Fidelis: Verifiable Keyword Search with No Trust Assumption

Laltu Sardar[1][a] and Subhra Mazumdar[2][b]

[1] *Institute for Advancing Intelligence, TCG-CREST, Kolkata, India*

[2] *TU Wien and Christian Doppler Laboratory Blockchain Technologies for the Internet of Things, Vienna, Austria*

Keywords: Searchable Encryption, Keyword Search, Fair Payment, Smart Contract, No-Trust Assumption.

Abstract: A searchable encryption (SE) scheme allows a client to outsource its data to a cloud service provider (CSP) without the fear of leaking sensitive information. The latter can search over the outsourced data based on the client's query. Such a scheme prevents a malicious CSP from sending incorrect results. However, a malicious client can deny receipt of the correct result and wrongly blame the CSP. Existing SE schemes fail when the client acts maliciously.

In this paper, we have studied searchable encryption schemes where none of the parties trust each other. We propose Fidelis, a novel blockchain-based SE scheme, with keyword-search functionality, that is verifiable by both parties. None of the parties can cheat, and an honest CSP gets payment upon providing the result. We implement and evaluate an instance of the protocol on real-life data using Ethereum as the blockchain platform, deploying it in the Ropsten test network. Upon comparing with existing schemes, we observe that our protocol is efficient and scalable.

1 INTRODUCTION

Outsourcing storage, as well as computation to a cloud service provider (CSP), has acquired significant popularity among *Small and Medium-sized Enterprises* (SMEs). Outsourcing increases security effectiveness and reduces maintenance costs. However, directly uploading the data in plaintext form to *cloud service providers (CSP)* is subject to risk. An unauthorized person can misuse or steal the data stored in the cloud. To counter the issue, a different encryption technique called *searchable encryption (SE)* is used. SE allows a user to outsource its data in an encrypted form to a cloud service provider. The cloud can perform any number of queries over that encrypted data at the request of the client. In this process, the client reveals some controlled amount of information about the data to the CSP. Thus, privacy of such schemes is determined by the amount of leakage upon querying over the encrypted database. A malicious server may not return the correct result, but the users should have the ability to check whether the result is complete and derived from the actual state of the database. This is possible only when the client can verify an SE scheme. A client can be malicious and try to cheat CSP. If the client is the only one involved in verification, it can intentionally claim that it got an incorrect result to avoid paying the CSP. So, when none of the parties are trusted, we need an SE scheme that is verifiable by both parties. Blockchain provides service integrity for storage as well as computation. In addition, by using Ethereum-based smart contracts (Buterin, 2020), all operations can be executed automatically and trustfully. It efficiently makes data sharing convenient (Jiang et al., 2019).

However, it is a challenging task to outsource data in an untrusted distributed environment without leaking a significant amount of information about the dataset and queries. In most of the previous works (Fan et al., 2020),(Wang et al., 2018),(Sardar and Ruj, 2019), the client is assumed to be honest. Some recent work, such as Jiang *et al.* (Jiang et al., 2019) consider a fully-malicious setup where the client can behave maliciously from the beginning of the protocol. They used cloud storage to store encrypted files and smart contracts to store encrypted indexes for a single keyword search. However, the actual result of a search is not returned. Secondly, the blockchain is simply used as verifiable storage, incurring a high cost. Guo *et al.* (Guo et al., 2020) used the smart contract to store the verification tag. Thus, in both cases, most of the computation is done either by the client (Jiang

[a] https://orcid.org/0000-0002-7433-0497

[b] https://orcid.org/0000-0002-3089-2535

Sardar, L. and Mazumdar, S.
Fidelis: Verifiable Keyword Search with No Trust Assumption.
DOI: 10.5220/0012082700003555
In *Proceedings of the 20th International Conference on Security and Cryptography (SECRYPT 2023)*, pages 698-703
ISBN: 978-989-758-666-8; ISSN: 2184-7711
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

et al., 2019) or by the server (Guo et al., 2020), and the blockchain is used for storage purposes.

In another work (Guo et al., 2022), the authors have proposed a solution where both the cloud server and the client are malicious, but their schemes consider the client to be quasi-malicious. The assumption is used for all existing schemes, including (Li et al., 2020), (Guo et al., 2020), (Miao et al., 2022) etc. These schemes fail when the client uploads incorrect encrypted data and/or manipulated encrypted search index. If the pre-computed search results and verification information are incorrect, there is no way for the CSP to check. Such a client can cheat and deny payment to the cloud, even if the latter has performed the search correctly. Our aim is to suggest a scheme that addresses all these shortcomings.

Our Contribution: We propose a novel single keyword search scheme called Fidelis that aims to provide verifiability without requiring any trust assumptions. To achieve this goal, we utilize the Ethereum blockchain platform to enable fair interactions between the cloud and the client. Unlike other schemes, we do not store any data on the blockchain except for certain constant-size information that is crucial for valid search and result verification. Our protocol ensures fairness and confidentiality even in the presence of fully malicious participants. Additionally, we develop prototypes and test them on the Ropsten test network using real-world data. Our experimental analysis indicates that our proposed scheme is highly efficient and feasible to implement, making it a promising solution for practical use cases.

Organization: Section 2 presents necessary background knowledge. Section 3 provides detailed description of Fidelis. In Section 4, we evaluate the efficiency of Fidelis. We conclude the paper in Section 5.

2 PRELIMINARIES

System and Adversarial Model: There are three entities - owner, server and user.

The owner of the database *Owner* (O) is fully-malicious and may follow two strategies: (a) correctly uploading data and search index but falsely blaming the cloud to obtain search results without payment, or (b) manipulating the data initially to obtain search results without payment.

Server ($\mathcal{S}$) stores encrypted data and search index, and provides storage and computation services. It may intentionally provide incomplete or incorrect results to reduce its costs, and is considered fully malicious.

User ($\mathcal{U}$) is a client who uses the database. The owner can also be one of the users. It is fully-malicious as owner. We assume, the owner and the users belong to the same organization, a user trusts the owner, and the owner will never collude with the server to cheat a user.

Apart from these, *Blockchain* ($\mathbb{B}$) used here is a public blockchain (e.g. Ethereum) that helps to verify the execution of the search protocol. The client may act as an owner of the database as well as the user.

TSet: A tuple set `TSet` (Cash et al., 2013) is a data structure consisting of a tuple of algorithms (`TSetSetup`, `TSetGetTag`, `TSetRetrieve`) briefly described as follow.

• `TSetSetup` takes as input a security parameter λ and an array $\mathbf{T}$ of lists of equal-length bit strings indexed by the elements of $\mathcal{W}$. It outputs a pair (`TSet`, K_T) where `TSet` is a tuple set data structure and K_T is a key. Thus `TSetSetup` initiates the tuple set data structure `TSet` and chooses a key K_T at random.

• `TSetGetTag` takes as input the key K_T and a keyword w and outputs a search trapdoor $stag_w \in \{0,1\}^\lambda$.

• `TSetRetrieve` takes the `TSet` and a search trapdoor $stag_w$ as input and returns a list of strings.

We say that a `TSet` is correct if for all $\mathcal{W}$, $\mathbf{T}$, and any $w \in \mathcal{W}$, $\texttt{TSetRetrieve}(\texttt{TSet}, stag_w) = \mathbf{T}[w]$ when $(\texttt{TSet}, K_T) \leftarrow \texttt{TSetSetup}(\mathbf{T})$ and $stag_w \leftarrow \texttt{TSetGetTag}(K_T, w)$.

Interval based Sorted Merkle Tree (`IbSMT`): Since we only require non-membership proof, we take interval-based sorted Merkle tree (`IbSMT`). `IbSMT` is the same as SMT (as Sorted Merkle Tree) (Dahlberg et al., 2016) except for the leaves. For example, given a sorted set $S = \{x_1, x_2, \ldots, x_n\}$, SMT keeps x_is in the leaf node. However, `IbSMT` uses elements of $\bar{S} = \{y_0, y_1, \ldots, y_n\}$ as its leaf node, where $y_i = x_i || x_{i+1}$. The two additional elements, x_0 and x_{n+1}, are the minimum and maximum bound and $\forall i \in [1,n], x_i \in (x_0, x_{n+1})$. x_0 and x_{n+1} can be taken as bit-strings of all 0 and 1 respectively. An `IbSMT` data structure consists of three algorithms - *BuildTree, NMSearch* and *NMVerify*:

• $\texttt{IbSMT} \leftarrow BuildTree(S)$: It takes a set S and outputs the sorted Merkle tree `IbSMT` where the leaves are the intervals constructed as above, i,e., elements from $\bar{S}$.

• $pf_n \leftarrow NMSearch(\texttt{IbSMT}, x)$: Given an element x, the algorithm finds two x_i and x_{i+1} such that $x_i < x < x_{i+1}$ and outputs the membership proof pf_n for the interval $y_i \leftarrow x_i || x_{i+1}$ in $\bar{S}$. pf_n is the non-membership proof of x in S.

• $b \leftarrow NMVerify(\texttt{IbSMT}.root, pf_n, x)$: It outputs membership verification bit b result for x and the

proof pf_n. The verification is done with the help of the root of IbSMT.

Definitions and Terminologies: We define a verifiable keyword search scheme VKS as a tuple of algorithms (KeyGen, Build, SrchTknGen, Search, VerifySrch) briefly described as follows.

• $K \leftarrow \texttt{KeyGen}(1^\lambda)$: is a Probabilistic Polynomial Time (PPT) algorithm run by the $\mathcal{O}$ that takes a security parameter 1^λ and outputs secret key K.

• $(EDB, \xi, \alpha) \leftarrow \texttt{Build}(\mathcal{DB}, K)$: $\mathcal{O}$ runs this PPT algorithm that takes the dataset $\mathcal{DB}$ and the secret key K as input and outputs an encrypted database EDB, an encrypted index ξ, and auxiliary data α. α consists of two parts α_o and α_s that are stored by the owner and the server, respectively.

• $\tau_w \leftarrow \texttt{SrchTkn}(w, K)$: $\mathcal{O}$ runs this PPT algorithm that generates an encrypted search trapdoor τ_w for a keyword w with the help of K.

• $(R_w, pf_w) \leftarrow \texttt{Search}(\xi, \alpha_s, \tau_w)$: with this PPT algorithm, $\mathcal{S}$ searches over ξ for τ_w and returns the search result R_w to the client together with a proof pf_w.

• $b_w \leftarrow \texttt{VerifySrch}(R_w, pf_w, \alpha_o)$: Given the result R_w, proof pf_w and auxiliary data α_o, $\mathcal{U}$ runs this PPT algorithm, interacting with cloud, and outputs the verification bit b_w.

Correctness: Given a security parameter $\lambda \in \mathbb{N}$, a verifiable keyword search scheme is said to be *correct*, if for any key K generated using $\texttt{KeyGen}(1^\lambda)$ and for all sequences of search operations, the algorithm Search outputs the correct set of identifiers, except with a negligible probability, and for each received correct result, the result will be verified correctly by the algorithm VerifySrch.

3 OUR PROPOSED SCHEME

The steps of the protocol Fidelis are encoded in the smart contract. *Owner* shares the encrypted database with the *cloud server* and sends the keyword space to the *user*. The latter needs to pay the *cloud server* for retrieval of data against a query. This is enabled by locking coins, denominated in the native currency of the blockchain, in the smart contract. *User* sends a query to the *server* and the latter returns the result. If the *server* has performed his actions correctly, it gets the coins, else the coins are refunded to the *user*. The outline of the protocol is provided in Fig. 1.

Initialization: A tuple of keys, $K = (K_s, \bar{K}_s, K_T)$, is generated by the *owner*. K_s is a λ-bit string, taken at random, used to encrypt the keywords that a *user* intends to query. $\bar{K}_s$ is a λ-bit string, taken at random, used to compute verification tags corresponding to each queried keyword. A pseudo-random function (PRF) $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \rightarrow \{0,1\}^\lambda$ is used for encryption using keys K_s and $\bar{K}_s$. Different hash func-

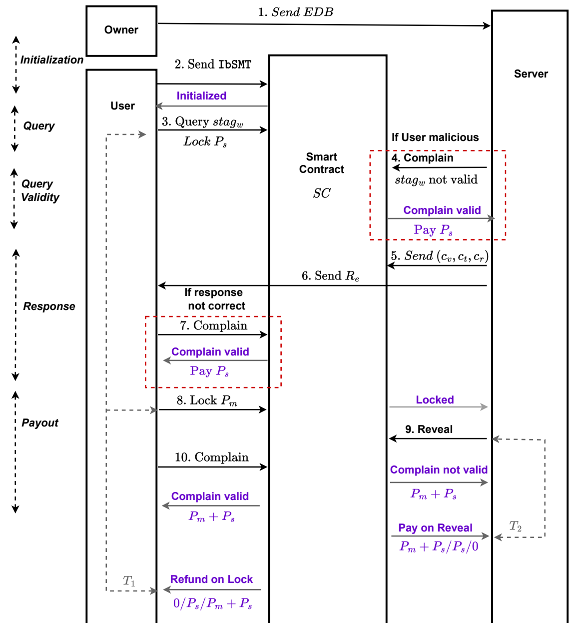


Figure 1: Outline of the protocol.

Algorithm 1: Fidelis.$\texttt{Init}(\texttt{DB})$.

$K_s, \bar{K}_s \xleftarrow{\$} \{0,1\}^\lambda$ for PRF F
Parse $\mathcal{DB}$ as $\{id_1, \ldots, id_d\}$
Find $\{W_1, \ldots, W_d\}$
$\mathcal{W} \leftarrow \cup_{i=1}^{d} W_i$; $StgSET = \phi$; $\mathbf{T} = \phi$
for $w \in \mathcal{W}$ **do**
  Initialize **t** to be an empty list
  Set $K_w \leftarrow F(K_s, w)$; $vtag_w \leftarrow H_1(\bar{K}_S, w)$
  for $i = 1$ **to** $n_w = |\texttt{DB}(w)|$ **do**
    $e_i \leftarrow \texttt{Sym.Enc}(K_w, id_i^w)$
    append e_i to **t**.
  end
  $h_w \leftarrow H_2(vtag_w || e_1 || e_2 || \ldots || e_{n_w})$
  append h_w to **t**.
  Set $\mathbf{T}[w] \leftarrow \mathbf{t}$.
end
$(\texttt{TSet}, K_T) \leftarrow \texttt{TSetSetup}(\mathbf{T})$.
for *each* $w \in \mathcal{W}$ **do**
  $stag_w \leftarrow \texttt{TSetGetTag}(K_T, w)$
  $StgSET \leftarrow StgSET \cup \{stag_w\}$
end
$\texttt{IbSMT} = BuildTree(StgSET)$
$\mathcal{O}$ keeps $K = (K_s, \bar{K}_s, K_T)$
$\mathcal{O}$ sends $\texttt{EDB}, \xi = \texttt{TSet}, \texttt{IbSMT}$ to $\mathcal{S}$.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

tions H_1, H_2, and H_3 are used for generating commitments. Each of them is a cryptographic one-way hash function $\{0,1\}^* \rightarrow \{0,1\}^\lambda$. The *owner* uses a symmetric encryption `Sym` to encrypt each file in $\mathcal{DB}$ and generates EDB. We assume every encrypted file f_i can be accessed with the identifier id_i, and the *owner* gives this encryption key to the *users*.

We use the `TSet` (see Section 2) data structure to construct encrypted index for the database $\mathcal{DB} = \{id_i : i = 1, \ldots, d\}$. Let $\mathcal{W} = \cup_{i=1}^{d} W_i$ be the set of keywords present in $\mathcal{DB}$ where W_i is the set of keywords in the file with identifier id_i. For each keyword $w \in \mathcal{W}$, a list $\mathbf{t} = [e_1, e_2, \ldots, e_{n_w}, h_w]$ is generated by the owner. We define $DB(w) \subset \mathcal{DB}$ as the set of file identifiers that contain the keyword w and n_w is the cardinality of $DB(w)$. $e_i \leftarrow \texttt{Sym.Enc}(K_w, id_i^w), 1 \leq i \leq n_w$ is the encrypted identifiers of the files containing w and $h_w \leftarrow H(vtag_w || e_1 || e_2 || \ldots || e_{n_w})$ is the hash value that binds the set of identifiers and $vtag_w$. Here $vtag_w = H_1(\bar{K}_S, w)$ is the verification tag used to verify the correctness of search results. An array $\mathbf{T}$ of length $\mathcal{W}$ is used to store the list $\mathbf{t}$ for each keyword, i.e., $\mathbf{T}[w] = \mathbf{t}$. Finally, `TSet` for the array $\mathbf{T}$ is generated as $(\texttt{TSet}, K_T) \leftarrow \texttt{TSetSetup}(\mathbf{T})$.

We encrypt each keyword $w \in \mathcal{W}$ using key K_T, generate $stag_w$, and store it in $StgSET$. The encryption is done by `TSetSetup`. An `IbSMT` is built for $StgSET$. The key $K = (K_s, \bar{K}_s, K_T)$ and root of `IbSMT`, defined $\alpha_o = \texttt{IbSMT}.root$, are shared with the *user*. The encrypted search index $\xi = \texttt{TSet}$ is outsourced to the cloud along with $\alpha_s = \texttt{IbSMT}$. Algo. 2 describes the initialization which is generating the keys and building the encrypted database.

Query: To query a keyword w, *user* computes search token $\tau_w = stag_w$ and sends it to the *server*. It initializes the smart contract SC_w with the root of `IbSMT` (see Algo. 3). The user calls *StoreandLock* in SC_w, where it locks P_s coins for time T_1 and stores $stag_w$.

On receiving $stag_w$, the *server* checks if the value exists in `IbSMT`. If $stag_w \notin \texttt{IbSMT}$, the *server* generates a proof of non-membership, pf_n, using `IbSMT`.*root* and submits it to SC_w. Upon evaluation, if the proof pf_n is found to be correct, then the *server* receives P_s coins, and the protocol aborts. If the proof is not valid, then the *server* does not get any payment, P_s gets unlocked, and the protocol is aborted.

Response and Commit: If $stag_w$ exists in `IbSMT`, the *server* searches for $stag_w$ in `TSet`. Then it encrypts $\mathbf{t}$ using key K_v and generates R_e. It shares R_e with the *user* and sends commitment of t (c_t), commitment of R_e (c_r), commitment of K_v (c_v), and h_w to SC_w.

Check Response: The *user* checks if the commitment of R_e is c_r. If not, then it complaints to SC_w and unlocks P_s coins. Else, the it regenerates $vtag_w$, which is used to generate h_w. The *user* submits $vtag_w$ to SC_w and locks additional P_m coins.

Algorithm 2: Fidelis.Search(w).

User ($\mathcal{U}$):
takes $K = (K_s, \bar{K}_s, K_T)$ and a keyword w.
computes $stag_w \leftarrow \texttt{TSetGetTag}(K_T, w)$
executes $\texttt{StoreandLock}(stag_w, P_s)$ in SC_w
  and locks P_s coins till time T_1.

Server ($\mathcal{S}$):
if $\texttt{IbSMT}.root$ in $SC_w \neq \alpha_s$, **then** $\mathcal{S}$ aborts
if *$stag_w$ does not exist* **then**
  $pf_n \leftarrow NMSearch(\texttt{IbSMT}, x)$
  if QueryValid$(pf_n) = valid$ **then**
    $\mathcal{S}$ gets P_s coins and protocol aborts.
  else
    P_s gets unlocked and abort.
else if *$stag_w$ exists* **then**
  $\mathbf{t} \leftarrow \texttt{TSetRetrieve}(\texttt{TSet}, stag_w)$
  $[e_1, e_2, \ldots, e_{n_w}, h_w] \leftarrow \mathbf{t}$; $K_v \xleftarrow{\$} \{0,1\}^\lambda$;
  $R_e \leftarrow \texttt{Sym.Enc}(K_v, \mathbf{t})$
  $c_t \leftarrow H_3(\mathbf{t})$; $c_v \leftarrow H_3(K_v)$; $c_r \leftarrow H_3(R_e)$
  $\texttt{SendCommitment}(c_t, c_r, c_v, h_w)$
  Sends R_e to the $\mathcal{U}$
end

User:
if $c_r \neq H_3(R_e)$ **then**
  $\mathcal{U}$ complains as $v_b =$ $\texttt{ComplainResponse}(R_e)$;
  if *v_b is* valid *bit* **then**
    B unlocks P_s coins and aborts.
else
  Computes $vtag_w \leftarrow H_1(\bar{K}_S, w)$
  $\texttt{StoreandLock}(vtag_w, P_m)$
  locks P_m coins by $\mathbb{B}$ in SC_w till time T_1.
end

Server:
if $h_w \neq H_2(vtag_w || e_1 || e_2 || \ldots || e_{n_w})$ **then**
  abort!
else $\texttt{Reveal}(K_v)$

User:
$[e_1, e_2, \ldots, e_{n_w}, h_w] \leftarrow Dec(K_v, R_e)$
$h'_w \leftarrow H_2(vtag_w || e_1 || e_2 || \ldots || e_{n_w})$
if $h'_w \neq h_w$ *or* $H_3(t) \neq c_t$ **then**
  $pf_e = (R_e, t)$
  executes $\texttt{Complain}(pf_e)$ by $\mathbb{B}$ in SC_w
  if *current time* $< T_2$ **then**
    $\mathbb{B}$ evaluates correctness of pf_e
  else $\mathcal{S}$ gets $P_m + P_s$ coins
else if $h'_w = h_w$ **then**
  $\mathcal{U}$ re-generates $K_w \leftarrow F(K_s, w)$;
  $\mathcal{U}$ decrypts each e_i as
   $id_i^w \leftarrow \texttt{Sym.Dec}(K_w, e_i)$
  $\mathcal{U}$ gets $R = \{id_1, id_2, \ldots, id_{n_w}\}$
  $\mathcal{S}$ receives payment after time T_2.
end

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Commit Reveal: Using $vtag_w$ and $e_i's$, if it is possible to generate h_w, then the *server* reveals the decommitment of c_v. Else, it aborts.

Algorithm 3: Smart Contracts.

```
Function Initialize(IbSMT.root):
  Record IbSMT.root in 𝔹
Function StoreandLock(value,P):
  Lock P in the contract
  Record value in 𝔹
  if current time is T1 then
    Unlock coins locked in contract for User
Function QueryValid(pf_n):
  if pf_n is valid w.r.t IbSMT.root then
    Unlock P_s for Server
Function SendCommitment(c_t,c_r,c_v,h_w):
  Record c_t,c_r,c_v,h_w in 𝔹
Function Reveal(K_v):
  Store K_v in 𝔹
  if current time is T2 then
    Unlock coins locked in contract for Server
Function ComplainResponse(R_e):
  if c_r ≠ H3(R_e) then
    Unlock coins locked in contract for User
Function Complain(pf_e):
  Parse pf_e
  if pf_e = (K_v,c_v) and c_v ≠ H3(K_v) then
    Unlock P_s + P_m for User
  else
    Compute t'=Dec(K_v,R_e)
    (e_1||e_2||...||e_{n_i},h_w) ← Parse(t')
    Compute h'_w = H2(vtag_w||e_1||e_2||...||e_{n_i})
    if h'_w ≠ h_w or R_e ≠ Sym.Enc(K_v,t) then
      Unlock P_s + P_m for User;
    else
      Unlock P_s + P_m for Server;
    end
  end
```

Response Retrieval: The *user* checks the commitment of K_v. If unmatched, it raises a complaint and submits a proof $pf_e = (K_v, c_v)$ to the function *complain* in SC_w before time T_2. The *client* has to raise the complaint and send proof of error pf_e within time T_2, else the *server* gets the payment $P_m + P_s$. The waiting times T_1 and T_2 can be fixed at the beginning of the protocol.

If K_v is valid, the *user* decrypts R_e using K_v and gets **t**. Next, he parses **t** to get $e_i's$ and recomputes h'_w. If $h'_w \neq h_w$ or $c_t! = Commitment(\mathbf{t})$, he submits a proof of invalid response to SC_w, again before time T_2. SC_w evaluates whether the complaint raised by the *user* is valid. If valid, then the *user* gets a refund of $P_m + P_s$ coins. If not, then the *server* gets the total amount for performing the task correctly.

If $h'_w = h_w$, the *user* decrypts e_is using $K_w = F(K_s, w)$. If the *user* has not raised a complaint within T_2, then the *server* claims the full amount.

Fidelis is verifiable by both the *user* and the *server*. If anyone cheats, it can be detected by others. Fairness is guaranteed as well. The *server* cannot get payment from the *user* without giving the correct result. Moreover, there is no way a *user* can get a correct result from the *server* without payment. Confidentiality is retained as there is no leakage of information that would benefit an adversary. The proofs justifying these properties along with the proof for correctness and soundness will be discussed in the full version of the paper.

4 PERFORMANCE ANALYSIS

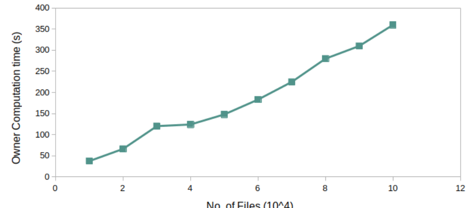
We implement and evaluate the protocol w.r.t. Ethereum as a blockchain platform. The smart contracts are written in Solidity language and deployed in the Ropsten test network.The gas price is 38.66 GWei (or 0.00007132 USD on 9th, February, 2023). We vary number of files from 10K to 100K, keywords from 35K to 109K and queries from 10 and 1000.

The gas cost for deploying the smart contract and for executing `Initialize` are around 163.957 USD (gas usage: 2298905) is 12.277 USD (gas usage: 172144) respectively. For querying a existing or a non-existing keyword, the gas cost for executing `StoreandLock` for a single query remains the same which is is around 7.366 USD (gas usage: 103287).

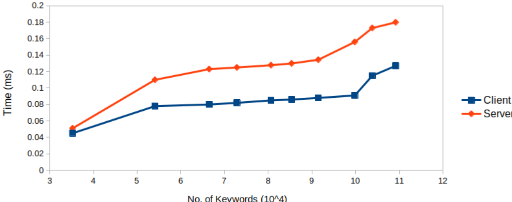
(i) *Searching existing keyword*: For a single existing keyword, executing `SendCommitment` takes 16s and costs is 9.487 USD (gas usage: 133021) payed by the *server*. They are same for `StoreandLock`. However, executing `Reveal`, they are 21 s, and 10.045 USD (gas usage: 140846). Finally, the time taken to finalize the payment is 18s and the gas cost is 2.857 USD (gas usage: 40063).

(i) *Searching a non-existing keyword*: The only function that gets executed in `QueryValid`, where the proof size varies with the size of the database. The time taken for execution varies between 20s and 34s whereas the gas cost increases slightly from 41.296 USD (gas usage: 579030) to 43.528 USD (gas usage: 610329).

Apart from this, we plot the initial encrypt time taken by *owner* in Fig.2(a). It varies between 38s



(a) Owner Computation Time upon varying file size



(b) User and Server's Computation Time upon varying number of keywords

Figure 2: Time taken for computation.

and 360s with the increase in the size of the database. For the bad case, when a given keyword is not a part of the database, the computation time of *user* is around 0.02ms but in the case of *server*, the time varies between 25ms to 70ms. If the keyword is part of the database, then with an increase in the number of keywords, the *user* time varies between 0.045ms to 0.1ms, and the *server* time varies between 0.05ms and 0.15ms, as shown in Fig.2(b).

We observe that querying existing keywords, the gas cost is around 23 USD. In Guo *et al.* (Guo et al., 2022), the gas cost for uploading a digest increases with the increase in index pairs, varying between 712.08 USD (gas usage: 9984351, for 1K index pairs) and 14,252.8388 USD (gas usage:199843506, for 16K index pairs). In Fidelis, the gas cost for uploading data/locking coins is around 46.877 USD (gas usage: 65727700), being invariant with the frequency of occurrence of the keyword. The cost of storage on-chain is fixed (only 320 B) for our protocol, compared to (Guo et al., 2022), where the cost of storage increases to 1MB when the number of index-pairs is 20K. For non-existing keywords, the smart contract needs to verify whether the proof sent by *server* is correct. The length of the proof increases with the size of the database, hence the gas cost is much higher.

5 CONCLUSION

We have proposed Fidelis, a blockchain-based searchable encryption scheme that operates without any trust assumption and ensures the verifiability of search results. We provide proof of its security and fairness. We deploy and test our prototype in the Ethereum Ropsten testnet with real-life data which demonstrates the feasibility and efficiency of our proposed scheme. We can see that the protocol can be executed by all involved parties efficiently.

REFERENCES

Buterin, V. (2020 (accessed November 4, 2020)). *Ethereum*.

Cash, D., Jarecki, S., Jutla, C. S., Krawczyk, H., Rosu, M., and Steiner, M. (2013). Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conf.*, pp 353–373.

Dahlberg, R., Pulls, T., and Peeters, R. (2016). Efficient sparse merkle trees - caching strategies and secure (non-)membership proofs. In *Secure IT Systems - 21st Nordic Conf., NordSec 2016*, vol. 10014 of *LNCS*, pp. 199–215.

Fan, C., Dong, X., Cao, Z., and Shen, J. (2020). VCKSCF: efficient verifiable conjunctive keyword search based on cuckoo filter for cloud storage. In, *19th IEEE Int. Conf. on Trust, Security and Privacy in Computing and Communications, TrustCom 2020*, pp. 285–292. IEEE.

Guo, Y., Zhang, C., and Jia, X. (2020). Verifiable and forward-secure encrypted search using blockchain techniques. In *ICC 2020 - 2020 IEEE Int. Conf. on Communications (ICC)*, pp. 1–7.

Guo, Y., Zhang, C., Wang, C., and Jia, X. (2022). Towards public verifiable and forward-privacy encrypted search by using blockchain. *IEEE Trans. on Dependable and Secure Computing*, pp. 1–1.

Jiang, S., Liu, J., Wang, L., and Yoo, S. (2019). Verifiable search meets blockchain: A privacy-preserving framework for outsourced encrypted data. In *2019 IEEE Int. Conf. on Communications, ICC 2019*, pp. 1–6. IEEE.

Li, H., Zhou, H., Huang, H., and Jia, X. (2020). Verifiable encrypted search with forward secure updates for blockchain-based system. In *Wireless Algorithms, Systems, and Applications - 15th Int. Conf., WASA 2020*, vol. 12384, *LNCS*, pp. 206–217. Springer.

Miao, Y., Tong, Q., Deng, R., Choo, K.-K. R., Liu, X., and Li, H. (2022). Verifiable searchable encryption framework against insider keyword-guessing attack in cloud storage. *IEEE Trans. on Cloud Computing*, 10(2):835–848.

Sardar, L. and Ruj, S. (2019). Fspvdsse: A forward secure publicly verifiable dynamic sse scheme. In *Provable Security - 13th Int. Conf., ProvSec 2019*, pp. 355–371.

Wang, J., Chen, X., Sun, S., Liu, J. K., Au, M. H., and Zhan, Z. (2018). Towards efficient verifiable conjunctive keyword search for large encrypted database. In *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018*, vol. 11099, *LNCS*, pp. 83–100. Springer.