Green Tracking: decentralized and private electricity tokens matching

Antonella Del Pozzo[1*], Sara Tucci[1*] and Eloi Besnard[1,2*]

[1*]Tutors: LICIA, CEA, Saclay, France.
[2]Intern: ASR, Telecom SudParis, Evry, France.

*Corresponding author(s). E-mail(s): antonella.delpozzo@cea.fr; sara.tucci@cea.fr; eloiposeidon@gmail.com;

Abstract

Physical markets such as electrical energy markets make the foundations of modern world societies. Yet, electrical energy is hardly traceable. Therefore the common approach is to virtually exchange ownership of a quantity of energy that was already physically exchanged. Blockchains, by their decentralized nature, can provide automation and traceability in the purpose of this exchange. Taking advantage of the benefits of decentralized systems is a key element in this work. Nonetheless, matching offer and demand of energy on public networks can also jeopardize privacy of the data. Moreover, the challenging environment in which this work is embedded also contributes to foster privacy between actors. Using several layers of encryption can prevent ill-intentioned actors to gain access to unauthorized data. Hybrid symmetric and asymmetric encryption schemes ensure privacy between participants of an exchange market, while homomorphic encryption guarantees computation on ciphered data. We propose a working implementation of this approach and address some limitations of the work.

Keywords: Electricity, Blockchain, Fully Homomorphic Encryption

Contents

1 Introduction

Electricity markets are intricate systems connecting electricity producers to customers. Producers are actors that convert any type of energy into electrical energy. Customers (i.e. companies, private households, ...) consume electricity. Even if the physical flow directly links producers to customers, virtual actors called suppliers can act as intermediaries. The suppliers purchase a part of the produced electricity and sell it back to the customers, while the rest is directly sold by the producers to the customers. Once that the customers has subscribed to an electricity provider (either to a supplier or to a producer), customers can express preferences over the type of electrical energy they will buy. In other words, they can ask to receive electricity coming from renewable sources rather than from non-renewable sources.
Yet, it is straight-forward to understand that electrons can not be tracked over the grid. Thus, it is impossible to tell where the electricity actually comes from. The usual solutions to tackle this issue is to represent the quantity of energy produced by the producers and the quantity of energy desired by the customers. These quantities can be expressed through certificates that are created proportionally to the demand and the production. For instance, Renewable Energy Certificates (RECs) [1] are instruments available in the United-States power grid. They are used to prove the origin of the electricity that customers can buy. In the European Union, similar certificates exist under the name of Guarantee of Origin (G.O.). Yet, these certificates encompass large time duration and large surfaces. Their granularity levels are unsatisfactory. Moreover, the exchange of certificates is done under blurred conditions that lack of trust[2].
Some requirements need to be met for the Green Tracking approach to be consistent. In the **Green Tracking** approach, we want to put forward the use of decentralized distributed systems such as blockchains to improve the trustworthiness and automation of the process. The computation for the exchange of electrical certificates is mostly done in a centralised manner [2] and is prone to errors [3]. Instead of relying on suppliers to make the exchange of the certificates, one would tend to adopt distributed systems to gain trust and traceability[4]. The supplier, which originally operates the computation, wishes to keep the computation private. In other words, the algorithm that exchanges the certificates should not be disclosed on public networks. Moreover, the computation can be quite intensive. For highly-decentralized systems, the costs linked to the computation can rise easily [5]. A need for an outsourced computation with zero-knowledge proofs is highlighted.
Furthermore, the issuance of the certificates can be embedded in a competitive frame between the customers and the privacy of the data can be jeopardized. Encryption schemes that enable computations over ciphertexts (Homomorphic Encryption schemes[6]) are relevant for that regard. Yet, combining computation on ciphertexts and decentralized distributed systems can be cumbersome because of the replication of heavy computations[6] over the network[5]. Nonetheless, multiple solutions exist to centralize the computation while guaranteeing the benefits of decentralized distributed systems.
In [7], they leverage blockchain for tracebility, fully homomorphic encryption for privacy and oracles for integrity to address the same problematic frame.

Our solution, built on top of [7], aims to create a traceable and privacy-preserving matching service to exchange electrical energy certificates in a trustworthy and private manner.

2 Prerequisites

The technical aspects of the project involve both decentralized and cryptographic systems. While blockchains and smart contracts contribute to leverage the decentralisation and automation part, the different encryptions schemes explained in 2.2 meet the privacy requirements.

2.1 Decentralization aspects

2.1.1 Blockchains

Blockchains are massively distributed ledgers that run on many nodes exchanging transactions (of money, of information,...). Each node holds an copy of the ledger. Updating this copy can be done by anyone submitting a new transaction. Transactions are signed to prevent other people than their intended sender from sending them. Signing transactions is done thanks to algorithms like ECDSA (Elliptic Curves Digital Signature Algorithm)[8]. These transactions are concatenated in blocks (from what the term "blockchain" was coined) by actors that can have different functions. The latter are encouraged to do so with an economic incentive composed of the assets of the chain. The mechanism which governs how participants agree on an unique state is called consensus mechanism. The consistency of the historic of the blocks is maintained with a Merkle Tree. In other words, each block holds within itself a hash of the whole historic of the blocks before itself. Proposing corrupted blocks (with double spendings transactions for example), changing the content or changing the order of a block in the blockchain is possible. Yet, there are mechanisms to thwart ill-intentioned behavior. In the case of Bitcoin [9] with a consensus relying on Proof of Work, resolving hard cryptographic puzzles is needed to submit a new block. Modifying a block would therefore require to re-compute the whole historic of the blocks after the corrupted one. Unless having more than 51% of the computational power of the network, it's eventually unfeasible to create a new valid history. The most trusted chain is the one with the most computational effort put into it. In the case of Ethereum, the consensus mechanism relies on Proof of Stake. Only nodes that have enough staked assets (Ethers) can propose to submit a new block to the blockchain. In other words, submitting a corrupted block that could benefit yourself would eventually make the system and your assets' value tumble. Plus, techniques such as slashing are established to check for corrupted blocks.

2.1.2 Smart Contracts

With Bitcoin [9], the aim was to introduce a decentralized digital money which relies on mathematical tools to be trusted. With Ethereum [10], the introduction of smart contracts leverages the capacity to execute instructions on the blockchain. The code written on a smart contract can contain functions and variables that are scattered

accross each node of the blockchain. Changing the state of smart contract by calling its functions and variables on a particular node will make the change on every node of the blockchain. Therefore, modifying the state is costly and each deployment of smart contract or function call cost some amount of the asset of the chain (called gas). Smart contracts can communicate with offline networks through *events*. It is a log that can contain information and arguments and helps to notify users that were previously listening to the blockchain. Smart contracts enable the anchoring. Anchoring is a way to prove thanks to a digital fingerprint, that external data of the blockchain is correct and authentic at a certain point in time.

2.2 Privacy aspects

Different encryption schemes are involved throughout the process and leverage different privacy. The first to take into account is the fully homomorphic encryption (FHE) scheme. Briefly, it enables computation over ciphered data. The hybrid symmetric and asymmetric schemes derive from the FHE scheme. Indeed, at the moment of writing, no multi-key FHE scheme exists in production state. Therefore, since only one key homomorphically ciphers the data, one user could easily decipher their competitor's data in a competiting context. Leveraging the hybrid symmetric and asymmetric encryption scheme helps to prevent this problem.

2.2.1 Hybrid symmetric and asymmetric encryption schemes

Symmetric and asymmetric encryptions schemes are the easiest ones used in the approach. Even if these schemes are used on top of the FHE scheme, they are described first. Their understanding will facilitate the reading of section 2.2.2. As previously mentioned in 2.2, in a competitive approach, privacy of the data should not only be ensured between the actors and the computational operator. Privacy should also be preserved between the actors themselves. The fully homomorphic encryption scheme enables the privacy between the actors and the computational operator, while the hybrid symmetric and asymmetric scheme ensures privacy between actors. It is to note that neither the asymmetric nor the symmetric encryption schemes enables computation on the ciphered data.

a. Asymmetric encryption: asymmetric encryption (or public key encryption) is used to add another layer of encryption over each actor's data ciphered under FHE. It is efficient in this context since asymmetric encryption does not require any prior exchange of private information between the actors and the computation operator. It is called asymmetric since there is an asymmetry between the keys. One key, the public key pk, is used to cipher the data. Another key, the secret key sk, is used to decipher the data. Let's take an input x, an asymmetric encryption function ENC, an asymmetric decryption function DEC, a public key pk and a private key sk.

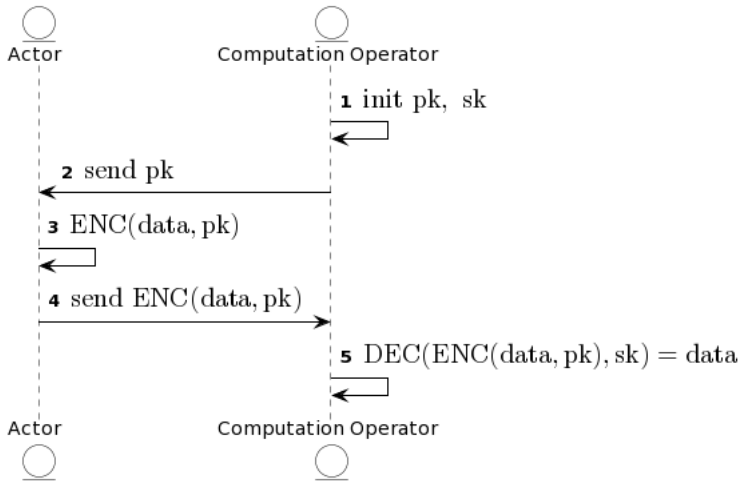

Fig. 1 Asymmetric encryption

ciphering:

$$x \longrightarrow ENC(x, pk) \tag{1}$$

deciphering:

$$ENC(x, pk) \longrightarrow DEC(ENC(x, pk), sk) \longrightarrow x \tag{2}$$

Based on the work of Diffie Hellman[11] in 1976, an asymmetric scheme was created with Rivest–Shamir–Adleman[12] in 1978. Stronger asymmetric encryption schemes have been created notably through Koblitz work[13]. A whole interactive process is described in fig1. Yet, there are some limitations on the size of the data you can cipher with asymmetric encryption.

b. Symmetric encryption: symmetric encryption, on the other hand, can cipher large chunks of data. In symmetric encryption, there is only one key that ciphers and deciphers the data. Let's take an input x, a symmetric encryption function *ENC*, a symmetric decryption function *DEC* and a key *k*.

ciphering:

$$x \longrightarrow ENC(x, k) \tag{3}$$

deciphering:

$$ENC(x, k) \longrightarrow DEC(ENC(x, k), k) \longrightarrow x \tag{4}$$

Solely encrypting data under a symmetric scheme exposes participants to the discovery of the key by ill-intentioned actors. [14] is a symmetric algorithm widely used in the industry now.

c. hybrid symmetric and asymmetric encryption: Combining both symmetric and asymmetric encryptions can help to securely cipher big chunks of data. This work has first been introduced by [15]. It is presented in fig2.

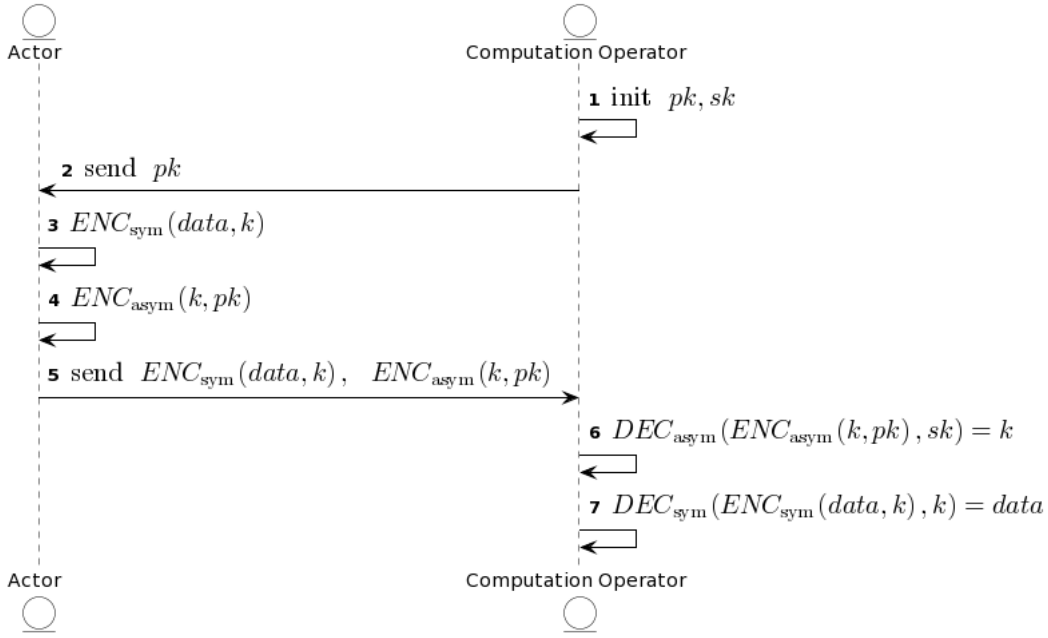

Fig. 2 Hybrid symmetric and asymmetric scheme

2.2.2 Fully homomorphic encryption (FHE) scheme

Homomorphic encryption has first been introduced by Rivest[16], providing a solution to perform limited operations over ciphered data. A scheme allowing to compute any function over ciphered data was proposed by Gentry[17] in 2009. The function they want to evaluate is a set of logic gates (e.g. NAND, OR, XOR, MUX, ...) called a *circuit*. In [17], they define what a *fully homomorphic encryption* scheme is. First, they take a public-key encryption scheme ϵ composed of 3 functions $KeyGeneration_\epsilon$, $Encrypt_\epsilon$, $Decrypt_\epsilon$ and a function $Eval_\epsilon$. They then take a set of n plaintext $\Pi = [\pi_1,...,\pi_n]$ and a set of n ciphertexts $\Psi = [\psi_1,...,\psi_n]$, where $\psi_i = Enc(pk, \pi_i)$. They also consider a security parameter λ.

Such a scheme is said to be *correct* if for any circuit C, any valid pair of keys (pk, sk) output by $KeyGeneration(\lambda)$, and any ciphertexts Ψ:

$$\psi_{Eval} \leftarrow Eval_\epsilon(C, pk, \Psi) \Rightarrow C(\Pi) = Decrypt_\epsilon(\psi_{Eval}, sk) \tag{5}$$

Then, they explained that such a scheme is *homomorphic* if for any circuits C, ϵ is *correct* and $Decrypt_\epsilon$ has a complexity poly(λ). Finally, they put forward that such a scheme is *fully homomorphic* if it is *homomorphic* for any circuit C.
Nonetheless, each gate adds a little bit of noise to the ciphertext. If the noise exceeds a certain threshold, it will be impossible to decipher it.
Two main approaches exist to thwart increasing noise. The first one is to build a circuit of a fixed depth that will keep the noise under a certain threshold. This approach is called a leveled FHE scheme. It was first introduced by BGV[18] in 2011 and later upgraded in 2017 by CKKS[19] scheme.
Another way to do so is a process called bootstrapping. The idea is that one way to reduce the noise of a ciphertext is to decipher it. But deciphering would entail the loss of privacy of the data. Hence, [17] has suggested to homomorphically decipher the ciphertext. I.e. the function we want to evaluate over the ciphertext consists of its own decryption circuit. Yet, no real computation is made on the ciphertext at this stage. The main advancement made by [17] is to ensure that we can evaluate the decryption circuit plus one logic gate. Therefore, the bootstrapping approach leverages a computation over ciphered data for circuit of arbitrary depth.
It can be understood through the following algorithm. Let's consider two pairs of keys (pk_1, sk_1) and (pk_2, sk_2) from a fully homomorphic scheme ϵ. Let's take a decryption circuit $Decrypt_\epsilon$, and a ciphertext ψ_1 that is equal to $Encrypt_\epsilon(pk_1, \pi_1)$, with π_1 a plaintext. We denote the encryption of x under key k as $[x]_k$. The algorithm called *Recrypt* goes as follow:

$$\begin{aligned} &Recrypt_\epsilon(pk_2, Decrypt_\epsilon, sk_1, \psi_1) : \\ &\qquad [\psi_1]_{pk_2} \leftarrow Encrypt_\epsilon(pk_2, \psi_1) \\ &\qquad [sk_1]_{pk_2} \leftarrow Encrypt_\epsilon(pk_2, sk_1) \\ &\qquad \psi_2 \leftarrow Eval_\epsilon(pk_2, Decrypt_\epsilon, \langle [sk_1]_{pk_2}, [\psi_1]_{pk_2} \rangle) \end{aligned} \tag{6}$$

After encrypting under pk_2 the secret key sk_1 and ciphertext ψ_1, they evaluate the decryption circuit $Decrypt_\epsilon$ over the ciphertext ψ_1, along with the secret key sk_1. Therefore, ψ_2 is an encryption of the ciphertext output by $Decipher(sk_1, \psi_1)$ which is the original plaintext π. If the noise level in ψ_2 is inferior to the noise of ψ_1, an advancement is made. For the scheme to be *bootstrappable*, the evaluation of $Decrypt_\epsilon$ in 6 should induce little enough noise so that at least one NAND gate could be evaluated after. If so, then ϵ is said to be *bootstrappable*. The encrypted secret keys are called *bootstrapping keys*.

Schemes that take advantage of this approach are GSW [20] in 2013, FHEW[21] in 2014 and TFHE[6] in 2016. The last one being the scheme used throughout this approach.

The FHE schemes presented here are all single-key schemes. In other words, the original pair of keys (pk_1,sk_1) that is taken to begin the FHE computation can not be changed. Therefore, since every participant that cipher their data need to cipher it with (pk_1,sk_1), they can easily decipher their opponent's ciphertext. Thus, since

actors can be embedded in a competitive frame, leveraging an encryption scheme that protects each actor's privacy while enabling FHE is necessary.

3 Related work

3.1 Cryptographic proofs

Cryptographic proofs enable actors to delegate their computation and still be guaranteed of the integrity of the computation. The origin of interactive proofs can be attributed to [22]. In short, an actor (called the verifier) outsources a computation to another actor (called the prover). Once the computation is done, the prover will try to prove that the computation is correct by submitting a cryptographic proof to the verifier. Multiple types of proofs exist. They can differ from the number of interactions that take place between the verifier and the prover: many interactions[23] or one[24]. They can be zero-knowledge [25] [26] (i.e. the verifier does not learn more than the statement the prover wants to prove) or not. They can differ by size, complexity. They can differ by their public parameters or trusted set-ups, or by being transparent[27]. These proofs are powerful tools that enable the outsourcing of a heavy computation. Nonetheless, provers gain knowledge over the computation they're being sent and this completely threatens the privacy of the computation and the inputs. A solution could consist in combining cryptographic proofs (for computational correctness) and FHE (for computational privacy).

The work done in [28] enables private and verifiable computation. Based on the work of [29] that made advancements in public verifiable computation, [28] added privacy to the manner by replicating the computation on several actors. [30] added privacy to verifiable in their work with differential privacy. Yet, these schemes do not have an actual practical solution to build on top of it. Moreover, they lack of traceability and durability. A centralised sequencer producing proofs could be faulty and induce loss of traceability.

3.2 Trusted Execution Environment (TEE)

Trusted Execution Environment are isolated parts of the processor[31]. Applications inside TEEs can be run without interfering with the rest of the system. The integrity of the data stored inside TEEs is ensured as outside code can not modify it. This is often done by ciphering parts of the memory. TEEs enable private computation on remote hardware, like in cloud-computing [32],[33]. Several brands leverage TEEs such as Intel [34], ARM [35], AMD[36],... as highlighted by [37]. Some projects have taken advantage of blockchains' decentralisation and security combined with TEE computation as in [38]. Yet, they heavily rely on the manufacturer that molds the chips[39]. Moreover, they have been prone to many hacks and attacks as put forward in [40].

3.3 Energy Web Chain

Energy Web Chain [41] is a project leveraging blockchain to alleviate data and workflows related to the electricity market. The aim in this approach is to consider broader

schemes than traditional electricity grids. Anyone possessing electrical generators such as wind turbine or solar panels can plug in the grid and sell their surplus. This is mainly automated by the use of a blockchain called "Energy Web Chain". This blockchain is based on a Proof Of Authority (PoA)[42] consensus. This consensus is not permissionless and is prone to censorship and blacklisting [42], [43].

To sum up, cryptographic proofs promise ground breaking impact in the verifiable and private computation world, yet this trail-blazing technology is not mature enough and still present some flaws. The TEE still fail to be completely independant of their manufacturer and are still prone to some attacks. The Energy web chain, by choosing PoA consensus, omits to propose easily accessible solution.

4 Application and system context

With the technical tools seen in 2 and related work seen in 3, we may try to design a solution for the **Green Tracking** approach with the involved actors and the data.

4.1 Proposed application

In short, a proposed application includes blockchain-based orchestration of the exchange of certificates. The computation needed for the exchange is done with an external actor, called an *oracle*. The oracle needs to respect integrity of computation, privacy of the computation and privacy of the data. As seen in 3, no actual scheme implementing these 3 criterias exist. We choose a design where the oracle lacks of integrity but the traceability at stake is ensured with the blockchain. Indeed, blockchains offer trust (2.1) on which we can rely to anchor our data. Moreover, since the oracle should not learn anything about the data it handles, the use of FHE schemes is relevant. Here is a table to understand what is done under the **Green Tracking** application.

Feature	Fulfilled	Technical aspect
Traceability	yes	blockchain
Integrity	no	centralised oracle
Privacy of algorithm	yes	centralised oracle
Privacy of data between users	yes	asymmetric/symmetric encryption
Privacy of data between users and oracle	yes	homomorphic encryption

Fig. 3 Requirements met in the **Green Tracking** application

The whole approach is described in details in the section 5.

4.2 Actors

Different actors take place in the application.

- The **FHE key generator** generates the FHE keys needed in the process.
- The **user** is either a producer or a customer. They only differ by the type of certificates they have.
- The **smart contracts** form a decentralized orchestrator of the application. Act as an anchor and communicator.
- The **distributed storage (DS)** alleviates on-chain heavy data.
- The **oracle** is the external actor doing the computation on the ciphertexts.
- The **supplier** is there to trigger the matching.

4.3 Data

During the application process, each actor exchange information to other actors:

- A pair of **bootstrapping** and **secret** keys is created by the FHE key generator.
- The users create two types of data. Certificates and a symmetric key.
  1. Certificates of electrical energy created by the users are no longer called certificates but rather tokens because of their digital aspect. In the application, there are two types of tokens, the **Grey Token** (GyT) and the **Green Token** (GnT).
     - The GnT is held by the producers. One GnT represents one kWh of generated electric energy. It is available to trade.
     - The GyT is held by the customers. One GyT represents one kWh of electric energy demand.
  2. The other type of data created by users is a **symmetric keys** k_{sym}.
- The oracle creates a pair of **asymmetric keys** called (pk_{asym}, sk_{asym})

5 The approach

5.1 General approach

The general approach is described with the sequence diagram 4. The whole process starts with the supplier's interaction with the smart contracts through an event called *GenKeys* (3). The FHE Key Generator and the oracle were previously listening to the blockchain (1,2). Once the *GenKeys* event is triggered (4,5), two simultaneous process take place : the oracle creates a pair of asymmetric keys (6) and the FHE key generator creates a pair of FHE keys (9). Once the asymmetric keys are created, the oracle forwards the public key to the smart contracts and triggers the *Asymmetric* event (7). The oracle then waits for the *cipheredToken*(*num_user*) event (8) (The *num_user* is the number of users taking place in the process. Users are assigned id in ascending order. Therefore once the user with id *num_user* will have ciphered and published their token, the oracle will be triggered back). Meanwhile, the FHE key generator has initialised the FHE bootstrapping keys and the FHE secret key (9). The FHE key generator forwards the BSKey on the Distributed Storage (10), which returns a hash that serves as a resource locator (11). The hash is then anchored on the smart contracts, while the event *FHE* is emitted (12). Users were waiting for both the *Asymmetric* and *FHE* events (13). Now that they are both emitted, the users can start the ciphering of

their token (14). They first fetch the public Asym key on the smart contracts (15,16). They then retrieve the FHE bootstrapping key to the FHE Key Generator (17,18). They cipher their token (19) as described in section 5.2. The users then forward their ciphered token on the distributed storage (20). The latter returns a hash, which consists as a resource locator (21). The users finally publish the hash on the smart contracts and trigger a *cipheredToken*(*id*) event (22). Once this event is emitted, the oracle is triggered back (23). The oracle then fetches all users' respective hash and looks for the users' ciphered token on the distributed storage (24,25). The oracle also fetches the FHE bootstrapping key hash on the smart contracts and retrieves the key on the distributed storage. Once the oracle has gathered each user's ciphered token and the FHE bootstrapping key (27), it can proceed to the matching (28). Once it has sorted the tokens, it publishes the ciphered tokens back on the distributed storage (29,30). The respective hashes are published on the smart contracts (31). Users can find their matched ciphered token with their id.

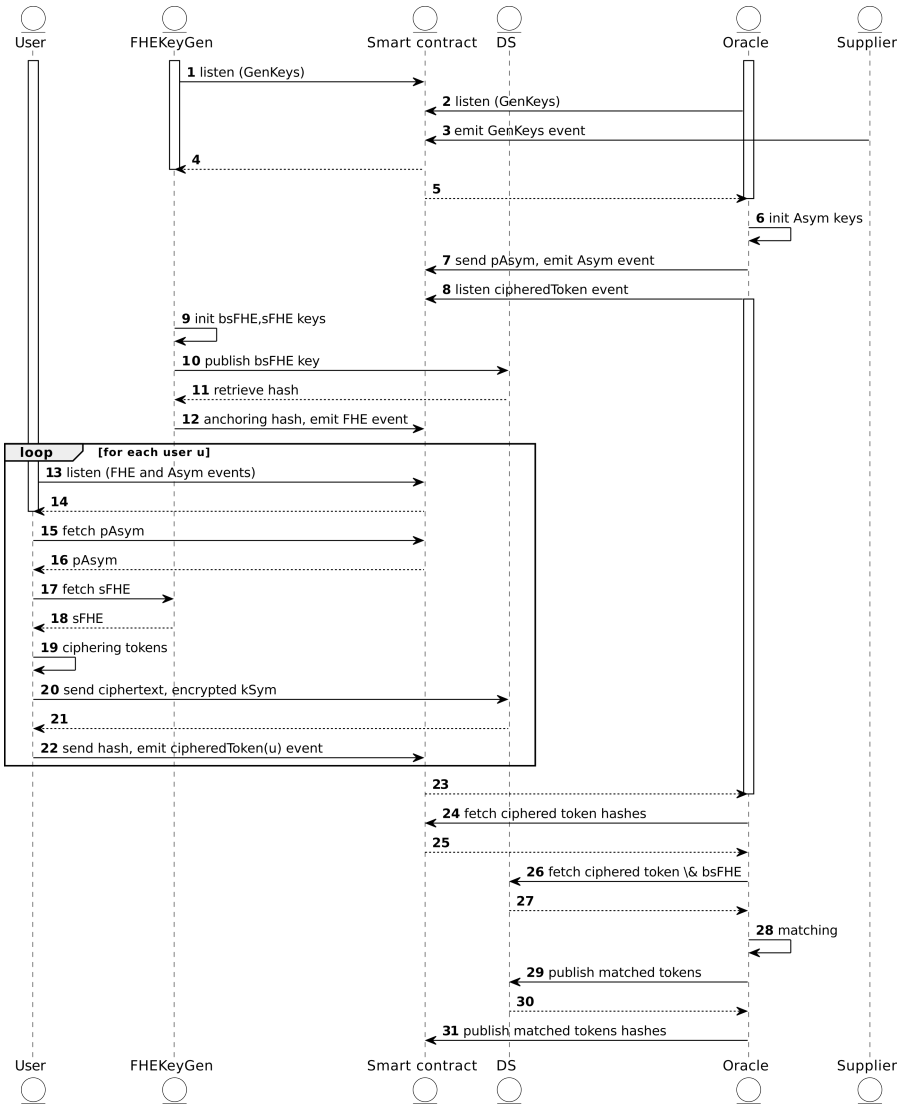

Fig. 4 Sequence diagram

5.2 Encryption of tokens by users

This part deals with the steps 13 to 22 in the diagram 4. No data is ciphered in any encryption schemes before the step 19.

Users:
As mentionned in 4, users are either a producer or a customer. In the approach, users are characterized by 4 features:

- a plaintext boolean: the **type**
- a plaintext int: their **GyT**
- a plaintext int: their **GnT**
- a plaintext int: their **id**

A producer p has type 0.
A customer c has type 1.
For the moment, producers' GyT are set to 0 and customers' GnT too. Until step 19 of diagram 4, all of these values are in plaintext.

Encryption of tokens:
After the reception of the FHE secret key ($sFHE$) from the FHE key generator and the Asym public key ($pAsym$) from the oracle, the users can proceed to cipher their data. They first cipher their GyT and GnT values under FHE using a FHE encryption function ENC_{FHE} as in 7,8:

$$GnT \rightarrow ENC_{FHE}(GnT, sFHE) \rightarrow [GnT]_{sFHE} \tag{7}$$

$$GyT \rightarrow ENC_{FHE}(GyT, sFHE) \rightarrow [GyT]_{sFHE} \tag{8}$$

The oracle will make computation over these ciphertexts. Then, they cipher their FHE offer under the symmetric scheme using a symmetric encryption function ENC_{sym} as in 9,10:

$$[GnT]_{sFHE} \rightarrow ENC_{sym}([GnT]_{sFHE}, kSym) \rightarrow [[GnT]_{sFHE}]_{kSym} \tag{9}$$

$$[GyT]_{sFHE} \rightarrow ENC_{sym}([GyT]_{sFHE}, kSym) \rightarrow [[GyT]_{sFHE}]_{kSym} \tag{10}$$

Then, the $kSym$ is ciphered thanks to the $pAsym$ key from the oracle, using an asymmetric encryption function ENC_{asym} as in 11:

$$kSym \rightarrow ENC_{asym}(kSym, pAsym) \rightarrow [kSym]_{pAsym} \tag{11}$$

The diagram 5 sums up the encryption process on the users' side. The $sAsym$ and $bsFHE$ are showed in this diagram but are not used for the encryption of the tokens.

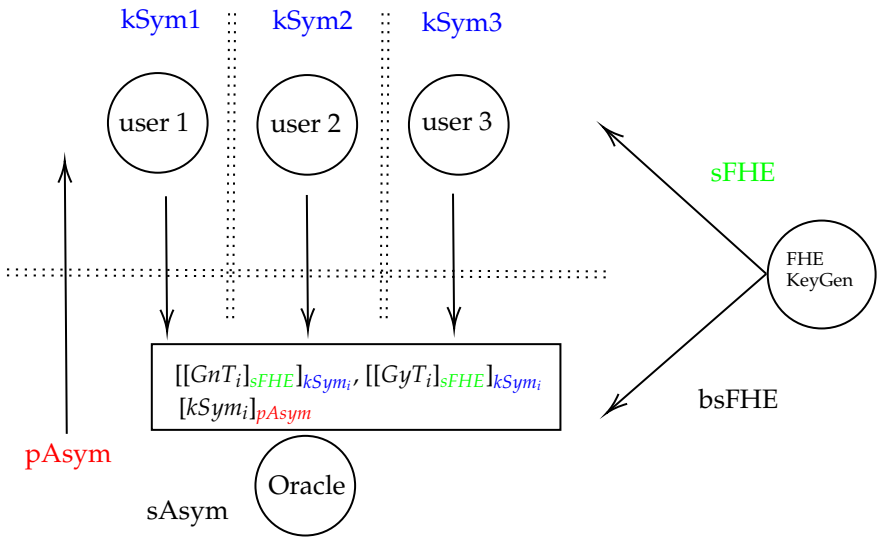

Fig. 5 Encrypted tokens

Sending the data:
Once the $kSym$ and GyT and GnT are correctly ciphered, the users proceed to send them on the blockchain. The GyT and GnT FHE ciphertexts can be quite heavy (refer to table 14). Therefore, they are sent on the distributed storage. In return, hashes that act as resource locator are received by the users. They can then manage to publish theses hashes and the $kSym$ (lighter than the FHE ciphertexts) on the smart contracts.

5.3 Oracle

This part deals with steps 6-7 and 23-27 of the diagram 4.
The oracle has multiple tasks as illustrated in 4. After it has generated asymmetric keys, it waits for the ciphered tokens from the users. Once they are all published, the oracle retrieves the ciphered tokens on the distributed system with the help of the hashes fetched on the smart contracts. It gets in the same time the encrypted symmetric keys. The oracle first needs to decipher the symmetric key for each user i, $i \in [0, N_u]$ (with N_u the number of users) and then can decipher the symmetric encryption layer on the FHE data. For the decyption of the asymmetric scheme, we use the decryption function DEC_{asym}. For the decyption of the symmetric scheme, we use the decryption function DEC_{sym}.
The decryption of each symmetric key goes as follow (12):

$$[kSym_i]_{pAsym} \rightarrow DEC_{asym}([kSym_i]_{pAsym}, sAsym) \rightarrow kSym_i \tag{12}$$

Then, using $kSym_i$, it can decipher the tokens encrypted under symmetric encryption for each user as in 13, 14.

$$[[GyT_i]_{sFHE}]_{kSym_i} \rightarrow DEC_{sym}([[GyT_i]_{sFHE}]_{kSym_i}, kSym_i) \rightarrow [GyT_i]_{sFHE} \quad (13)$$

$$[[GnT_i]_{sFHE}]_{kSym_i} \rightarrow DEC_{sym}([[GnT_i]_{sFHE}]_{kSym_i}, kSym_i) \rightarrow [GnT_i]_{sFHE} \quad (14)$$

The oracle can then proceed to make computation over the ciphertexts, using the FHE Boostrapping key.

$$matching([[GnT_i]_{sFHE}, [GyT_i]_{sFHE}]_i, bsFHE) \quad (15)$$

The $[[GnT_i]_{sFHE}, [GyT_i]_{sFHE}]_i$ is a vector composed of all the ciphered Grey and Green tokens of the users. The matching algorithm will be presented in more details in 5.4.

5.4 Matching algorithm

The matching algorithm is a sequence of operations on FHE ciphertexts of the GyT and the GnT of each user. It is done in the step 28 of the diagram 4. This algorithm aims to match customers' GyT with producers' GnT according to some rules and in a private manner.
If a user is *matched*, it means that their GnT amount will be their GyT amount and their GyT amount will be 0 at the end of the algorithm.

5.4.1 Parameters

The matching engine is called once all data from the users is gathered. We denote the number of customers as N_c, the number of producers as N_p, the number of users as N_u, the number of bits in ciphertexts as N_b. These numbers are in plaintext.
The *cipher_texts* array that comprises each user's FHE-ciphered data is of the following format:

$$[0, [GyT_0]_{sFHE}, [GnT_0]_{sFHE}, ..., i, [GyT_i]_{sFHE}, [GnT_i]_{sFHE}, ..., N_{u-1}, [GyT_{N_{u-1}}]_{sFHE}, [GnT_{N_{u-1}}]_{sFHE}] \quad (16)$$

The oracle can easily reach each user's *id*, $[GyT]_{sFHE}$ or $[GyT]_{sFHE}$ as it knows N_c and N_p.
The **parameters** are the following:

1. an array of empty ciphertexts *result* that will be filled with the result of the matching. Its length is 3*N_u.
2. the array of the users' ciphertexts as described in 16, called *ciphertexts*.
3. the number of customers N_c.
4. the number of producers N_p.
5. the number of bits N_b.
6. the bootstrapping key *bsFHE*.

5.4.2 Rules

The rules of the matching are the following:

1. match **biggest** GyT demand first. The sorting is done using bubble sort. This algorithm has not the best average complexity (n^2, if n is the length of the array to sort) but it is the only one found in the literature [44].
2. match **as many** customers as possible, even if bigger (i.e. bigger GyT demand) customer can not be matched

5.4.3 Prerequisites

Here are some prerequistes needed to better understand the matching algorithm.

- **IF conditions:** Since no value can be read, computing with FHE ciphertexts prevents the use of if condition. The way to deal with if is to use MUX gates as in most of the functions here. MUX gates take three bits as inputs a,b and c and homomorphically computes the following: $(1-a)*b+a*c$. Therefore, it is feasible to do an if condition. Using MUX gates, either we recopy bit by bit the original value of the variable (i.e. it acts like we did not meet the if condition) or we recopy another value into the original variable (i.e. we entered the if condition).

- **Bootstrapping keys:** It is to note that each function in **bold font** is computing over FHE ciphertexts. It therefore needs the $bsFHE$. Yet, for the sake of simplicity, we remove the bootstrapping key in each function that computes over FHE ciphertexts. I.E. instead of writing $function(args, bsFHE)$ we note it as **function**$(args)$.

5.4.4 Pseudo-code

Here is the pseudo-code of the matching algorithm. As said earlier, the function in bold font are dealing with FHE ciphertexts. In other words, only the variable in the for loops are plaintexts (γ, Γ, β and N_c, N_p, N_b). The other variables are FHE ciphertexts.

```
1  matching(result, cipher_texts as c_t, N_c, N_p, N_b):
2      CONSTANT(zero,0)
3      for π in range(N_p):
4          sum_GnT = add(c_t[π].GnT, sum_GnT)
5      for γ in range(N_c):
6          for β in range(N_b):
7              COPY(GyT_array[γ] + β, c_t[γ].GyT + β)
8      bubble_sort(GyT_array)
9      for Γ in range(N_c):
10         for γ in range(N_c):
11             is_max = compare(c_t[γ].GyT, GyT_array[Γ])
12             is_not_zero = compare(c_t[γ].GyT, zero)
13             is_enough = sup_or_eq(c_t[γ].GyT, sum_GnT)
14             is_max_and_not_zero = AND(is_max, is_not_zero)
15             to_match = AND(is_enough, is_max_and_not_zero)
16             for β in range(N_b):
17                 c_t[γ].GnT + β = MUX(to_match, GyT_array[Γ] + β, c_t[γ].GnT + β)
18                 c_t[γ].GyT + β = MUX(to_match, zero, c_t[γ].GyT + β)
19                 GnT_to_rmv + β = MUX(to_match, c_t[γ].GnT + β, zero)
20             sum_GnT = subtract(sum_GnT, GnT_to_rmv)
21     for π in range(N_p):
22         for β in range(N_b):
23             COPY(result[π].GyT + β, zero)
24             COPY(result[π].GnT + β, sum_GnT + β)
25     for γ in range(N_c):
26         for β in range(N_b):
27             COPY(result[γ].GyT + β, c_t[γ].GyT + β)
28             COPY(result[γ].GnT + β, c_t[γ].GnT + β)
```

Fig. 6 matching algorithm

5.4.5 Toy example

For a better understanding of the matching algorithm, let's take simple values and run the matching algorithm.

In 17, we call the matching algorithm with simple parameters. We take 3 customers (customer A, B and C) which respectively have 10 GyT, 25 GyT and 8 GyT. We take one producer with 34 GnT. All the values are encrypted under 8 bits.

$$matching([], [0, [10]_{FHE}, [0]_{FHE}, 1, [25]_{FHE}, [0]_{FHE}, 2, [8]_{FHE}, [0]_{FHE}, 3, [0]_{FHE}, [34]_{FHE}], 3, 1, 8) \tag{17}$$

In 18, we collect the GnT from the producers in *sum_GnT* (line 4 of the algorithm):

$$sum\_GnT = [34]_{FHE} \tag{18}$$

In 19, we collect the GyT from the producers in GyT_array (line 7 of the algorithm):

$$GyT\_array = [[10]_{FHE}, [25]_{FHE}, [8]_{FHE}] \tag{19}$$

In 20, we sort GyT_array (line 8 of the algorithm):

$$GyT\_array = [[25]_{FHE}, [10]_{FHE}, [8]_{FHE}] \tag{20}$$

We will explain the rest of the algorithm with a table 7:

	$\Gamma=0$ $\gamma=0$	$\Gamma=0$ $\gamma=1$	$\Gamma=0$ $\gamma=2$	$\Gamma=1$ $\gamma=0$	$\Gamma=1$ $\gamma=1$	$\Gamma=1$ $\gamma=2$	$\Gamma=2$ $\gamma=0$	$\Gamma=2$ $\gamma=1$	$\Gamma=2$ $\gamma=2$
sum_GnT	34	34	9	9	9	9	9	9	9
$GyT\_array[\Gamma]$	25	25	25	10	10	10	8	8	8
$c\_t[\gamma].GyT$	10	25	8	10	0	8	10	0	8
is_max	0	1	0	1	0	0	0	0	1
is_not_zero	1	1	1	1	0	1	1	0	1
is_enough	1	1	1	0	1	1	0	1	1
$is\_max\_and\_not\_zero$	0	1	0	1	0	0	0	0	1
to_match	0	1	0	0	0	0	0	0	1
GnT_to_rmv	0	25	0	0	0	0	0	0	8
$c\_t[\gamma].GnT$	0	25	0	0	25	0	0	25	8
$c\_t[\gamma].GyT$	10	0	8	10	0	8	10	0	0

Fig. 7 Table to explain line 9 to line 20 of algorithm 5.4.4

And the final result of the matching is:

$$[0, [10]_{FHE}, [0]_{FHE}, 1, [0]_{FHE}, [25]_{FHE}, 2, [0]_{FHE}, [8]_{FHE}, 3, [0]_{FHE}, [1]_{FHE}]$$

Users B and C have been matched, but not user A.

To sum up, the matching algorithm is the centerpiece of the Green Tracking approach. This is when the actual computation on the FHE ciphertexts is done. At the end of the matching, users can find out whether they have been matched.

6 Implementation

The approach described in 5 is implemented in this codebase: https://freyja.intra.cea.fr/EB274036/greentracking.

Encryption schemes: The FHE scheme used is TFHE [45] and the software development kit can be found at this url: https://tfhe.github.io/tfhe/.

The asymmetric encryption scheme is RSA [12] with implementation of https://github.com/andrewkiluk/RSA-Library/blob/master/rsa.c. The symmetric encryption scheme used is AES [14] with implementation of https://github.com/wolfSSL/wolfssl-examples/blob/master/crypto/aes/aes-file-encrypt.c.

Security parameters: The size of the RSA key is 2048 bytes. The size of the AES key is of 128 bytes. The security parameter for TFHE is $\lambda = 110$.

Distributed systems: The blockchain used is Ethereum (https://ethereum.org/fr/) and we replicated an ethereum node using foundry https://github.com/foundry-rs/foundry. We used IPFS as a distributed storage https://ipfs.tech/. For this approach, no code has been deployed in real networks so far and tested against the stress of such environment. The deployment were done on local nodes simulated by the machine.

Programming languages:
Table 8 highlights the programming languages chosen for the code.

RSA	AES	TFHE	smart contracts	IPFS
c	c	c	solidity & python	python

Fig. 8 Programming languages used

Running the Green Tracking implementation was done on one computer of CPU Intel Core i7-10875H 2.30GHz. The communication with the blockchain was done using python's library *web*3. The communication with IPFS was done through the python *requests* library. Since the FHE schemes used in this approach could only be written in C, the communication between the blockchain and the C programs that used TFHE was done through the interaction of python code. Each actor using TFHE (users, oracle, FHE key-generator) has an associated C code (*users.c*, *oracle.c*, *fhe_key_generator.c*). As said earlier, communicating with any blockchain using C-language is cumbersome (it can be done through the use of sockets, but no actual library can handle this properly). So each C code had its relative python code to alleviate the communication (*users.c* linked to *init_user.py*, *oracle.c* linked to *init_oracle.py*, *fhe_key_generator.c* linked to *init_fhe.py*).

7 Evaluation

The evaluation section focuses on three features of the process. The time taken throughout the whole implementation and the memory size of the different pieces of data. The price to deploy these pieces of data on Ethereum's smart contracts is also put forward.

7.1 Time

We evaluated the time it took for the process to be run. Table 9 highlights the percentage of time taken for each steps of the diagram 4, for different numbers of users and for values encoded on 8 bits.

steps:	1-8	9 - 12	13 - 22	23 - 27	28	29 - 31	total time (m)
10 users	4%	6%	12%	6%	60%	12%	8.3
20 users	1%	2%	3%	2%	89%	3%	22.2
50 users	.2%	.3%	.5%	.3%	98.2%	.5%	131

Fig. 9 Time taken per steps

It is straightforward to understand that the matching algorithm takes the most time. Diagram 7.1 highlights the time (in m) taken to process the matching of n users. Both plots were done using a Intel Core i7-10875H 2.30GHz.

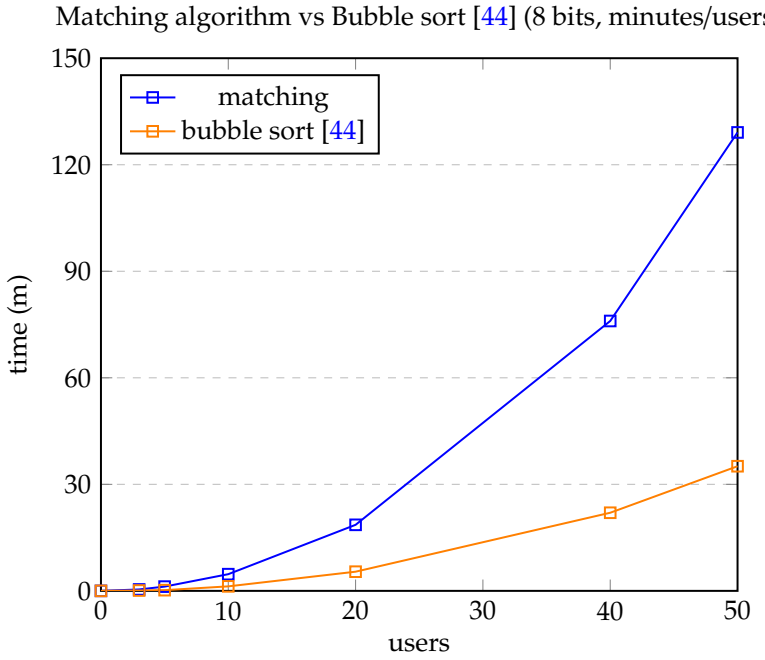

Fig. 10 Matching algorithm vs Bubble sort [44] graph

For 50 inputs, the time taken by the matching algorithm is **131m** and the time taken by the bubble sort is of **34m**.
For 50 inputs, the experimental overhead is of **5641s**.

The overhead between our approach and [44] can be explained by the extra steps taken in the matching algorithm. We calculated the theoretical time of each part of the matching algorithm to compare with the experimental time.
Every function used to make computation with FHE uses FHE logic gates at some point. Table 11 puts forward the time taken for each gate.

FHE Gate	time (s)
AND	0.015
MUX	0.015
OR	0.015
XNOR	0.015
XOR	0.015
NOT	10^{-6}
COPY	10^{-6}
CONST	10^{-6}

Fig. 11 Time taken for each FHE gate

With the table 11, it is possible to compute the theoretical time taken by the different functions in 5.4.4. Figure 12 highlights the different time taken by each function.

functions (5.4.4)	time (s)
add	0.6
compare	0.24
sup_or_eq	0.24
subtract	0.6

Fig. 12 Time taken for each FHE function

Then, it becomes feasible to have a theoretical line-by-line understanding of the time taken to perform the operations, as shown in 13. The lines highlighted are those from the matching algorithm 5.4.4. Let's take a reference of 50 users that encrypted their value under 8 bits.

lines (5.4.4)	time (s)
4	30
7	0.0004
8 (bubble sort)	2000
11	600
12	600
13	600
14	37
15	37
17	300
18	300
19	300
20	1500
23	0.0004
24	0.0004
27	0.0004
28	0.0004
total	6304

Fig. 13 Time taken for each FHE line of 5.4.4

In theory, the overhead is of **4004s**.

The main bottleneck of the Green Tracking approach is the matching algorithm which takes more than half of the total time for 10 inputs, to roughly 99% of the total time for >50 inputs. Since this matching algorithm part deals with FHE programming, there are ways to improve the time taken. The computation was done on one computer, but it can be parallelized for better results. Indeed, the library used for FHE in this implementation [45] explains that it enables multi-threading. Sharing the computation over several machines or using GPUs could be a way to improve the bottleneck of the matching algorithm.

7.2 Storage

The FHE schemes deal with large sizes of data. The sizes of the different objects are described in the table 14. The price associated to deploy such data on the smart contracts of Ethereum is shown too.

Files	size (MB)	price to deploy on SC ($)
FHE secret key	109	$800000
FHE bootstrapping key	109	$800000
FHE ciphertext	0.3	$2500
AES (FHE(ciphertext))	0.3	$2500
RSA keys	10^{-3}	$8
AES keys	10^{-4}	$0.8

Fig. 14 Sizes (MB) and price ($) of data used in the approach

We can see that the relatively big size of the FHE keys (109MB, roughly a video of 1 m in 4k resolution [46]) urge the approach to use distributed storage. We can compute the price to upload such data on Ethereum. Saving 1 KB of data costs 250 000 gas [47]. At the time of writing, one unit of gas is of $\$3.4 * 10^{-5}$, 1 *ETH* being \$1700. Therefore, for 109 MB of data, the cost would be roughly $250000 * 3.4 * 10^{-5} * 100 * 1000 = \$800000$. For 0.3 MB, the price would be of \$2550.

To sum up, the FHE ciphertexts and keys are heavy pieces of data that will take huge amount of storage. Deploying them on distributed storage and anchoring them using hashes (few bytes) is a much less costly way of doing.

8 Threat analysis

This approach relies on several assumptions that can be jeopardized. There exists several scenarios that could lead to a loss of one of the properties of the Green Tracking approach. The properties ensured in the approach are:

- traceability
- privacy of algorithm
- privacy of data between users
- privacy of data between users and oracle

The traceability is hardly challengeable. Since we rely on Ethereum, we can have relative trust to the decentralisation of this blockchain [48]. Anchoring the hashes of the data sent on IPFS helps to support the traceability. Nonetheless, the privacy of the algorithm can be jeopardized in more than a *honest but curious* attitude from the oracle. Since the Green Tracking approach needs to trust the oracle, the latter can run the algorithm on public networks without the supplier's consent. The privacy of the data between users is ensured since the symmetric encryption prevents users from collecting their competitor's data. Unless possessing the symmetric key of their competitors, no user can decipher other's data. Yet, the privacy can be thwarted when it comes to fully homomorphic encryption. The oracle could take advantage of a user or of the FHE-key-generator and retrieve the FHE secret key. Indeed, only the users and the FHE-key-generator should have the FHE secret key that can decrypt FHE ciphertexts. Doing this would enable the oracle to read the content of the FHE-ciphered data it perceives.

9 Conclusion

In this work, we implemented a practical approach of the theoretical frame proposed in [7]. Our approach is focused on matching units of electrical energy called tokens in a private and decentralized manner. It relies on blockchain, symmetric, asymmetric and fully homomorphic encryption schemes to provide traceability and multi layer privacy on the data. The implementation 6 shows that such a scheme can be functional. Although it is operational, several limitations should be taken into account. Considering 8, several drawbacks can still hinder the process in an attitude exceeding the threshold of *honest but curious* from the oracle, a user or the fhe-key-generator.

Exchanging FHE keys would let the FHE encryption collapse and would let the oracle know about every user's data. As the oracle is also centralised, the integrity of the computation can also be challenged.
Since the FHE scheme used in this approach is single-key, then the approach is burdened by the use of the hybrid symmmetric and asymmetric scheme.
On another side, the time it takes to match a relatively small group of users (50 users, $\tilde{2}$ h) can consist of a obstacle. Indeed, if there are temporal guidelines and deadlines from the supplier, they could be difficult to meet. Parallelizing the computations seems to be the best option to reduce this temporal bottleneck. Moreover, the size of the FHE ciphertexts and FHE keys are relatively heavy ($.1 - 100$MB) and can sometimes impede the correct exchange of these heavy pieces of data over distributed storages.
Running experiments on the main network of Ethereum can consist in a milestone for this project, as it would prove to be functional (or not) in real conditions.
There are numerous projects working on private and verifiable computation schemes that would greatly benefit this approach, such as zero-knowledge FHE rollups [49].

References

[1] Renewable Energy Certificates (RECs). https://www.epa.gov/green-power-markets/renewable-energy-certificates-recs#one

[2] Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguía, L.-M., Rothchild, D., So, D., Texier, M., Dean, J.: Carbon emissions and large neural network training (2021)

[3] How Iceland sold the same Green Electricity twice. https://industrydecarbonization.com/news/how-iceland-sold-the-same-green-electricity-twice.html

[4] Agrawal, T.K., Kumar, V., Pal, R., Wang, L., Chen, Y.: Blockchain-based framework for supply chain traceability: A case example of textile and clothing industry. Computers Industrial Engineering **154**, 107130 (2021) https://doi.org/10.1016/j.cie.2021.107130

[5] Sguanci, C., Spatafora, R., Vergani, A.: Layer 2 blockchain scaling: a survey. ArXiv **abs/2107.10881** (2021)

[6] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds. Cryptology ePrint Archive, Paper 2016/870 (2016). https://eprint.iacr.org/2016/870

[7] Henry T, B.E.L.N.G.W. Hatin J: Towards trustworthy and privacy-preserving decentralized auction (2023). https://doi.org/hal-04145599

[8] The Elliptic Curve Digital Signature Algorithm (ECDSA) (2001). https://doi.org/10.1007/s102070100002

[9] Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)

[10] Buterin, V.: Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. (2014)

[11] Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory **22**(6), 644–654 (1976) https://doi.org/10.1109/TIT.1976.1055638

[12] R.L. Rivest, A.S., Adleman, L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems (1978)

[13] Koblitz, N.: Elliptic Curve Cryptosystems (1987)

[14] Dworkin, B.E.N.J.F.J.B.L.R.E. M., Dray, J.: Advanced Encryption Standard (AES) (2001). https://doi.org/10.6028/NIST.FIPS.197(AccessedJune30,2023)

[15] Mahalle, V.S., Shahade, A.K.: Enhancing the data security in cloud by implementing hybrid (rsa & aes) encryption algorithm. 2014 International Conference on Power, Automation and Communication (INPAC), 146–149 (2014)

[16] Rivest, R.L., Adleman, L., Dertouzos, M.L., *et al.*: On data banks and privacy homomorphisms. Foundations of secure computation **4**(11), 169–180 (1978)

[17] Gentry, C.: A Fully Homomorphic Encryption Scheme. Stanford university, ??? (2009)

[18] Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully Homomorphic Encryption without Bootstrapping. Cryptology ePrint Archive, Paper 2011/277 (2011). https://eprint.iacr.org/2011/277

[19] Costache, A., Curtis, B.R., Hales, E., Murphy, S., Ogilvie, T., Player, R.: On the precision loss in approximate homomorphic encryption. Cryptology ePrint Archive, Paper 2022/162 (2022). https://eprint.iacr.org/2022/162

[20] Gentry, C., Sahai, A., Waters, B.: Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. Cryptology ePrint Archive, Paper 2013/340 (2013). https://eprint.iacr.org/2013/340

[21] Ducas, L., Micciancio, D.: FHEW: Bootstrapping Homomorphic Encryption in less than a second. Cryptology ePrint Archive, Paper 2014/816 (2014). https://eprint.iacr.org/2014/816

[22] GOLDWASSER, S., MICALI, S., RACKOFF, C.: The knowledge complexity of interactive proof systems (1989)

[23] Cormode, M.M.T.J. G.: : Practical verified computation with streaming interactive proofs. (2012)

[24] Groth, J.: On the size of pairing-based non-interactive arguments. (2016)

[25] Blum, M., Feldman, P., Micali, S.: Non-interactive zero-knowledge and its applications. (1988)

[26] Goldreich, O.Y. O.: Definitions and properties of zero-knowledge proof systems. J. Cryptology. https://doi.org/10.1007/BF00195207

[27] Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046. https://eprint.iacr.org/2018/046 (2018). https://eprint.iacr.org/2018/046

[28] Schoenmakers, V.M.d.V.N. B.: Trinocchio: Privacy-Preserving Outsourcing by Distributed Verifiable Computation (2016). https://doi.org/10.1007/978-3-319-39555-5_19

[29] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly Practical Verifiable Computation (2013). https://doi.org/10.1109/SP.2013.47

[30] Tsaloli, G., Mitrokotsa, A.: Differential Privacy meets Verifiable Computation: Achieving Strong Privacy and Integrity Guarantees (2019)

[31] Mohamed Sabt, A.B. Mohammed Achemlal: Trusted execution environment: What it is, and what it is not. 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (2015)

[32] Shih, M.-W., Kumar, M., Kim, T., Gavrilovska, A.: S-nfv: Securing nfv states by using sgx. (2016)

[33] K. Krawiecka, A.P.M.M. A. Kurnikov, Asokan, N.: Protecting web passwords from rogue servers using trusted execution environments. (2017)

[34] Intel: Intel Software Guard Extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/332680-001-720907.pdf

[35] arm: TrustZone for Armv8-A. https://documentationservice.arm.com/static/6021

[36] Kaplan, D.: AMD X86 Memory Encryption Technologies. USENIX Association, Austin, TX (2016)

[37] Rabimba, K., Xu, L., Chen, L., Zhang, F., Gao, Z., Shi, W.: Lessons learned from blockchain applications of trusted execution environments and implications for future research. In: Workshop on Hardware and Architectural Support for Security and Privacy. ACM, ??? (2021). https://doi.org/10.1145/3505253.3505259 . https://doi.org/10.1145%2F3505253.3505259

[38] Xie, H., Zheng, J., He, T., Wei, S., Hu, C.: Tebds: A trusted execution environment-and-blockchain-supported iot data sharing system. Future Generation Computer Systems **140**, 321–330 (2023) https://doi.org/10.1016/j.future.2022.10.016

[39] Tim Geppert, D.S. Stefan Deml, Ebert1, N.: Trusted execution environments: Applications and organizational challenges. (2022). https://doi.org/10.3389/fcomp.2022.930741

[40] Antonio Muñoz, R.R.J.L. Ruben Ríos: A survey on the (in)security of trusted execution environments. (2023)

[41] The energy web chain: Accelerating the energy transition with an open-source, decentralized blockchain platform. energy web. (2019)

[42] Joshi, S.: Feasibility of Proof of Authority as a Consensus Protocol Model (2021)

[43] Wang, Q., Li, R., Wang, Q., Chen, S., Xiang, Y.: Exploring Unfairness on Proof of Authority: Order Manipulation Attacks and Remedies (2022)

[44] Wang, C., Chen, J., Zhang, X., Cheng, H.: An efficient fully homomorphic encryption sorting algorithm using addition over tfhe. In: 2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS), pp. 226–233 (2023). https://doi.org/10.1109/ICPADS56603.2022.00037

[45] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: Fast Fully Homomorphic Encryption Library. https://tfhe.github.io/tfhe/ (August 2016)

[46] https://www.circlehd.com/blog/how-to-calculate-video-file-size

[47] Exploring Ethereum's Data Stores: A Cost and Performance Comparison. https://arxiv.org/pdf/2105.10520.pdf

[48] Measuring Decentralization in Bitcoin and Ethereum using Multiple Metrics and Granularities. https://arxiv.org/pdf/2101.10699.pdf

[49] https://www.zama.ai/post/private-smart-contracts-using-homomorphic-encryption