the first two protocols and evaluate their performance.

1 Introduction

As presented in *Atomic and Fair Data Exchange via Blockchain*, [1], the cloud data storage market is growing but we lack a fair way for a client to retrieve stored data from a server. This is where Fair Data Exchange protocol comes to the rescue. In this article we present the first fair data exchange protocol, created by Ertem Nusret Tas et al., which guarantees correctness, client-fairness, server-fairness and efficiency. It relies on smart contracts as third parties and verifiable encryption under committed key (VECK), a new cryptographic primitive, for data integrity.
We then explore two new protocols leveraging homomorphic encryption. Homomorphic encryption is used to verify a predicate about the data on its encrypted equivalent. In our case this predicate is the hash of said data. With our new protocols, we obtain the same properties as protocols using VECK minus server fairness. Indeed, these two first protocols obtain only *relaxed* server fairness, since a malicious client could potentially learn a single bit of data. We create three new protocols which obtain perfect server fairness at the cost of higher computational load on the server side.
Finally, we evaluate the performance of the two first protocols; we find that the computation time remains prohibitive for practical deployment. However, we present a few avenues to explore to reduce the computation time and reduce the gap between the theoretical work and real life deployment.

2 Fair Data Exchange (FDE)

The Fair Data Exchange (FDE) [1] problem addresses the atomic swap of a data file for payment. A client wants to retrieve data from a server, the client receives the data if and only if the server receives the agreed upon payment.
As proven by H. Pagnia and F.C. Gärtner in *On the Impossibility of Fair Exchange without a Trusted Third Party* [2] a fair exchange protocol needs a third party, which can take form in a smart contract on the blockchain. One trivial way to use that smart contract could be the following:

1. The client sends some payment to the smart contract alongside a known hash H. $\mathsf{H} = \mathsf{Hash}(\mathsf{data})$, where Hash is a hash function known by the client, the server, and the smart

contract.

2. The server checks that the client sent the correct hash to the smart contract, if not, it aborts the protocol.

3. The server sends data on-chain.

4. The smart contract verifies that $\mathsf{H} = \mathsf{Hash}(\mathsf{data})$. If this holds, the payment is released to the server, and the client can retrieve the data. Otherwise, the protocol is aborted.

The issue with this protocol is that on-chain storage of data is very expensive, making this protocol unfeasible.

In *Atomic and Fair Data Exchange via Blockchain* [1], Ertem Nusret Tas et al. solved this problem by introducing the concept of a blockchain Fair Data Exchange (FDE) protocol. The primary contribution of this work is to exchange the data off-chain rather than on-chain. In order to keep fairness, an encrypted version of the data is sent off-chain from the server directly to the client. The server and the client then use an on-chain protocol to exchange a decrypting key against some payment. The FDE protocol maintains fairness while minimizing blockchain load. The approach relies on the blockchain to enforce atomicity, and uses a new cryptographic primitive called Verifiable Encryption under Committed Key (VECK) to guarantee data integrity before payment release. The FDE protocol features constant-sized on-chain communication (only three transactions). It is done in the following way:

1. The server encrypts the data with a secret key $ct = \mathsf{Enc}(sk, \mathsf{data})$ and sends a public key pk to the smart contract. This public key is a *commitment* to the secret key.

2. The server sends the ciphertext ct and a proof π (generated with VECK) which guarantees the integrity of ct.

3. The client verifies π. If it is valid, it locks payment by sending money to the smart contract.

4. The server sends the secret key sk to the smart contract.

5. The smart contract verifies the validity of sk by using the verification key pk. If the verification is successful, it releases the secret key to the client and the payment to the server. Otherwise, the protocol is aborted.

6. The client recovers its data by computing $\widehat{\mathsf{data}} = \mathsf{Dec}(sk, ct)$.

The paper presents two instantiations of the VECK protocol, one based on ElGamal and the other on Paillier encryption. Both use KZG commitments to ensure data integrity.

2.1 Properties of Fair Data Exchange

- **Correctness:** Honest parties always complete the exchange successfully, the client recovers the exact data matching a previously known commitment, and the server obtains the payment.

- **Client-Fairness:** The server cannot extract payment unless the client obtains the correct decryption key, and thus the data.

- **Server-Fairness:** The client cannot learn any information about the data (beyond negligible leakage) without locking the payment.
- **Efficiency:** The asymptotic communication complexity between the server and client is linear in the number of data blocks.

3 Fully Homomorphic Encryption

A fully homomorphic encryption scheme is a cryptographic scheme that encrypts plaintexts in $\mathcal{P}$ to ciphertexts in $\mathcal{X}$ and vice-versa. It also enables the evaluation of all circuits in the space $\mathcal{C}$ directly on ciphertexts in $\mathcal{X}$. Homomorphic encryption schemes are defined by four algorithms (the following definitions are taken from *A guide to fully homomorphic encryption* [3]):

- $\mathsf{Gen} : \mathbb{N} \rightarrow \mathcal{K}_s \times \mathcal{K}_p \times \mathcal{K}_e$
  $\mathsf{Gen}(1^\lambda) \rightarrow (sk, pk, evk)$: Generates a private/public/evaluation key triple. pk is used for encryption, sk is used for decryption, and evk is used for evaluation. λ is the security parameter.
- $\mathsf{Enc} : \mathcal{K}_p \times \mathcal{P} \rightarrow \mathcal{X}$
  $\mathsf{Enc}(pk, m) \rightarrow c$: Encrypts the message m with the encryption key pk and outputs the corresponding ciphertext c.
- $\mathsf{Dec} : \mathcal{K}_s \times \mathcal{Z} \rightarrow \mathcal{P}$
  $\mathsf{Dec}(sk, c) \rightarrow m$: Decrypts a ciphertext c with the decryption key sk and outputs the plaintext m.
- $\mathsf{Eval} : \mathcal{K}_e \times \mathcal{C} \times \mathcal{Z}^* \rightarrow \mathcal{Y}$
  $\mathsf{Eval}(pk, C, \langle c_1, .., c_n\rangle) \rightarrow c'$: Evaluates a circuit C in $\mathcal{C}$, on a tuple of encrypted inputs to obtain a new ciphertext c'. **Eval()** is what makes homomorphic encryption different from standard public key encryption.

Here $\mathcal{X} \cup \mathcal{Y} = \mathcal{Z}$.

3.1 Properties of Homomorphic Encryption

The properties of fully homomorphic encryption are the following:

- **Correct Decryption:** A FHE scheme (Gen, Enc, Eval, Dec) is said to *correctly decrypt* if for all $m \in \mathcal{P}$,
  $$\Pr\big[\mathrm{Dec}(sk,\ \mathrm{Enc}(pk, m)) \ = \ m\big] \ = \ 1,$$
  where sk and pk are outputs of $\mathsf{Gen}(1^\lambda)$.
- **Correct Evaluation:** A FHE scheme is said to *correctly evaluate* all circuits in $\mathcal{C}$ if for all $c_i \in \mathcal{X}$, where $m_i \leftarrow \mathsf{Dec}(sk, c_i)$, for every $C \in \mathcal{C}$, and some negligible function ϵ,
  $$\Pr\big[\mathrm{Dec}(sk,\ \mathrm{Eval}(evk, C, c_1, .., c_n)) \ = \ C(m_1, ..., m_n)\big] \ = \ 1 - \epsilon(\lambda),$$
  where sk, pk and evk are outputs of $\mathsf{Gen}(1^\lambda)$.
  For simplicity, we say that an FHE scheme (Gen, Enc, Eval, Dec) is *correct* if it has the properties of correct evaluation and correct decryption.

- **Compactness:** A FHE scheme is *compact* if there is a polynomial p, such that for any key-triple (sk, pk, evk) output by $\mathsf{Gen}(1^\lambda)$, any circuit $C \in \mathcal{C}$ and all ciphertexts $c_i \in \mathcal{X}$, the output size $\mathsf{Eval}(evk, C, c_1, .., c_n)$ is not greater than $p(\lambda)$ bits, independent of the circuit size.

4 Protocols

4.1 Protocol I

Protocol I relies on the correct evaluation property of fully homomorphic encryption and on the security properties of cryptographic hash functions and commitments. It only uses homomorphic encryption.

4.1.1 Prerequisites

- The client and the server choose a hash function $\mathsf{Hash} : \mathsf{PreimageSp} \to \mathsf{ImageSp}$ for which the client knows a circuit C_{Hash} which corresponds to this hash function and which can be evaluated on homomorphically encrypted ciphertexts. Formally, for all $C_{hash} \in \mathcal{C}$ and $c \in \mathcal{X}$, where $m \leftarrow \mathsf{Dec}(sk, c)$, and some negligible function ϵ,

  $$\Pr\big[\mathrm{Dec}(sk,\, \mathrm{Eval}(evk, C_{Hash}, c)) \;=\; \mathsf{Hash}(m)\big] \;=\; 1 - \epsilon(\lambda),$$

  where sk, pk and evk are outputs of $\mathsf{Gen}(1^\lambda)$.
  We require the cryptographic hash function Hash to have collision resistance.

- The client initially knows $\mathsf{H} = \mathsf{Hash}(\mathsf{data})$.

- Both the client and the server trust a smart contract deployed on a blockchain.

- The client and server agree on a cryptographic commitment scheme. It consists of two algorithms:

  $$\mathsf{Commit} : \mathsf{ComSp} \to \mathsf{OpenSp}, \quad \mathsf{Open} : \mathsf{OpenSp} \to \mathsf{ComSp} \cup \{\bot\}.$$

  We require Correctness, Biding and Hiding.

4.1.2 Security Assumption

Homomorphic Evaluation Integrity Problem: We assume that the Homomorphic Evaluation Integrity problem is hard. That is, for any probabilistic polynomial time algorithm $\mathcal{A}$, the probability that the following game returns 1 is negligible in λ:

```
HEI(λ) :
    1 : $(sk,\, evk,\, c,\, C) \leftarrow \mathcal{A}(1^\lambda)$
    2 : $H_{ct} \leftarrow \mathsf{Eval}(evk,\, C,\, c)$
    3 : $H \leftarrow \mathsf{Dec}(sk,\, H_{ct})$
    4 : $pt \leftarrow \mathsf{Dec}(sk, c)$
    5 : **return** $1_{H \neq C(pt) \,\wedge\, H \neq \bot}$
```

A function $f(\lambda) = negl(\lambda) \iff \forall n \quad f(\lambda) = \mathcal{O}(\lambda^{-n})$

4.1.3 Protocol Execution Flow

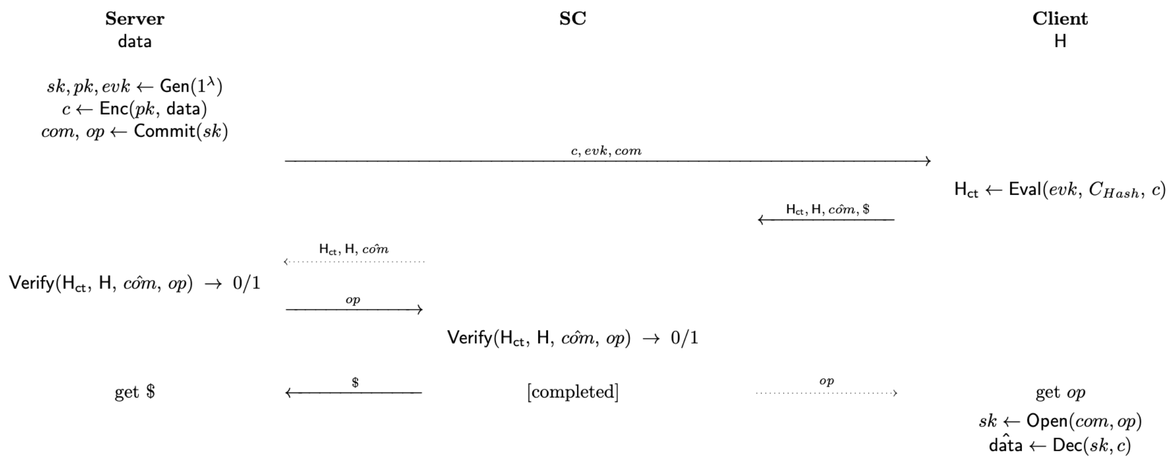

Server
data
$sk, pk, evk \leftarrow \mathsf{Gen}(1^\lambda)$
$c \leftarrow \mathsf{Enc}(pk, \mathsf{data})$
$com, op \leftarrow \mathsf{Commit}(sk)$
SC
Client
H
c, evk, com
$\mathsf{H_{ct}} \leftarrow \mathsf{Eval}(evk, C_{Hash}, c)$
$\mathsf{H_{ct}}, \mathsf{H}, \hat{com}, \$$
$\mathsf{H_{ct}}, \mathsf{H}, \hat{com}$
$\mathsf{Verify}(\mathsf{H_{ct}}, \mathsf{H}, \hat{com}, op) \rightarrow 0/1$
op
$\mathsf{Verify}(\mathsf{H_{ct}}, \mathsf{H}, \hat{com}, op) \rightarrow 0/1$
get \$
\$
[completed]
op
get op
$sk \leftarrow \mathsf{Open}(com, op)$
$\hat{\mathsf{data}} \leftarrow \mathsf{Dec}(sk, c)$

Figure 1: Protocol I - using homomorphic encryption for FDE

The only encryption scheme used in the above protocol is homomorphic encryption. The $\mathsf{Verify}(\mathsf{H_{ct}}, \mathsf{H}, \hat{com}, op) \rightarrow 0/1$ function enables fairness, if this verification rejects on the server side or later on the smart contract side, then the protocol aborts and the client gets its money back.
If the server does not run the $\mathsf{Verify}()$ function and directly sends the opening value to the commitment to the smart contract, the client can give a maliciously crafted value which would make the verification fail on the smart contract. The client would thus get back its money while being able to get the secret key by using the commitment and the opening value retrieved on the blockchain. This breaks server fairness.
If the smart contract doesn't run the $\mathsf{Verify}()$ function, the server can send a dummy opening value to the smart contract and retrieve the payment automatically. With this dummy opening value, the client would not get the secret key and wouldn't be able to decrypt the ciphertext. Thus, it wouldn't get its data. This breaks client fairness.

4.1.4 Protocol I Detailed Algorithms

In the above protocol execution flow, the step that is not fully explicit is the $\mathsf{Verify}()$ function. Its signature is:

- $\mathsf{Verify} : \mathcal{Z} \times \mathsf{ImageSp} \times \mathsf{ComSp} \times \mathsf{OpenSp} \rightarrow 0/1$

Algorithm 1: Protocol I - function for the server and smart contract

```
1 Function Verify($\mathsf{H_{ct}}, \mathsf{H}, \hat{com}, op$):
2     $\hat{sk} \leftarrow \mathsf{Open}(\hat{com}, op)$
3     if $\hat{sk} = \perp$ then
4         return 0
5     $\hat{\mathsf{H}} \leftarrow \mathsf{Dec}(\hat{sk}, \mathsf{H_{ct}})$
6     if $\hat{\mathsf{H}} = \mathsf{H}$ then
7         return 1
8     else
9         return 0
```

4.1.5 Properties of Protocol I

1. **Correctness**
   If the server and client are honest, they both perform correct computation and give expected data to the smart contract. First, on line 3 of the algorithm, we know that $\hat{sk} \neq \perp$ due to the honesty of the client and server in giving correct *com* and *op*. So the algorithm won't return 0 at line 4. Then on line 5 we have:

$$
\begin{aligned}
\hat{\mathsf{H}} &= \mathsf{Dec}(\hat{sk}, \mathsf{H_{ct}}) && (1)\\
&= \mathsf{Dec}(\mathsf{Open}(\hat{com}, op), \mathsf{H_{ct}}) && (2)\\
&= \mathsf{Dec}(\mathsf{Open}(com, op), \mathsf{H_{ct}}) && (3)\\
&= \mathsf{Dec}(\mathsf{Open}(\mathsf{Commit}(sk)), \mathsf{H_{ct}}) && (4)\\
&= \mathsf{Dec}(sk, \mathsf{H_{ct}}) && (5)\\
&= \mathsf{Dec}(sk, \mathsf{Eval}(evk,\ C_{Hash},\ c)) && (6)\\
&= \mathsf{Dec}(sk, \mathsf{Eval}(evk,\ C_{Hash},\ \mathsf{Enc}(pk, \mathsf{data}))) && (7)\\
&= C_{Hash}(\mathsf{data}) && (8)\\
&= \mathsf{Hash}(\mathsf{data}) && (9)\\
&= \mathsf{H} && (10)
\end{aligned}
$$

   Here (3) comes from the honesty of the client, (4) from the honesty of the server, (5) comes from the correctness property of cryptographic commitments. (6) and (7) come from the construction of $\mathsf{H_{ct}}$ and c respectively, (8) comes from the Correct Evaluation property and (9) from the construction of C_{Hash}.

   The condition $\hat{\mathsf{H}} = \mathsf{H}$ (on line 6) will thus be satisfied and both instances of the Verify algorithm will return 1.

   The protocol thus terminates correctly: the server gets paid, and the honest client recovers the correct *op* which enables them to get *sk*.

2. **Client Fairness**
   Given that the smart contract is trusted, and given the construction of our protocol, we know that it is impossible for the server to get any payment without the Verify algorithm returning 1 on the smart contract. However, we want to ensure that if this

function verifies, the client is able to recover a key which correctly decrypts the cipher. More formally, we want the **Malicious Server** problem to be hard. The MS (Malicious Server) problem is hard if for all PPT adversary $\mathcal{S}$, the probability that the following game returns 1 is negligible:

```
$\mathsf{MS}(\lambda)$ :
    $1 : (com,\ evk,\ c,\ C_{Hash},\ H,\ state) \leftarrow \mathcal{S}()$
    $2 : H_{ct} \leftarrow \mathsf{Eval}(evk,\ C_{Hash},\ c)$
    $3 : op \leftarrow \mathcal{S}(state,\ H_{ct})$
    $4 : sk \leftarrow \mathsf{Open}(com,\ op)$
    $5 : pt \leftarrow \mathsf{Dec}(sk,\ c)$
    $6 : \textbf{return } 1_{\mathsf{Verify}(H_{ct},\, H,\, com,\, op)=1\ \wedge\ C_{Hash}(pt)\neq H\ \wedge\ H\neq\perp}$
```

Proof: We prove that if the HEI problem is hard then the MS problem is hard too by constructing $\mathcal{A}$ such that $\mathcal{S}$ wins MS $\implies$ $\mathcal{A}$ wins HEI.

```
$\mathcal{A}()$ :
    $1 : (com,\ evk,\ c,\ C_{Hash},\ H,\ state) \leftarrow \mathcal{S}()$
    $2 : H_{ct} \leftarrow \mathsf{Eval}(evk,\ C_{Hash},\ c)$
    $3 : op \leftarrow \mathcal{S}(H_{ct},\ state)$
    $4 : sk \leftarrow \mathsf{Open}(com,\ op)$
    $5 : \textbf{return } sk,\ evk,\ c,\ C_{Hash}$
```

Indeed, $\mathcal{S}$ wins $\implies$ $\mathsf{Verify}(H_{ct},\ H,\ com,\ op) = 1 \ \wedge\ C_{Hash}(pt) \neq H \implies H = \mathsf{Dec}(sk, H_{ct}) \ \wedge\ C_{Hash}(pt) \neq H \implies \mathsf{Dec}(sk, H_{ct}) \neq C_{Hash}(pt) \implies \mathcal{A}$ wins.

Since in our security assumption we have the hardness of the HHI problem, we can conclude that under this assumption the MS problem is hard too and thus, we have client fairness.

3. **Relaxed Server Fairness**
   Given an honest server $\mathcal{S}$, for all PPT $\mathcal{C}^*$, the probability that $\mathcal{C}^*$ learns anything more than $\mathsf{Hash}(\mathsf{data}) = \mathsf{H}$ and one additional bit of information on data is negligible if $\mathcal{C}^*$ doesn't provide the agreed upon payment. The only scenario where the server doesn't get money from the client is when the Verify function returns 0. In that case the client doesn't pay anything but still gets the information "Verify returned 0". This is why the Server Fairness definition needs to be relaxed for this protocol. We will see how to get strict server fairness in the **Attack on the Protocols** section of this paper.

4.1.6 Comments on Protocol I

Advantages of the protocol

1. The most expensive operation (the Eval) is performed on the client-side. The steps on the server size can be done efficiently, limiting the probability of a successful denial-of-service attack on the server from a malicious client.

2. The off-chain communication is limited to a single message.

3. The on-chain communication is limited to two messages.

Disadvantages of the protocol

1. The sk and $\mathsf{H_{ct}}$ put on the smart contract are longer than what is put on the smart contract in VECK. For example, in the Rust implementation of TFHE, the sk is around 9.2 KB (precisely 75904 bits). Moreover, $\mathsf{H_{ct}}$ is longer than H, in tfhe-rs, 815 KB are needed to send the ciphertext of a 256 bits bitstring.
2. The Eval step is time consuming.
3. While the Verify algorithm can be computed in a time-effective way, it is still more expensive than the operations performed by the smart contract in the VECK protocols.
4. The protocol relies on a strong security assumption that might not be satisfied in concrete homomorphic encryption schemes.

Overall, this first protocol is not really usable as it puts a large amount of data on the smart contract which is very costly.

However, this protocol helps in getting the intuition as to how we can leverage homomorphic encryption for fair data exchange.

4.2 Protocol II

Protocol II tries to address the limitations of Protocol I. For that it uses Hybrid Homomorphic Encryption, which is a mix of symmetric and homomorphic encryption.

4.2.1 Background on Hybrid Homomorphic Encryption

Hybrid Homomorphic Encryption (HHE) is a scheme which reduces bandwidth compared to Homomorphic Encryption (HE). However, it does so at the cost of more expensive computational costs. In our case, the reduction of bandwidth in the server-client communication was not our main concern. However, we will be able to take advantage of this new scheme to send drastically less data to the smart contract. We will pay for this new improvement: the client will have to perform a more expensive computation in the encrypted domain.

Intuition: In a hybrid homomorphic encryption (HHE) setup, the server first encrypts the data with a symmetric cipher and gets the ciphertext ct. Moreover, the server encrypts the symmetric key k under a standard homomorphic scheme and get k_{ct}. The server sends the cipher and the encrypted symmetric key to the client.
The client will then "homomorphically decrypt" the ciphertext ct with the homomorphically encrypted key k_{ct} and perform operations on this "decrypted" ciphertext. The server will then try to convince the client that there is a key k, corresponding to the homomorphic decryption of k_{ct}, which correctly decrypts ct. The client and the server will then use a smart contract to exchange the key k with a payment. The client will then be able to recover the data by decrypting ct with k. [4]

Symmetric Encryption: [5] To fully understand why this works, it is important to recall a few properties about symmetric encryption.
A symmetric encryption scheme is defined by a triple of algorithms:

- $\mathsf{Gen}(1^\lambda) \to (k)$: generates a key $k \in \mathsf{KeySp}$. Here, λ is once again the security parameter.
- $\mathsf{Enc}(k, m) \to ct$: Encrypts a message $m \in \mathsf{MsgSp}$ with the symmetric key k the corresponding ciphertext $ct \in \mathsf{CipherSp}$.
- $\mathsf{Dec}(k, ct) \to m$: Decrypts a ciphertext $ct \in \mathsf{CipherSp}$ with the symmetric key k and outputs the plaintext $m \in \mathsf{MsgSp}$.

which satisfy the following property:

- **Correctness:** for all $k \in \mathsf{KeySp}, m \in \mathsf{MsgSp}, \mathsf{Dec}(k, \mathsf{Enc}(k, x)) = x$.

From now on, to avoid ambiguity, we will rename the symmetric encryption algorithms $\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}$ which will respectively become $\mathsf{Sym.Gen}, \mathsf{Sym.Enc}, \mathsf{Sym.Dec}$, while the four algorithms of homomorphic encryption $\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval}$ will become $\mathsf{HE.Gen}, \mathsf{HE.Enc}, \mathsf{HE.Dec}, \mathsf{HE.Eval}$.

4.2.2 Prerequisites

- The client and server agree on:
  - a public hash function with collision resistance, $\mathsf{Hash} : \mathsf{PreimageSp} \to \mathsf{ImageSp}$,
  - a symmetric-key scheme $\big(\mathsf{Sym.Gen}, \mathsf{Sym.Enc}, \mathsf{Sym.Dec}\big)$

  The client knows a circuit C_{Hash} which corresponds to this hash function and which can be evaluated on homomorphically encrypted ciphertexts. Formally, for all $c \in \mathcal{X}$, where $m \leftarrow \mathsf{HE.Dec}(sk, c)$, and some negligible function ϵ,

  $$\Pr\big[\text{HE.Dec}(sk,\ \text{HE.Eval}(evk, C_{Hash}, c)) \;=\; \mathsf{Hash}(m)\big] \;=\; 1 - \epsilon(\lambda),$$

  where sk, pk and evk are outputs of $\mathsf{HE.Gen}(1^\lambda)$.

- They define a circuit-builder

  $$\mathsf{BuildCirc} : \mathsf{CipherSp} \to \ \mathcal{C}, \quad \mathsf{BuildCirc}(ct) = C_{ct},$$

  such that for every $ct \in \mathsf{CipherSp}$, $C_{ct} = \mathsf{BuildCirc}(ct)$ exactly implements the composition

  $$k \to \mathsf{Hash}\big(\mathsf{Sym.Dec}(k,\ ct)\big),$$

  Formally, for all $k_{ct} \in \mathcal{X}, ct \in \mathsf{CipherSp}, C_{ct} = \mathsf{BuildCirc}(ct)$, where $k \leftarrow \mathsf{HE.Dec}(sk, k_{ct})$ and $m \leftarrow \mathsf{Sym.Dec}(k, ct)$, and some negligible function ϵ,

  $$\Pr\big[\mathsf{HE.Dec}(sk,\ \mathsf{HE.Eval}(evk, C_{ct}, k_{ct})) \;=\; \mathsf{Hash}(m)\big] \;=\; 1 - \epsilon(\lambda),$$

  where sk, pk and evk are outputs of $\mathsf{HE.Gen}(1^\lambda)$ and k is the output of $\mathsf{Sym.Gen}(1^\lambda)$.

- The client and the server define a challenge circuit-builder function

$$\mathsf{BuildChalCirc} : \mathbb{Z}^3 \times \mathsf{ImageSp}^2 \rightarrow \mathcal{C}, \quad \mathsf{BuildChalCirc}(a, b, c, H_1, H_2) \rightarrow C_{Chal},$$

  such that for every $a, b, c \in \mathbb{Z}^3$ and every $H_1, H_2 \in \mathsf{ImageSp}$,
  $C_{Chal} = \mathsf{BuildChalCirc}(a, b, c, H_1, H_2)$ exactly implements

$$v_1,\, v_2 \;\mapsto\; a \,+\, b \,\times (v_1 \,-\, H_1) \,+\, c \,\times\, (v_2 - H_2),$$

  for all $v_1,\, v_2 \in \mathcal{Z}^2$

- The client initially knows $\mathsf{H} = \mathsf{Hash}(\text{data})$.

- Both the client and server trust a smart contract deployed on a blockchain.

4.3 Security Assumption

For this protocol, we assume that from a circuit's evaluation output, the adversary learns nothing about the circuit itself beyond the output value. In our assumption, we require this property to hold only on C_{chal} circuits. We formalize it with the **Challenge Privacy** game. We require the **CP** game to be hard, when there is no secret key sk known to the adversary such that

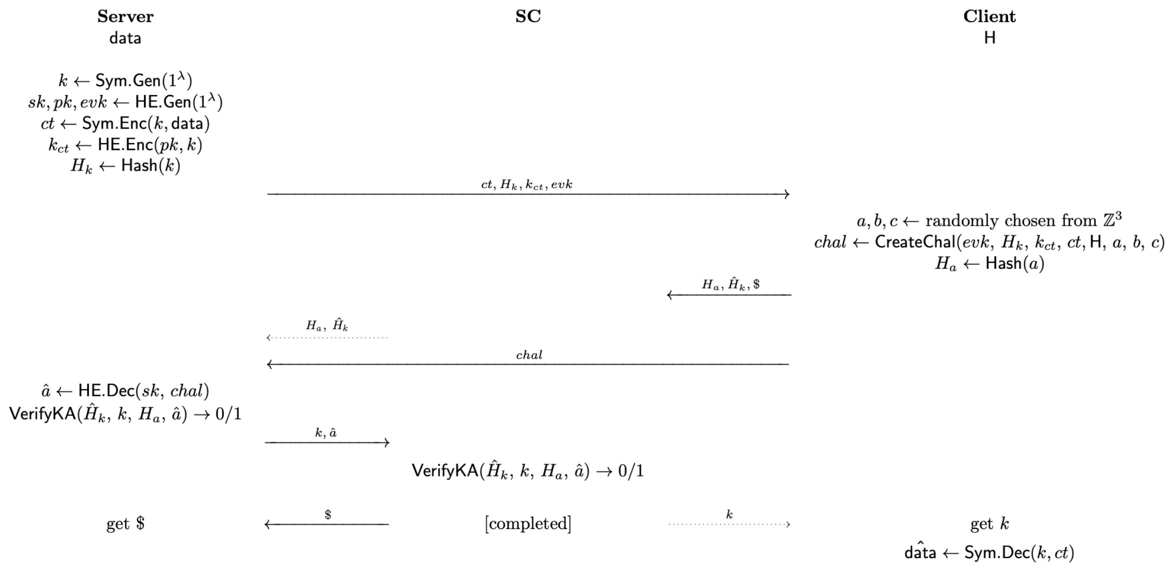
$$\mathsf{Dec}(sk, v_1) = H_1 \;\wedge\; \mathsf{Dec}(sk, v_2) = H_2$$

, for the v_1, v_2, H_1, H_2 first outputed by the adversary. We call this the hard scenario. This means that for any such PPT algorithm $\mathcal{A}$ the probability of winning that game is negligible in λ. For different setups, we make no assumption about the hardness of this problem.

```
$\mathsf{CP}(\lambda)$ :
    1 : $(evk,\, v_1,\, v_2,\, H_1,\, H_2,\, state) \leftarrow \mathcal{A}()$
    2 : Pick $a,\, b,\, c \,\in \mathbb{Z}^3$
    3 : $H_a \leftarrow C_{Hash}(a)$
    4 : $C_{chal} \leftarrow \mathsf{BuildChalCirc}(a, b, c, H_1, H_2)$
    5 : $chal \leftarrow \mathsf{HE.Eval}(evk,\, C_{chal},\, v1,\, v2)$
    6 : $\hat{a} \leftarrow \mathcal{A}(chal,\, H_a,\, state)$
    7 : **return** $1_{\hat{a}=a}$
```

Let $m_1 \leftarrow \mathsf{HE.Dec}(sk, v_1)$, $m_2 \leftarrow \mathsf{HE.Dec}(sk, v_2)$, in the hard scenario $m_1 - H_1 \neq 0$ or $m_1 - H_1 \neq 0$, for all sk known to the adversary. This means that for all x, there are infinitely many a such that $a + b \,\times\, (m_1 - H_1) + c \,\times\, (m_2 - H_2) = x$ has a solution. So, the adversary must have found a from the circuit information hidden in $chal$ and, possibly, the help from H_a. We want the probability of succeeding in this in polynomial time to be negligible.

4.3.1 Protocol Execution Flow



Server
data
SC
Client
H
$k \leftarrow \mathsf{Sym.Gen}(1^\lambda)$
$sk, pk, evk \leftarrow \mathsf{HE.Gen}(1^\lambda)$
$ct \leftarrow \mathsf{Sym.Enc}(k, \mathsf{data})$
$k_{ct} \leftarrow \mathsf{HE.Enc}(pk, k)$
$H_k \leftarrow \mathsf{Hash}(k)$
ct, H_k, k_{ct}, evk
$a, b, c \leftarrow$ randomly chosen from $\mathbb{Z}^3$
$chal \leftarrow \mathsf{CreateChal}(evk,\ H_k,\ k_{ct},\ ct, \mathsf{H},\ a,\ b,\ c)$
$H_a \leftarrow \mathsf{Hash}(a)$
$H_a, \hat{H}_k, \$$
$H_a,\ \hat{H}_k$
$chal$
$\hat{a} \leftarrow \mathsf{HE.Dec}(sk,\ chal)$
$\mathsf{VerifyKA}(\hat{H}_k,\ k,\ H_a,\ \hat{a}) \rightarrow 0/1$
$k, \hat{a}$
$\mathsf{VerifyKA}(\hat{H}_k,\ k,\ H_a,\ \hat{a}) \rightarrow 0/1$
get $\$$
$\$$
[completed]
k
get k
$\widehat{\mathsf{data}} \leftarrow \mathsf{Sym.Dec}(k, ct)$

Figure 2: Protocol II - using hybrid homomorphic encryption for FDE

The above protocol uses the VerifyKA function to ensure fairness. This function works in two steps, it first compares H_a with $\mathsf{Hash}(\hat{a})$, this part only succeeds if there exists a key k whose hash is H_k and which decrypts ct in some data $\widehat{\mathsf{data}}$ whose hash is H. Finally, it compares k and $\hat{H}_k$ to ensure that the submitted key k has the expected hash $\hat{H}_k$.
This function is run once on the server side to ensure server fairness. Indeed, if the server missed this step, the client could just give dummy values to the smart contract, making VerifyKA fail on the smart contract, and getting the precious k while getting its money back. VerifyKA is then run for a second time on the smart contract, this time for client fairness. Without this check, the server could give a dummy k, which wouldn't enable the client to retrieve the data.
If any of these two instances reject, the protocol is aborted.

4.3.2 Protocol Detailed Algorithms

This section aims to give the pseudocode of the two functions that are not fully explicit in the above protocol execution flow. Namely, CreateChal and VerifyKA whose signatures are:

- $\mathsf{CreateChal} : \mathcal{K}_e \times \mathsf{ImageSp} \times \mathcal{X} \times \mathsf{CipherSp} \times \mathsf{ImageSp} \times \mathbb{Z}^3 \rightarrow \mathcal{Y}$
- $\mathsf{VerifyKA} : \mathsf{ImageSp}^2 \times \mathsf{KeySp} \times \mathbb{Z} \rightarrow 0/1$

Algorithm 2: Protocol II - function for the client

```
1 Function CreateChal($evk$, $H_K$, $k_{ct}$, $ct$, $\mathsf{H}$, $a$, $b$, $c$):
2     $C_{ct} \leftarrow \mathsf{BuildCirc}(ct)$
3     $part_c \leftarrow \mathsf{HE.Eval}(evk,\ C_{ct},\ k_{ct})$
4     $part_b \leftarrow \mathsf{HE.Eval}(evk,\ C_{Hash},\ k_{ct})$
5     $C_{Chal} \leftarrow \mathsf{BuildChalCirc}(a,\ b,\ c,\ H_k, \mathsf{H})$
6     $chal \leftarrow \mathsf{HE.Eval}(evk,\ C_{Chal},\ part_b,\ part_c)$;
7     return $chal$
```

Algorithm 3: Protocol II - function for the server and smart contract

```
1 Function VerifyKA($\hat{H}_k$, $k$, $H_a$, $\hat{a}$):
2     $\hat{H}_a \leftarrow \mathsf{Hash}(\hat{a})$
3     if $H_a \neq \hat{H}_a$ then
4         return 0
5     $H_k \leftarrow \mathsf{Hash}(k)$
6     if $H_k = \hat{H}_k$ then
7         return 1
8     else
9         return 0
```

4.3.3 Properties of Protocol II

1. **Correctness**
   If the server and the client are honest, they both perform correct computations and give expected data to the smart contract. We first prove that this implies that $\hat{a} = a$

$$
\begin{aligned}
\hat{a} &= \mathsf{HE.Dec}(sk,\ chal) && (1)\\
&= \mathsf{HE.Dec}(sk, \mathsf{Eval}(evk, C_{Chal}, \mathsf{HE.Eval}(evk, C_{Hash}, k_{ct}), \mathsf{HE.Eval}(evk, C_{ct}, k_{ct}))) && (2)\\
&= C_{Chal}(\mathsf{HE.Dec}(sk, \mathsf{HE.Eval}(evk, C_{Hash}, k_{ct})), \mathsf{HE.Dec}(sk, \mathsf{HE.Eval}(evk, C_{ct}, k_{ct}))) && (3)\\
&= C_{Chal}(C_{Hash}(k),\ C_{ct}(k)) && (4)\\
&= a\ +\ b\ \times (C_{Hash}(k) - H_k)\ +\ c\ \times\ (C_{ct}(k)\ -\ \mathsf{H}) && (5)\\
&= a\ +\ b\ \times (\mathsf{Hash}(k) - H_k)\ +\ c\ \times\ (\mathsf{Hash}(\mathsf{Sym.Dec}(k, ct))\ -\ \mathsf{H}) && (6)\\
&= a\ +\ b\ \times (\mathsf{Hash}(k) - H_k)\ +\ c\ \times\ (\mathsf{Hash}(\mathsf{data})\ -\ \mathsf{H}) && (7)\\
&= a\ +\ b\ \times 0\ +\ c\ \times\ 0 && (8)\\
&= a && (9)
\end{aligned}
$$

   where (2) comes from the construction of *chal*, (3) and (4) come from the Correct Evaluation property, (5) and (6) come from the constructions of $C_{Chal}, C_{Hash}, andC_{ct}$. (7) comes from the correctness property of symmetric encryption and (8) from the construction of H_k and H.

Knowing $\hat{a} = a$, it is easy to see that VerifyKA won't return 0 on line 4. Moreover, if the client is honest, it will send $\hat{H}_k$ to the smart contract such that $\hat{H}_k = H_k$ and thus VerifyKA will return 1 on line 7.

The protocol will be completed successfully and the client will get the key k. Due to the correctness property of symmetric encryption, the client will recover data as expected.

2. **Client Fairness**
Given that the smart contract is trusted, and given the construction of our protocol, we know that it is impossible for the server to get any payment without the VerifyKA algorithm returning 1. However, we want to ensure that if this function verifies, the client gets a key which correctly decrypts the cipher. We want the **Malicious Server II** problem to be hard. The MSII (Malicious Server II) problem is hard if for all PPT $\mathcal{S}^*$, the probability that the following game returns 1 is negligible in λ:

```
MSII($\lambda$) :
    1 : $(evk,\ ct, C_{Hash},\ H,\ H_k,\ k_{ct},\ state) \leftarrow \mathcal{S}()$
    2 : pick $a, b, c \in \mathbb{Z}^3$
    3 : $H_a \leftarrow C_{Hash}(a)$
    4 : $chal \leftarrow \mathsf{CreateChal}(evk,\ H_k,\ k_{ct},\ ct,\ H,\ a,\ b,\ c)$
    5 : $\hat{a},\ k \leftarrow \mathcal{S}(chal,\ H_a,\ state)$
    6 : $pt \leftarrow \mathsf{Sym.Dec}(k, c)$
    7 : return $1_{\mathsf{VerifyKA}(H_k, k, H_a, \hat{a}) \to 1 \ \wedge\ C_{Hash}(pt) \neq H \ \wedge\ H \neq \perp}$
```

Proof : We show that winning the MSII game is equivalent to breaking the collision resistance property of the C_{Hash} function or winning the challenge privacy game under the hard scenario, and that the probability of winning this game is thus negligible. Let us do this by going through several scenarios.

1. Assume $\hat{a} \neq a$, then there is a hash collision and the server was able with H_a to break the collision resistance property of the hash function.
2. Now assume $\hat{a} = a$, but there exists no secret key sk such that

$$\mathsf{Dec}(sk, \mathsf{HE.Eval}(evk, C_{ct}, ct)) = H \ \wedge\ \mathsf{Dec}(sk, \mathsf{HE.Eval}(evk, C_{Hash}, k_{ct})) = H_k$$

   Then, we can reduce this scenario to winning the CP game under the hard scenario. We construct $\mathcal{A}$ such that $\mathcal{S}$ wins MSII with the two conditions above $\implies$ $\mathcal{A}$ wins CP.

```
$\mathcal{A}()$ :
    1 : $(evk,\ ct, C_{Hash},\ H,\ H_k,\ k_{ct},\ state) \leftarrow \mathcal{S}()$
    2 : $C_{ct} \leftarrow \mathsf{BuildCirc}(ct)$
    3 : $part_b \leftarrow \mathsf{HE.Eval}(evk,\ C_{Hash},\ k_{ct})$
    4 : $part_c \leftarrow \mathsf{HE.Eval}(evk,\ C_{ct},\ k_{ct})$
    5 : return $evk,\ part_b,\ part_c,\ H_k,\ H,\ state$
    6 : get $chal,\ H_a,\ state$
    7 : $\hat{a},\ \_ \leftarrow \mathcal{S}(chal,\ H_a,\ state)$
    8 : return $\hat{a}$
```

We need to show that $\hat{a} = a$. Due to our steps, (2) to (7) the challenge created for $\mathcal{A}$, will look the same as the challenge that would have been created for $\mathcal{S}$ with the same a, b, c chosen. So, if $\mathcal{S}$ is able to win with $\hat{a} = a$ (first condition), then $\mathcal{A}$ also wins by returning the correct a. However, we assumed that CP game was hard when $\mathcal{A}$ didn't know of sk such that

$$\mathsf{Dec}(sk, part_b) = H_k \ \wedge \ \mathsf{Dec}(sk, part_c) = H,$$

which is clearly the case here from our second condition. So in this scenario, winning MSII breaks the hardness of the CP game.

3. Final scenario, assume $\hat{a} = a$, and there exist a secret key sk such that

$$\mathsf{Dec}(sk, \mathsf{HE.Eval}(evk, C_{ct}, ct)) = H \ \wedge \ \mathsf{Dec}(sk, \mathsf{HE.Eval}(evk, C_{Hash}, k_{ct})) = H_k$$

Due to the Correct Evaluation Property, this means that there exists a symmetric key k' whose hash is H_k and which decrypts ct such that $\mathsf{Hash}(\mathsf{Sym.Dec}(k, ct)) = H$. Thus in this final scenario, $\mathcal{S}$ only wins if it finds $k'_2 \neq k'$ with the same hash as k'. This is equivalent to breaking collision resistance of the hash function. If the server submits k' the client correctly recover its data, maintaining client fairness.

We have shown that winning the MSII game is equivalent to breaking the preimage resistance property of the C_{Hash} function or winning the challenge privacy game, and that the probability of winning this game is thus negligible.

3. **Relaxed Server Fairness**
Given an honest server $\mathcal{S}$, for all PPT $\mathcal{C}^*$, the probability that $\mathcal{C}^*$ learns anything more than $\mathsf{Hash}(\mathsf{data}) = \mathsf{H}$ and one additional bit of information on data is negligible if $\mathcal{C}^*$ doesn't provide the agreed upon payment. The only scenario where the server doesn't get money from the client is when the $\mathsf{VerifyKA}$ function returns 0. In that case the client doesn't pay anything but still gets the information "$\mathsf{VerifyKA}$ returned 0". This is why the Server Fairness definition needs to be relaxed for this protocol. We will see how to get strict server fairness in the **Attack on the Protocols** section of this paper.

4.3.4 Comments on Protocol II

Advantages of the protocol

1. Less data is put on the smart contract: only 2 hashes, a symmetric key (which is usually shorter than a homomorphic secret key) and $a \in \mathbb{Z}$.

2. The hash operations performed on the smart contract are cheap.

3. The on-chain communication is still limited to only two rounds.

4. The off-chain communication is still only two rounds.

Disadvantages of the protocol

1. The client performs a more expensive operation than in protocol I.

5 Attack on the Protocols

As mentioned in the previous section, in both Protocol I and II, there exists a way for the client to probabilistically gain one additional bit of knowledge about data. This bit of knowledge comes from the verification function run by the server. Indeed, if Verify or $\mathsf{VerifyKA}$ fails, the client will not have to pay any money (as the honest server will abort the protocol). The client will learn that the data it gave made the verification function fail. The client's expected information gain is less than one bit.

5.1 Concrete Examples of the Attack

5.1.1 Protocol I

Instead of running $\mathsf{H_{ct}} \leftarrow \mathsf{Eval}(evk,\, C_{Hash},\, c)$, a malicious client $\mathcal{C}^*$ could run:

$$\mathsf{H_{ct}} \leftarrow \mathsf{Eval}(evk,\, C_{AddHash},\, \mathsf{Eval}(evk,\, C,\, c))$$

where $C_{AddHash}$ is the circuit that implements:

$$v \rightarrow v + \mathsf{H},\ \forall\, v \in \mathcal{Z}$$

and C is any circuit of its choice.
$\mathsf{Dec}(\mathsf{sk},\, \mathsf{H_{ct}}) = \mathsf{Dec}(\mathsf{Eval}(evk,\, C_{AddHash},\, \mathsf{Eval}(evk,\, C,\, c))) = C_{AddHash}(C(\mathsf{Dec}(sk,\, c))) = C(\mathsf{data}) + \mathsf{H} = \mathsf{H} \iff C(\mathsf{data}) = 0.$

Either, $C(\mathsf{data}) = 0$, and Verify returns 1 on the server side and on the smart contract (since the server is honest, the smart contract and server run the protocol with the same values), the client has to pay, and server fairness is not broken. Either $C(\mathsf{data}) \neq 0$, Verify fails on the server side, and the server aborts the protocol. The client thus learns that $C(\mathsf{data}) \neq 0$ without paying anything and breaks strict server fairness. To maximize its information gain, the malicious client should choose a circuit C such that $C(\mathsf{data}) = 0$ with probability $\frac{1}{2}$.

5.2 Protocol II

In protocol II, instead of running $chal \leftarrow \mathsf{CreateChal}(evk,\, H_k,\, k_{ct},\, ct, \mathsf{H},\, a)$, a malicious client could run:

$$chal \leftarrow \mathsf{Eval}(evk,\, C_{AddA},\, \mathsf{Eval}(evk,\, C,\, c))$$

where, this time, C_{AddA} implements:

$$v \rightarrow v \,+\, a,\ \forall\, v \in \mathcal{Z}$$

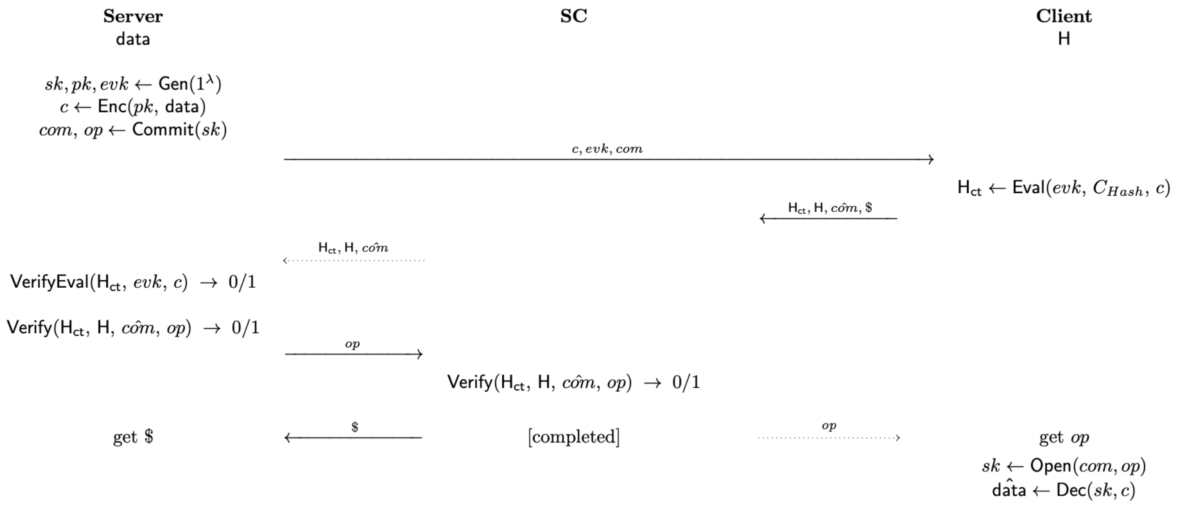
Same as for Protocol I, either, $C(\mathsf{data}) = 0$, $\mathsf{VerifyKA}$ returns 1 on the server side and on the smart contract, the client has to pay and server fairness is not broken. Either, $C(\mathsf{data}) \neq 0$, or $\mathsf{VerifyKA}$ fails on the server side, and the server aborts the protocol. The client once again learns that $C(\mathsf{data}) \neq 0$ without paying anything and breaks server fairness.

5.3 Defense

The way to avoid this attack is for the server to check that the values given by the client are the expected ones. We first present two variants of Protocols I and II that have server fairness. Both of these variants assume that the homomorphic evaluation is a deterministic function. We then present Protocol III, which is a variant to Protocol II with server fairness that doesn't rely on the determinism of homomorphic evaluation.

5.4 Protocol I.ii - a safer version of Protocol I

The **Prerequisites** are the same as the ones of the original Protocol I, plus the fact that homomorphic evaluation should be deterministic. The **Execution flow** is slightly different:



Server
data
SC
Client
H
$sk, pk, evk \leftarrow \mathsf{Gen}(1^\lambda)$
$c \leftarrow \mathsf{Enc}(pk, \mathsf{data})$
$com, op \leftarrow \mathsf{Commit}(sk)$
c, evk, com
$\mathsf{H_{ct}} \leftarrow \mathsf{Eval}(evk, C_{Hash}, c)$
$\mathsf{H_{ct}}, \mathsf{H}, \hat{com}, \$$
$\mathsf{H_{ct}}, \mathsf{H}, \hat{com}$
$\mathsf{VerifyEval}(\mathsf{H_{ct}}, evk, c) \rightarrow 0/1$
$\mathsf{Verify}(\mathsf{H_{ct}}, \mathsf{H}, \hat{com}, op) \rightarrow 0/1$
op
$\mathsf{Verify}(\mathsf{H_{ct}}, \mathsf{H}, \hat{com}, op) \rightarrow 0/1$
get \$
\$
[completed]
op
get op
$sk \leftarrow \mathsf{Open}(com, op)$
$\hat{\mathsf{data}} \leftarrow \mathsf{Dec}(sk, c)$

Figure 3: Protocol I.ii - Protocol I with server fairness

The difference from Protocol I comes from the VerifyEvals function. For completeness, we also provide here the algorithm for Verify even though it is the same as in Protocol I. Here are the signatures and pseudo-codes of both algorithms:

- $\mathsf{Verify} : \mathcal{Z} \times \mathsf{ImageSp} \times \mathsf{ComSp} \times \mathsf{OpenSp} \rightarrow 0/1$
- $\mathsf{VerifyEval} : \mathcal{Z} \times \mathcal{K}_e \times \mathcal{C} \rightarrow 0/1$

Algorithm 4: Protocol I.ii - function for the server and smart contract

```
1 Function Verify($\mathsf{H_{ct}}, \mathsf{H}$, $\hat{com}$, $op$):
2     $\hat{sk} \leftarrow \mathsf{Open}(\hat{com}, op)$
3     if $\hat{sk} = \perp$ then
4         return 0
5     $\hat{\mathsf{H}} \leftarrow \mathsf{Dec}(\hat{sk}, \mathsf{H_{ct}})$
6     if $\hat{\mathsf{H}} = \mathsf{H}$ then
7         return 1
8     else
9         return 0
```

Algorithm 5: Protocol I.ii - function for the server

```
1 Function VerifyEval($\mathsf{H_{ct}}$, $evk$, $c$):
2     $\hat{\mathsf{H}}_{\mathsf{ct}} \leftarrow \mathsf{Eval}(evk, C_{Hash}, c)$
3     if $\hat{\mathsf{H}}_{\mathsf{ct}} = \mathsf{H_{ct}}$ then
4         return 1
5     else
6         return 0
```

With this new check, the client can't maliciously craft his evaluations anymore to obtain information. If the client tries to do so, he will get caught and the protocol will be aborted independently of how they crafted the submitted values and more importantly, independently of whether the Verify function would have returned 1 or 0. We have now gained **Server Fairness**. **Correctness** and **Client Fairness** can be proven exactly as in Protocol I.

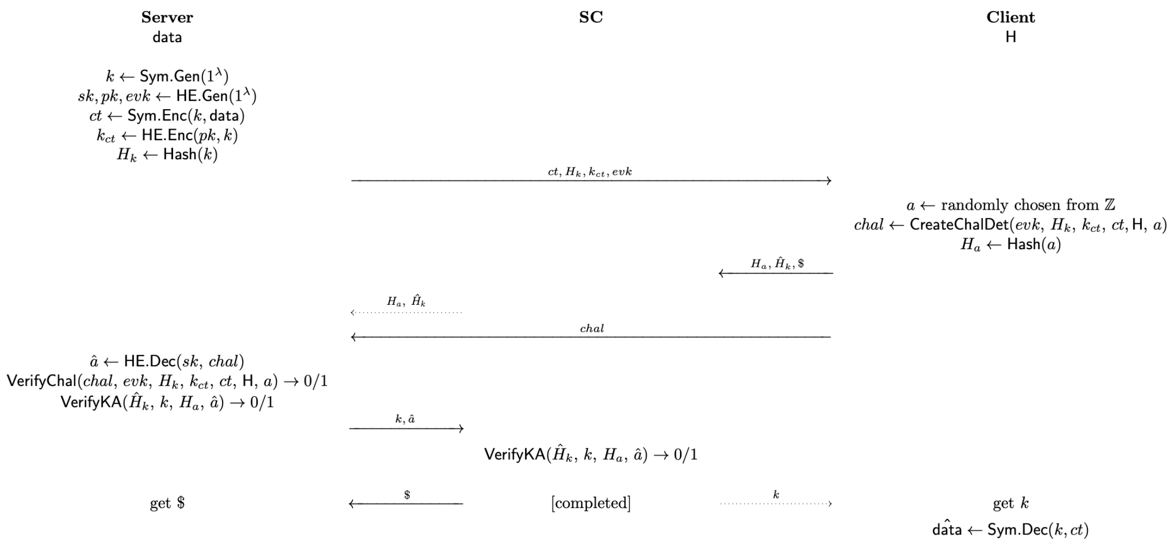
5.5 Protocol II.ii - a safer version of Protocol II

We will follow the same idea as in Protocol I.ii to get server fairness in this updated version of Protocol I.
We add two **Prerequisites** to the ones of the original Protocol II:

- Homomorphic evaluation is a deterministic operation.
- The client and the server agree on a pseudo-random number generator function PRNG, which generates a random number $r \in \mathbb{Z}$ from a seed $s \in \mathbb{Z}$.

$$\mathsf{PRNG} : \mathbb{Z} \rightarrow \mathbb{Z}$$

The **Execution flow** is slightly different:



Server
data
SC
Client
H
$k \leftarrow \mathsf{Sym.Gen}(1^\lambda)$
$sk, pk, evk \leftarrow \mathsf{HE.Gen}(1^\lambda)$
$ct \leftarrow \mathsf{Sym.Enc}(k, \mathsf{data})$
$k_{ct} \leftarrow \mathsf{HE.Enc}(pk, k)$
$H_k \leftarrow \mathsf{Hash}(k)$
ct, H_k, k_{ct}, evk
$a \leftarrow$ randomly chosen from $\mathbb{Z}$
$chal \leftarrow \mathsf{CreateChalDet}(evk,\ H_k,\ k_{ct},\ ct, \mathsf{H},\ a)$
$H_a \leftarrow \mathsf{Hash}(a)$
$H_a, \hat{H}_k, \$$
$H_a,\ \hat{H}_k$
$chal$
$\hat{a} \leftarrow \mathsf{HE.Dec}(sk,\ chal)$
$\mathsf{VerifyChal}(chal,\ evk,\ H_k,\ k_{ct},\ ct,\ \mathsf{H},\ a) \rightarrow 0/1$
$\mathsf{VerifyKA}(\hat{H}_k,\ k,\ H_a,\ \hat{a}) \rightarrow 0/1$
$k, \hat{a}$
$\mathsf{VerifyKA}(\hat{H}_k,\ k,\ H_a,\ \hat{a}) \rightarrow 0/1$
get \$
\$
[completed]
k
get k
$\widehat{\mathsf{data}} \leftarrow \mathsf{Sym.Dec}(k, ct)$

Figure 4: Protocol II.ii - Protocol II with server fairness

The differences from Protocol II come from the VerifyChal and the VerifyChalDet functions. VerifyChalDet is an equivalent to VerifyChal which depends only on a instead of a, b, c. This is useful to verify *chal* knowing only a. For completeness, we also provide here the algorithms for VerifyKA even though it is the same as in Protocol II. Here are the signatures and pseudocodes of the three algorithms:

- CreateChalDet : $\mathcal{K}_e \times \mathsf{ImageSp} \times \mathcal{X} \times \mathsf{CipherSp} \times \mathsf{ImageSp} \times \mathbb{Z} \rightarrow \mathcal{Y}$
- VerifyKA : $\mathsf{ImageSp}^2 \times \mathsf{KeySp} \times \mathbb{Z} \rightarrow 0/1$
- VerifyChal : $\mathcal{Y} \times \mathcal{K}_e \times \mathsf{ImageSp} \times \mathcal{X} \times \mathsf{CipherSp} \times \mathsf{ImageSp} \times \mathbb{Z} \rightarrow 0/1$

Algorithm 6: Protocol II - function for the client

```
1 Function CreateChalDet($evk$, $H_K$, $k_{ct}$, $ct$, H, $a$):
2     $C_{ct} \leftarrow \mathsf{BuildCirc}(ct)$
3     $part_c \leftarrow \mathsf{HE.Eval}(evk,\ C_{ct},\ k_{ct})$
4     $part_b \leftarrow \mathsf{HE.Eval}(evk,\ C_{Hash},\ k_{ct})$
5     $b \leftarrow \mathsf{PRNG}(a)$
6     $c \leftarrow \mathsf{PRNG}(b)$
7     $C_{Chal} \leftarrow \mathsf{BuildChalCirc}(a,\ b,\ c,\ H_k, \mathsf{H})$
8     $chal \leftarrow \mathsf{HE.Eval}(evk,\ C_{Chal},\ part_b,\ part_c)$;
9     return $chal$
```

Algorithm 7: Protocol II - function for the server and smart contract

```
1 Function VerifyKA($\hat{H}_k$, $k$, $H_a$, $\hat{a}$):
2     $\hat{H}_a \leftarrow$ Hash($\hat{a}$)
3     if $H_a \neq \hat{H}_a$ then
4         return 0
5     $H_k \leftarrow$ Hash($k$)
6     if $H_k = \hat{H}_k$ then
7         return 1
8     else
9         return 0
```

Algorithm 8: Protocol II.ii - function for the server

```
1 Function VerifyChal($chal$, $evk$, $H_k$, $k_{ct}$, $ct$, H, $a$):
2     $\hat{chal} \leftarrow$ CreateChalDet($evk$, $H_k$, $k_{ct}$, $ct$, H, $a$)
3     if $\hat{chal} = chal$ then
4         return 1
5     else
6         return 0
```

As in protocol I.ii, we now ensure that if client tries to trick the server, it will get caught. The server will thus abort the protocol, regardless of whether VerifyKA would have returned 1 or 0. We have now gained **Server Fairness**. **Correctness** and **Client Fairness** can be proven exactly as in Protocol II.

5.6 Protocol III - a variant of Protocol II.ii

In this protocol, we assume that the HE.Eval function is a probabilistic function. We emphasize this by now writing HE.Eval(evk; r, C, c).

In this protocol, the **Prerequisites** are the same as in Protocol II, but we add the fact that the client and server have to agree on a cryptographic commitment scheme with correctness, hiding and binding as in Protocol I.

5.6.1 Protocol Execution Flow

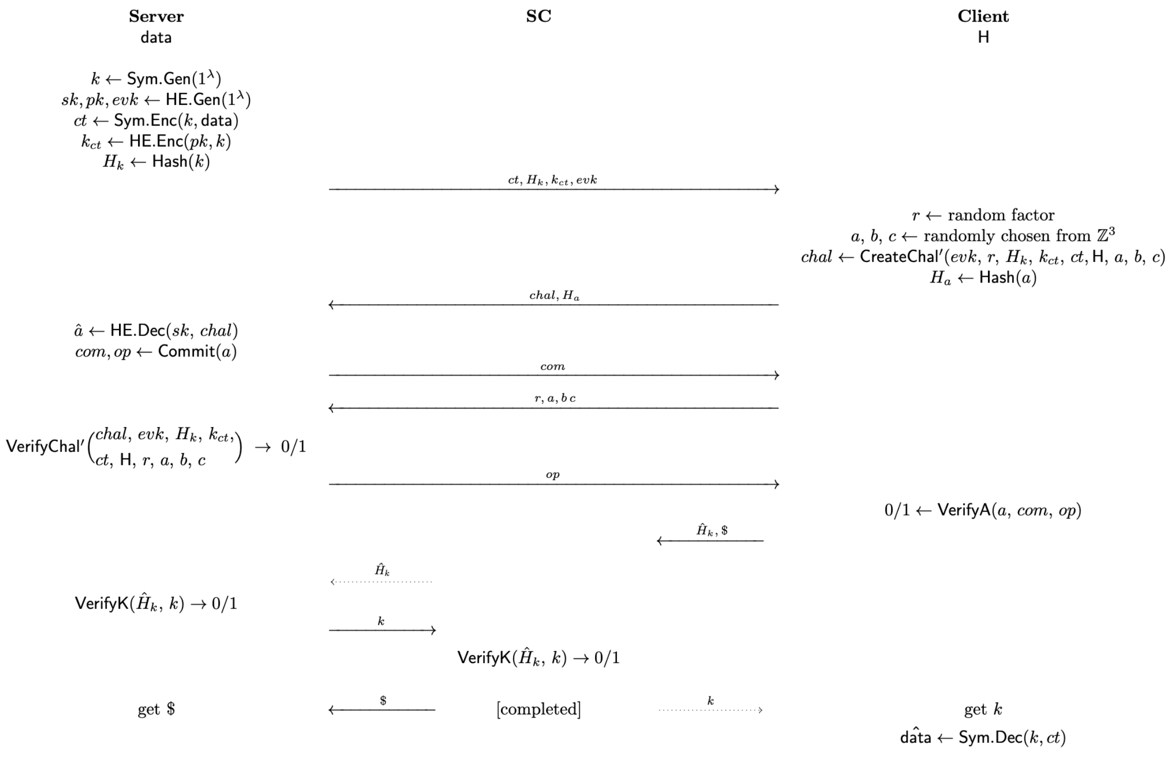

Server
data
SC
Client
H
$k \leftarrow$ Sym.Gen(1^λ)
$sk, pk, evk \leftarrow$ HE.Gen(1^λ)
$ct \leftarrow$ Sym.Enc(k, data)
$k_{ct} \leftarrow$ HE.Enc(pk, k)
$H_k \leftarrow$ Hash(k)
ct, H_k, k_{ct}, evk
$r \leftarrow$ random factor
$a, b, c \leftarrow$ randomly chosen from $\mathbb{Z}^3$
$chal \leftarrow$ CreateChal′(evk, r, H_k, k_{ct}, ct, H, a, b, c)
$H_a \leftarrow$ Hash(a)
$chal, H_a$
$\hat{a} \leftarrow$ HE.Dec(sk, $chal$)
$com, op \leftarrow$ Commit(a)
com
$r, a, b\, c$
VerifyChal′$\begin{pmatrix} chal,\ evk,\ H_k,\ k_{ct}, \\ ct,\ \mathsf{H},\ r,\ a,\ b,\ c \end{pmatrix} \rightarrow 0/1$
op
$0/1 \leftarrow$ VerifyA(a, com, op)
$\hat{H}_k$, \$
$\hat{H}_k$
VerifyK($\hat{H}_k$, k) $\rightarrow 0/1$
k
VerifyK($\hat{H}_k$, k) $\rightarrow 0/1$
get \$
\$
[completed]
k
get k
$\hat{\text{data}} \leftarrow$ Sym.Dec(k, ct)

Figure 5: Protocol III - Protocol II variant with server fairness and probabilistic evaluation

This new protocol is very similar to Protocol II.ii, with the difference that it doesn't assume that homomorphic evaluation is deterministic and that the challenge happens off-chain.

5.6.2 Protocol Detailed Algorithms

This section aims to give the pseudocode of the three functions that are not fully explicit in the above protocol execution flow. Namely, CreateChal′, VerifyChal′ and VerifyA and VerifyK. While CreateChal′ and VerifyChal′ are very close to their equivalent CreateChal, VerifyChal, we still provide their signatures and pseudo-code for completeness. We name ProbSp the space of the probabilistic elements needed for HE.Eval:

- CreateChal′ : $\mathcal{K}_e \times$ ProbSp $\times$ ImageSp $\times\ \mathcal{X} \times$ CipherSp $\times$ ImageSp $\times\ \mathbb{Z}^3 \rightarrow \mathcal{Y}$
- VerifyChal′ : $\mathcal{Y} \times \mathcal{K}_e \times$ ImageSp $\times\ \mathcal{X} \times$ CipherSp $\times$ ImageSp $\times$ ProbSp $\times\ \mathbb{Z}^3 \rightarrow 0/1$
- VerifyA : $\mathbb{Z} \times$ ComSp $\times$ OpenSp $\rightarrow 0/1$
- VerifyK : ImageSp $\times$ KeySp $\rightarrow 0/1$

Algorithm 9: Protocol III - function for the client

```
Function CreateChal′($evk$, $r$, $H_K$, $k_{ct}$, $ct$, H, $a$, $b$, $c$):
    $C_{ct} \leftarrow$ BuildCirc($ct$)
    $part_c \leftarrow$ HE.Eval($evk$; $r$, $C_{ct}$, $k_{ct}$)
    $part_b \leftarrow$ HE.Eval($evk$; $r$, $C_{Hash}$, $k_{ct}$)
    $C_{Chal} \leftarrow$ BuildChalCirc($a$, $b$, $c$, $H_k$, H)
    $chal \leftarrow$ HE.Eval($evk$; $r$, $C_{Chal}$, $part_b$, $part_c$);
    return $chal$

Function VerifyA($a$, $com$, $op$):
    $\hat{a} \leftarrow$ Open($com, op$)
    if $\hat{a} = a$ then
        return 1
    else
        return 0
```

Algorithm 10: Protocol III - function for the server

```
Function VerifyChal′($chal$, $evk$, $H_k$, $k_{ct}$, $ct$, H, $r$, $a$, $b$, $c$):
    $\hat{chal} \leftarrow$ CreateChal′($evk$; $r$, $H_k$, $k_{ct}$, $ct$, H, $a$, $b$, $c$)
    if $\hat{chal} = chal$ then
        return 1
    else
        return 0
```

Algorithm 11: Protocol III - function for the server and smart contract

```
Function VerifyK($\hat{H}_k$, $k$):
    $H_k \leftarrow$ Hash($k$)
    if $H_k = \hat{H}_k$ then
        return 1
    else
        return 0
```

5.6.3 Properties of Protocol III

1. **Correctness**
   Correctness follows the same proof as Protocoll II. We just have to add the fact that since the server is honest, then Open$(com, op) = \hat{a}$ and that VerifyKA is split in two steps: VerifyA and VerifyK.

2. **Client Fairness**
   Client fairness also follows the same ideas as with Protocol II. We just have to add that the binding property of the cryptographic commitment ensures that a malicious server can not change its guessed $\hat{a}$ after the client has revealed a. Once again, the proof

should be tweaked a bit, since VerifyKA was split in VerifyK and VerifyA. However, with that in mind, the proof of client fairness still follows the same idea.

3. **Server Fairness**
   As in Protocol II.ii VerifyChal′ spots any malformed values submitted by the client. The server aborts on such cheating attempt, ensuring Server Fairness.

5.7 Trade off between the Protocols

We now have two families of protocols, those which ensure only relaxed server fairness, and those which guarantee strict server fairness.
While ensuring strict server fairness is important, this latter group of protocols are way more expensive on the server side, as it forces the server to perform homomorphic evaluation, which is the most expensive operation in our protocols. This makes the servers running these protocols more vulnerable to denial-of-service attacks.
Moreover a direct consequence of the bigger server-side computation cost, is that the server will incur higher operational costs whenever a client requests. The server may thus charge buyers an increased fee for retrieving their results. While this compensates the server's investment, it may be perceived as unfair by honest clients.

Another solution for server fairness would be to let the client gain one bit of information by maliciously crafting the submitted values, but making sure that if Verify rejects on the smart contract, a part of the client's money (which must equal the economic value of obtaining a single bit of information) is given to the server. To avoid that the server gains from this new safety by making sure Verify fails on the smart contract and gaining the deposit with no effort (which would break client fairness) we can rely on the server reputation. This follows the idea presented in the **Server griefing** paragraph on the last page of the *Atomic and Fair Data Exchange via Blockchain* paper [1].
The best solution might be evaluated on a case-by-case basis, depending on how much the client and server trust each other, and the impact of gaining one bit of information on `data`.

6 Implementation of the Protocols

We decided to implement Protocol I and Protocol II in Rust using the zama's tfhe-rs's library [6]. The codebase, as well as guides for how to run or evaluate the protocols can be found on GitHub. [1]

The two main primitives needed for our protocols are :

- A homomorphic implementation of a hash function.
- An implementation of a hybrid homomorphism scheme.

6.1 TFHE

TFHE, for *Fast Fully Homomorphic Encryption over the Torus*, is a fully homomorphic scheme based on lattice and whose security comes from the Learning-With-Error (LWE)

[1] `https://github.com/roxannecvl/fde_homomorphic`

problem. TFHE has a highly optimized bootstrapping operation and can evaluate arbitrary Boolean circuits. [7].

6.2 Homomorphic hash

Zama's library already has an example of a homomorphic hash implementation, namely the implementation of SHA256. It can be found on their repository at `tfhe/examples/sha256_bool` [2]. However, we decided to implement a different hash function, SHA3-256, which also takes as input boolean strings, but has a different structure than SHA2 and uses less additions (which is already a fairly expensive homomorphic operation using the tfhe-rs library on a boolean string). SHA3-256 uses Keccak-f permutations [8], which we implemented homomorphically. We took advantage of some boolean operation helper functions created for SHA2 in our SHA3 implementation.
The homomorphic evaluation of a hash using our new implementation of SHA3-256 is around 500s for 128 bytes (on a Mac M1, with 8 GB of RAM) and grows linearly. While this is extremely slow, it is still about twice faster than the SHA256 implementation that was available (around 1000s for 128 bytes on a Mac M1).

6.3 Hybrid Homomorphism scheme

Once again, Zama's library already has an implementation for a hybrid homomorphic scheme which we used in Protocol II. It can be found on their GitHub repository, at `apps/trivium`[3]. Trivium is a stream cipher based on block cipher construction. It uses three registers, which add-up to an internal state of 288 bits and which are initialized with a 80 bit secret key and an 80 bit IV. To generate the key stream, 15 specific bits of the state are used, they both output one 1 bit of the keystream and update the internal state. The operations used for the key generation are bitwise XOR and AND, making it possible to use it homomorphically in the boolean API of tfhe-rs. [9]. This cipher was originally invented without homomorphic encryption in mind, however in *Trivial Transciphering With Trivium and TFHE* [10], the authors study how this scheme can be leveraged for the tfhe-rs library. While keeping the logic intact, we had to slightly rewrite the code, to make it compatible with the boolean API used for the SHA3 hash.

6.4 Performance Evaluation

For Protocols I and II, we created 3 binaries, a client, a server and a smart contract that communicate together via TCP sockets. For different data sizes, we evaluated the computation cost as well as the communication cost (off-chain and on-chain). The results of the computation and communication costs are presented in the graphs below. These evaluations were done by running three times each protocol on different data sizes and averaging the costs for each data size. The evaluations were run an on Macbook M1-chip with 8GB of RAM. Here are the results:

[2]`https://github.com/zama-ai/tfhe-rs/tree/main/tfhe/examples/sha256_bool`
[3]`https://github.com/zama-ai/tfhe-rs/tree/main/apps/trivium`

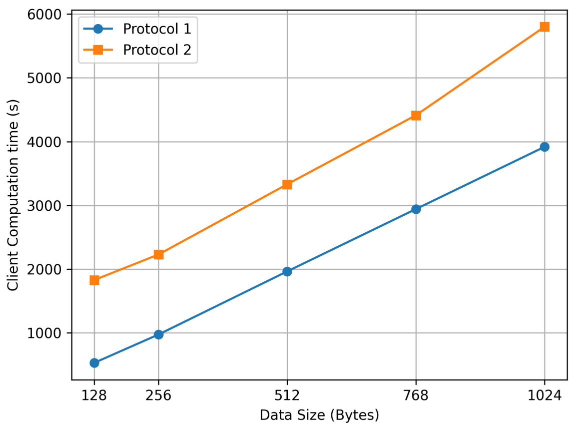


Figure 6: Client computation time (s) as a function of data size (bytes)

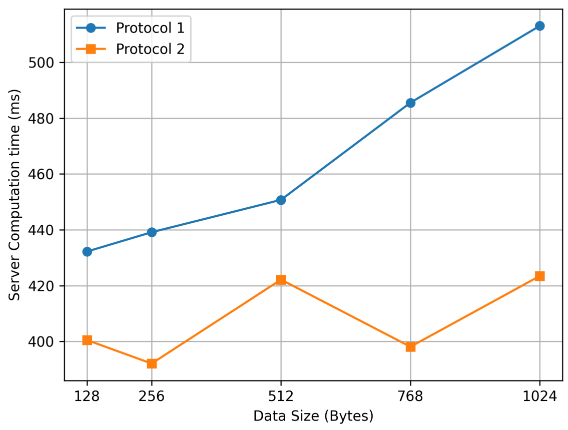


Figure 7: Server computation time (ms) as a function of data size (bytes)

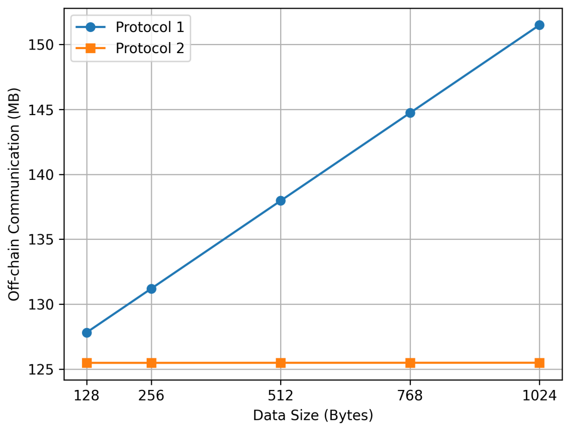


Figure 8: Off-chain computation cost (MB) as a function of data size (bytes)

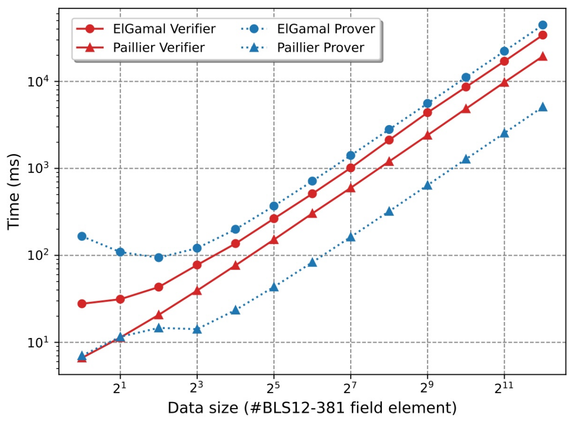


Figure 9: Client-Server (Verifier-Prover) computation cost (ms) in the VECK schemes

Due to the high computation cost of our protocols we only evaluate them on small data sizes. We can however see that the costs grows linearly with the data size. We provide the graphs of computation cost of VECK from the *Atomic and Fair Data Exchange via Blockchain* paper for comparison.

From the graphs, we can see that the client's and server's computation costs for Protocol I grow linearly as approximately

$$T_{ClientI}(n) \approx 3.85\,n \quad \text{(seconds)}, \quad T_{ServerI}(n) \approx 420 + 0.098\,n \quad \text{(milliseconds)}$$

For Protocol II we have,

$$T_{ClientII}(n) \approx 1300 + 4\,n \quad \text{(seconds)}, \quad T_{ServerII}(n) \approx 410 + 0.00191\,n \quad \text{(milliseconds)},$$

where n is the data size in bytes.

In Protocol I, the client spends almost all of his time computing the hash homomorphically and the server spends most of his time encrypting the data homomorphically, the homomorphic decryption time is negligible.
In Protocol II, the client spends 1300 seconds hashing the encrypted symmetric key homomorphically (∼490s) and evaluating the C_{chal} circuit (∼710s) (homomorphic evaluation of $a + b \times part_b + c \times part_c$). The 4 seconds per bytes are spent homomorphically decrypting the data (∼15% of the remaining time) and then evaluating its hash (∼85% of the remaining time). The server spends his time homomorphically encrypting the symmetric key, symmetrically encrypting the data and running VerifyKA.

As a comparison in the VECK protocol, the prover's (= server) and verifier's (=client) computation time for the ElGamal and Paillier instance are approximately:

$$T_{VerifierElGamal}(n) \approx 0.25\,n \quad \text{(milliseconds)}, \quad T_{ProverElGamal}(n) \approx 0.36\,n \quad \text{(milliseconds)}$$

$$T_{VerifierPaillier}(n) \approx 0.15\,n \quad \text{(milliseconds)}, \quad T_{ProverPaillier}(n) \approx 0.038\,n \quad \text{(milliseconds)}$$

where n is once again the data size in bytes and we use the conversion 1 BLS12-381 field element ≈ 32 bytes used in the VECK article.

While our clients' computation costs are significantly higher than VECK's, our server-side costs remain competitive. In fact, for Protocol II, the slope of our server cost line is smaller than VECK's lowest slope. This shifts most of the workload to the client, reducing the burden on the server and making denial-of-service attacks more difficult to carry out.

The off-chain communication cost grows linearly with the size of the data, with a big overhead for the homomorphic key that has to be sent and whose size doesn't depend on the data size. The relationship is approximately

$$C_{Off-chainI}(n) \approx 130'479'556 + 27'710\,n \quad \text{(bytes)}$$

$$C_{Off-chainII}(n) \approx 131'575'020 + 8.5\,n \quad \text{(bytes)}$$

for Protocol I and Protocol II respectively, where n is once again the data size in bytes. For Protocol I, this represents an overhead of 124 MB and then 27 KB per byte, for Protocol II, it represents 125 MB of overhead and then 8 bytes per byte. This is slightly more than expected as the messages were serialized before being sent. Without this overhead, we should get a slope of 1 for Protocol II, since the Trivium ciphertext has the same size as the plaintext, and there is no other variant factor impacting the off-chain communication in that protocol.

The on-chain computation and communication costs graphs are not provided as the costs are constant (you may however see the graphs on our GitHub repository in the evaluation folder [4]). In protocol I, the computation time on the smart contract is around 100µs to 200µs and less than 10µs for protocol II. The communication costs are of 844'232 bytes (where H_{ct} takes 834'568 bytes (about 815 KB), H and the commitment 72 bytes each, and the opening value for the commitment 9520 bytes) for Protocol I and 496 bytes for Protocol II. This is slightly more

[4] `https://github.com/roxannecvl/fde_homomorphic/tree/main/evaluation`

than the effectively used bytes, as messages are serialized and sent with their sizes as a header.

In real life, storing 844,232 bytes on a ETH smart contract, is infeasible (on EVM the current limit is 24,576 bytes) the load would have to be split, and the cost would be 183,046,400 gas ($1948.54 in June 2025 with gas price being 4 Gwei). On the other side, storing 496 bytes on a ETH smart contract costs only 131,200 gas ($1.40 in June 2025 with gas price being 4 Gwei).

6.5 Perfect Server Fair Protocols

We didn't implement the three perfectly server fair protocols, however, TFHE-rs fulfills the deterministic evaluation requirement, and the server would "just" have to run the same things as the client, making the total protocol execution time double.

7 Future Work

We leave the following open problems and improvements for future work:

Hash Optimization It would be great to try and implement hash functions homomorphically that are more suited for homomorphic computations, by having a low multiplicative depth. An interesting candidate would be the Poseidon hash function [11], which has few constraints per message. This function works over a field, and would thus likely need to be implented using another homomorphic encryption library (such as BFV which performs exact computations over finite fields).

Different Hybrid Homomorphic Encryption Schemes While Trivium is easy to use, it incurs high cost. It would be interesting to try and compare our protocols with different hybrid homomorphism schemes. As an example, the goal of the 7th season of zama's bounty programs (which ended in April 2025) was to implement a homomorphic AES-128. An other example is the Pasta scheme, presented in *Pasta: A Case for Hybrid HomomorphicEncryption* [4] but once again, this works over a field and is implemented using SEAL and HElib which internally uses the BFV and BGV homomorphic encryption schemes.

Different Predicate Another direction the work could follow is to test a different predicate on the data (i.e. not its hash), as an example we could imagine evaluating polynomial commitments on the homomorphically encrypted data instead of its hash.

8 Conclusion

Despite the fact that the current computation costs of our proposed FDE protocols remain too high for immediate, large-scale deployment, our work demonstrates how homomorphic encryption can be leveraged to achieve Fair Data Exchange. We have also shown that it is possible to transfer the most significant part of the workload onto clients' local environments. In particular, our Protocol II ensures that the server and smart contract require only minimal computation to validate the exchange (there are no homomorphic operations on the smart contract's side) and the on-chain communication remains relatively low. This, in turn, directly limits the transaction fees and gas costs that a client must pay to retrieve data, making the overall system more economically attractive.

Moreover, we presented alternatives which have perfect server fairness, when relaxed server fairness is not sufficient. This comes at the cost of higher computation costs on the server side. The relaxed-fairness versions are likely sufficient in most scenarios, more specifically when the stakes are quite low or when supplemented by economic penalty mechanisms (e.g., a security deposit that a cheating client forfeits).

Looking ahead, there are several practical directions to explore as presented in the above section. Moreover, continued improvements in FHE libraries, particularly in terms of bootstrapping efficiency might directly benefit our protocols.

9 Acknowledgments

This work was carried out at LASEC, the Security and Cryptography Laboratory at EPFL. I would like to thank my supervisor, professor Serge Vaudenay, for helping me in this project.

References

[1] Ertem Nusret Tas, István András Seres, Yinuo Zhang, Márk Melczer, Mahimna Kelkar, Joseph Bonneau, and Valeria Nikolaenko. Atomic and fair data exchange via blockchain. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3227–3241, 2024.

[2] Henning Pagnia, Felix C Gärtner, et al. On the impossibility of fair exchange without a trusted third party. Technical report, Citeseer, 1999.

[3] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A Reuter, and Martin Strand. A guide to fully homomorphic encryption. *Cryptology ePrint Archive*, 2015.

[4] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Pasta: A case for hybrid homomorphic encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):30–73, 2023.

[5] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 394–403. IEEE, 1997.

[6] zama ai. tfhe-rs. `https://github.com/zama-ai/tfhe-rs`, 2025.

[7] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak reference, version 3.0. *NIST SHA3 Submission Document (January 2011)*, 9, 2011.

[9] Christophe De Cannière and Bart Preneel. *Trivium*, pages 244–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] Thibault Balenbois, Jean-Baptiste Orfila, and Nigel Smart. Trivial transciphering with trivium and tfhe. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '23, page 69–78, New York, NY, USA, 2023. Association for Computing Machinery.

[11] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for {Zero-Knowledge} proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 519–535, 2021.