This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing

Enabling Reliable Keyword Search in Encrypted Decentralized Storage with Fairness

Chengjun Cai, *Student Member, IEEE,* Jian Weng, *Member, IEEE,* Xingliang Yuan, *Member, IEEE,* and Cong Wang, *Senior Member, IEEE*

Abstract—Blockchain has led the trend of decentralized applications and shown great use beyond cryptocurrencies. Decentralized storage such as Storj and Sia leverages blockchain to establish an open platform for sharing economy, which provides private and reliable file-outsourcing services. However, the ubiquitous keyword search function over encrypted files is yet to be supported. To enable this function, we first apply searchable encryption techniques to the decentralized setting. But this primitive can hardly ensure the service integrity. The reason is that decentralized storage commonly faces severe threats from both clients and service peers. Service peers may return partial or incorrect results, while clients may intentionally slander the service peers to avoid payments. To address these threats, we utilize the smart contract to record the logs of encrypted search (aka evidence) on the blockchain, and devise a fair protocol to handle disputes and issue fair payments. Using a dynamic-efficient searchable encryption scheme as an instantiation, we craft a concrete scheme that preserves encrypted search capability and enforces ecosystem healthiness, so that service peers are incentivized to make real efforts and jointly guarantee service reliability. We implement our scheme in Python and Solidity, and test its search performance and transaction costs on Ethereum.

Index Terms—Decentralized storage, Blockchain, Financial fairness, Encrypted search.

1 Introduction

OUTSOURCING data to a decentralized storage platform like Storj [2], Sia [3], and Filecoin [4] has become an alternative choice for users to enjoy flexible storage capacity with reduced costs [5]. Compared to centralized cloud storages, decentralized storage relies on individual service peers to provide the leasing storage volume and the blockchain to achieve service integrity, e.g., by anchoring the service metadata like file digests and storage contracts between peers. To protect user privacy, systems like Storj further implement client-side encryption to ensure the confidentiality of files stored in distributed service peers [2]. However, without expressive functions like keyword search, users currently can only fetch encrypted files via their identifiers, which inevitably degrades the user experience. Therefore, in this paper, we aim to enable encrypted content search in this new paradigm. But employing existing encrypted search techniques [6], [7], [8], which are also known as searchable encryption, in the decentralized setting is a non-trivial task. Particularly, we need to properly handle the following two key challenges.

To start with, existing searchable encryption schemes focus on a centralized setting, where the encrypted indexes and files are stored in a cloud server. Here, how to deploy those techniques in a decentralized setting is challenging. To start with, the placement of outsourced files and indexes (e.g., inverted search index) should be considered, because they might be split into partitions and distributed in different peers [2], [9]. Also, it needs to be compatible with the existing decentralized storage platforms, where peers are contracted before providing services. A proper placement method is desired to reduce the communication overhead of a search query while enabling encrypted search in decentralized storage, and minimize the development effort of deploying our proposed service on top of existing decentralized storage platforms.

In addition, achieving reliable encrypted search service in this new paradigm faces more severe threats than the typical client-server model. Specifically, in this decentralized service model, we need to consider both dishonest clients and dishonest service peers, because they are individual players without enforced regulations. Here, service peers are individuals who join to contribute for monetary rewards, and they could delete clients' files and compute the search results lazily so as to save their computing resources [10], [11]. Although verifiable searchable encryption techniques [12], [13] can detect a dishonest service peer who returns incorrect search result, it is insufficient to handle dishonest clients. In fact, a dishonest client could fraudulently slander the service peers even though the search result is correct, so that it can tarnish their reputation or repudiate the service fees [2]. Apparently, it will largely discourage service peers to make real efforts in our proposed encrypted search service. Therefore, the proposed search service should ensure *fair exchange* between clients and service peers so as to enable a healthy ecosystem, guaranteeing that a service peer will always be rewarded

- *Chengjun Cai and Cong Wang are with the Department of Computer Science, City University of Hong Kong, Hong Kong SAR, China.*
  Jian Weng is with the College of Information Science and Technology, Jinan University, Guangzhou, 510632, China.
  Xingliang Yuan is with the Faculty of Information Technology, Monash University, 3128, Australia.
  Cong Wang is also with City University of Hong Kong Shenzhen Research Institute, Shenzhen, 518057, China.
  E-mail: chencai-c@my.cityu.edu.hk; xingliang.yuan@monash.edu; congwang@cityu.edu.hk.
  corresponding author: Jian Weng, cryptjweng@gmail.com.

A preliminary version of this paper [1] was published in 2017 International Conference on Communications (IEEE ICC 2017).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing

as long as it faithfully contributes in our system.

In this paper, we enable effective encrypted search service in decentralized storage by integrating searchable encryption techniques, and preserving reduced query latency and communication overhead. In addition, we address both the dishonest clients and dishonest service peers via enforcing a healthy ecosystem empowered by the smart contract techniques [14], with further customized designs to minimize both the computation and storage overhead on the blockchain.

Supporting encrypted keyword search in decentralized storage with efficient updates. Decentralized storage like Storj [2] and Sia [3] adopts a contract-based mechanism to connect a client with eligible service peers, and localizes the uploaded file (shards) *only* in contracted service peers. Hence to support encrypted keyword search in this setting, we have to preserve file and index locality, such that each file and its index will be located in the same service peer. Since if the encrypted index for search is stored elsewhere, extra search latency and communication overhead might be caused across different service peers. By preserving data locality, the corresponding encrypted files can be efficiently retrieved without an additional round of communication when processing a given query. Besides, we also need to support secure and efficient file updates, so that clients can readily add new files to our proposed service without leaking protected sensitive information [15].

Fair payment per authenticated use. In addition to encrypted search functionality, incentives for the service peers to provide reliable keyword search is another vital requirement in this new paradigm, since they are individuals who join the service for profits and have no enforced regulations [2]. Here, a natural idea to achieve this requirement is by letting the clients pay for the search services provided by the contracted service peer(s). However, directly adopting the above payment strategy is insecure in decentralized storage, since whoever made the first move would have disadvantages [16]: first, if the service peers are paid in advance by client, they might be motivated to lower the expenses by conducting services lazily or maliciously [10], [11]; second, if the search results are sent to the client in advance, the client might later intentionally repudiate the service fees, even though the service peers indeed consume expenses and provide correct search results [17].

Therefore, fair payment between dishonest clients and service peers is thus highly demanded for enabling a healthy ecosystem [2]. However, most existing methods require a trusted arbiter to serve as a middleman to enforce that service peers will be rewarded by the clients once they have provided correct services. But as we target a decentralized storage network, introducing a centralized third party might largely eliminate the benefits of a decentralized system [17]. Hence, in the paper we follow a decentralized approach to achieve fair payment, via leveraging the emerging smart contract that runs atop public blockchains, e.g., Ethereum [14]. At a high-level, smart contract will be crafted between client and service peer to 1) store client's deposit, and 2) trigger automatic payment from the deposit if a correct search result is provided. With this construction, a service peer now cannot claim reward without providing correct results, and a client cannot repudiate the payment since the money will be deduced automatically from its deposit via the smart contract.

But while the deposit of client and automatic payment are naturally supported by the smart contract, how to verify the returned search results via the smart contract with affordable cost is challenging. Intuitively, it seems that we can directly follow the idea of a recent Hawk framework [18] to support verification of results generated by off-chain service peers via zero-knowledge proofs (ZKP). However, if we directly use ZKP to prove correctness of an encrypted keyword search operation, enormous proof generation overhead could be incurred for the service peers, given that an encrypted search operation might require complex operations which scale with the search index size. Besides, smart contract is required to verify the correctness proof of every search query returned from a service peer, causing additional monetary cost (e.g., gas cost in Ethereum) which the contracted service peer might be reluctant to embrace [19].

Hence, we instead use smart contract to just anchor metadata of the search operations as undeniable evidences, and perform judgment operations only when dispute happens between client and service peers. At a high-level, instead of performing verification via the smart contract, we off-load the result checking process to the client, and issue a time-locked payment [20] to the service peers who provided the requested search service. During the lock time, the client can perform result verification to determine whether the result is correct, and issue a judgment request to the arbiter shard for judgment [21]. Fair judgment procedure will then be initiated by an arbiter shard to settle the dispute basing on previously anchored evidences, i.e., halting the time-locked payment if confirming that the service peer is dishonest. Consequently, we enforce a healthy ecosystem where service peers will always be rewarded once they have faithfully provided service, which thus can bring stronger incentive for service peer to make real efforts in our system. The contributions of this work are summarized as follows:

- We bring dynamic-efficient and secure keyword search to encrypted decentralized storage with reduced query communication overhead.
- We enforce healthy ecosystem to incentivize trustworthy behaviors in the proposed encrypted search service, and craft a concrete scheme Ω_{fair} that enables secure and dynamic queries while minimizing the execution and storage overhead on the blockchain for fair payments.
- We implement Ω_{fair} in Python and Solidity, and conduct experiments to test its latency in encrypted search processes and transaction cost on Ethereum. The experimental results demonstrate the search efficiency (around 460 ms in 400k records) in the long run and the affordable costs of our reliable payment design (the gas cost per transaction for search charge and judgment are \$0.64 and \$0.15 US dollar respectively, with an USD/ETH exchange rate \$410/ETH.).

The rest of the paper is structured as follows. Section 2 overviews some related works and Section 3 explains the background knowledge. After that, in Section 4 we elaborate the system model of our proposed design, including the architecture overview and threat model. Section 5 presents

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing

Scheme	Computation		Communication		Client Storage	Soundness	Fairness
	Search	Update	Search	Update			
KPR12 [7]	$O(r_w)$	$O(\|\mathsf{W_{id}}\|)$	$O(a_w)$	$O(\|\mathsf{W_{id}}\|)$	-	✗	✗
CJJ+14 [8]	$O(r_w)$	$O(\|\mathsf{W_{id}}\|)$	$O(a_w)$	$O(\|\mathsf{W_{id}}\|)$	-	✗	✗
HK14 [22]	$O(N)/O(1)$	$O(\|\mathsf{W_{id}}\|)$	$O(a_w)$	$O(\|\mathsf{W_{id}}\|)$	$O(M)$[1]	✗	✗
BFP16 [12]	$O(r_w + \log M)$	$O(\|\mathsf{W_{id}}\| \log M)$	$O(a_w + \log M)$	$O(\|\mathsf{W_{id}}\| \log M)$	-	✓	✗
B16 [23]	$O(r_w)$	$O(\|\mathsf{W_{id}}\|)$	$O(a_w)$	$O(\|\mathsf{W_{id}}\|)$	$O(M(\log D + \lambda))$	✓	✗
Ω_{fair}	$O(N + 2k_t)/O(1 + 2k_t)$	$O(\|\mathsf{W_{id}}\| + 2k_t)$	$O(a_w)$	$O(\|\mathsf{W_{id}}\|)$	$O(M(1 + \lambda))$	✓	✓

TABLE 1: Comparison with existing dynamic searchable encryption schemes. N is the number of entries (i.e., independent keyword/document identifier pairs) in the database, while r_w is the number of times the queried w has been *historically added*, a_w is the size of search result list of w, and $|\mathsf{W_{id}}|$ represents the number of keyword changes in an update. In particular, the search operation in [22] is executed in two cases, one is for the keywords that have not been searched, and the other one is for searched keywords, which result in computation complexity $O(N)$ and $O(1)$ respectively. k_t stands for the computation of generating one transaction, M is the number of distinct keywords, D is the number of documents, and λ represents the security parameter in the set hash function. Here, we consider only the single processor scenario.

our design of Ω_{fair}, the design rationale behind, and security analysis. In Section 6, we conduct experiments to test the efficiency and viability of our proposed design. Finally, we provide discussion on achieving forward security in Section 7, before we elaborate the conclusions in Section 8.

2 Related Work

Keyword search over encrypted data. Our proposed design is closely related to the line of works [6], [7], [8], [24], [25] (to list a few) called searchable symmetric encryption (SSE), i.e., efficient and secure search over the encrypted data. Since Curtmola et al. improve the security of SSE and introduce new constructions with sub-linear search time [6], many works have been devoted to enhance its functionalities, such as enabling dynamic operations [7], supporting boolean queries [26], [27], personalizing the search results [28], and extending SSE to support arbitrarily-structured data, e.g., graphs, labeled data or matrices [29]. However, all of these works address the security against a semi-honest adversary.

To mitigate the security threats from a malicious adversary that may return incorrect answers, another line of works, also named verifiable searchable encryption, have aroused interests in recent years [12], [23], [30]. Among which, [12] and [23] enable client-side verifiability by introducing authenticated structures like verifiable hash table (VHT) and set hashes [31]. Nevertheless, they only consider the typical client-server model and a honest client that will faithfully process the verification procedures. They do not consider the threats in the decentralized setting. Our recent work [1] focuses on achieving service reliability by enabling fair judgment on the integrity of search results, they do not consider financial fairness as their core requirement. Table 1 presents the comparison between Ω_{fair} and some dynamic searchable encryption schemes.

Blockchain applications. Emerging cryptocurrency blockchains and their respective stability protocols in P2P networks have shown great use beyond transferring money [32]. With the cryptographically auditable, immutable, and incentive driven ledgers they provided, many exemplary applications like decentralized storage [2], [4], distributed naming [33], sustainable supply chain [34], and medical records management [35] have been built and deployed in use. One key property of all applications above is that they employ the ledger as a tamper-proof content store to record their application data and ensure authenticity without putting trust in a single party, like a certificate authority (CA). Apart from them, blockchain-based cryptocurrency system like Bitcoin [36] can also be employed to devise fair protocols and ensure *financial fairness* [37] in many multi-user scenarios, such as the multi-party computations [20], [38]. With their proposed design, if one malicious party aborts the protocol, the money will be automatically transferred to the honest party. To ease the privacy concern on smart contracts, in [39], they store encrypted data on the blockchain and use zero-knowledge proofs to enforce the correctness of contract execution and money conservation. In light of previous works, in this work we aim to ensure service reliability and financial fairness in searchable encryption so that both the clients and service peers are incentivized to conduct trustworthy behaviors.

Decentralized storage services. Decentralized storage platforms [2], [3], [4], [40] are designed to comprise individual peers who are willing to lease their spare storage spaces and earn money. To ensure service integrity, most of them use blockchain to store the storage contracts and facilitate automatic payment with cryptocurrencies. Among which, Filecoin [4] implements blockchain-structured file storage and adopts proof-of-retrievability (PoR) to encourage proper maintenance of the outsourced files. But it does not provide privacy protection on the outsourced files. Sia [3] and Storj [2] enable end-to-end encryption to protect the confidentiality of files, but the user clients can only retrieve the outsourced files via their identifiers. In this paper, motivated by the limited search capability of these systems, we seek solutions from the searchable encryption techniques, and design an dynamic efficient scheme that supports trustworthy and private keyword search over the encrypted data on decentralized storage. Furthermore, we enable money conservation of our proposed query service to enhance its reliability.

3 Background

Blockchain. Blockchain serves as a fundamental structure of emerging cryptocurrencies such as Bitcoin [36]. In essence,

1. A search history θ is maintained by client for update operations, and the complexity is $O(M)$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing

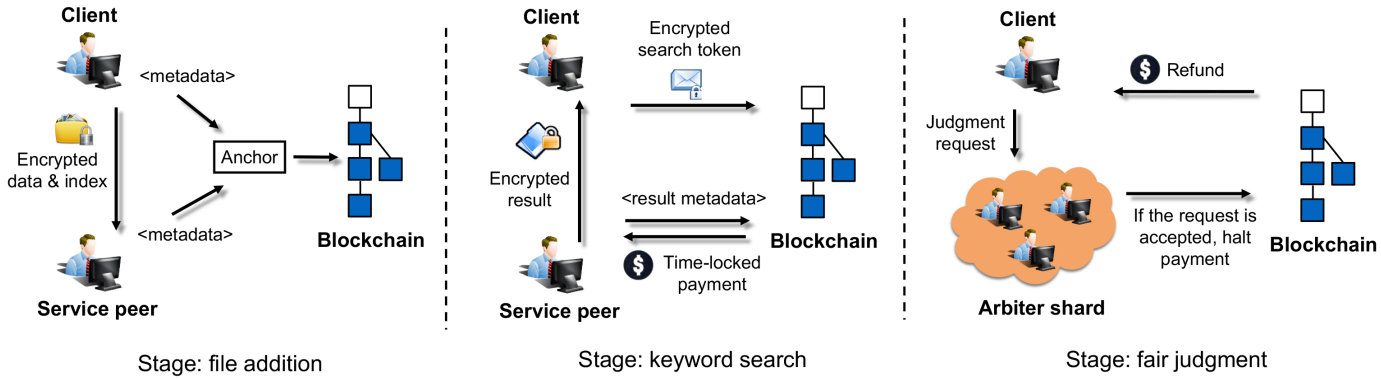


Fig. 1: Architecture overview.

blockchain is a distributed database where transactions (can be any format of data) are batched into an ordered list of continuously growing and time-stamped blocks. The main characteristics of the blockchain are shown as follows. *Transparency*: The transactions recorded on the blockchain are visible to all participants in the network. *Liveness*: all participants can reach the same blockchain and new blocks with valid transactions will continue to be added [39]. *Eventual consensus*: The transactions recorded on the blockchain are authenticated and a secure consensus protocol is ran among all participants to agree upon its global state [32].

Smart contract. In this paper, the smart contract is denoted as a self-enforcing contract programmed on the blockchain [14]. Specifically, each participant of the smart contract system runs a transaction-based state machine, begins with a genesis state and executes transactions on the blockchain to morph it into some final state. As only valid transactions are included on the blockchain, its final state can reach consensus automatically among all participants. Functions of different events (usually result in state changes) are preprogrammed in the contract and triggered when executing related transactions.

(Multi)set hash. (Multi)set hashing was introduced by Clarke *et al*. [31]. And it is defined as quadruple of probabilistic polynomial algorithms $(\mathbf{H}, \equiv_{\mathbf{H}}, +_{\mathbf{H}}, -_{\mathbf{H}})$ such that $\mathbf{H}$ maps sets included in the superset $\mathbb{S}$, for all $S \subset \mathbb{S}$,

- $\mathbf{H}(S) \equiv_{\mathbf{H}} \mathbf{H}(S)$.
- $\forall x \in \mathbb{S} \backslash S,\ \mathbf{H}(S \cup \{x\}) \equiv_{\mathbf{H}} \mathbf{H}(S) +_{\mathbf{H}} \mathbf{H}(\{x\})$.
- $\forall x \in S,\ \mathbf{H}(S \backslash \{x\}) \equiv_{\mathbf{H}} \mathbf{H}(S) -_{\mathbf{H}} \mathbf{H}(\{x\})$.

In this paper, we adopt the definition of (multi)set hash function $\mathrm{MSet-Mu-Hash}$ in [12]. It is defined as follows:

$$\mathbf{H}(S) : \mathbb{P}^{\mathbb{Z}} \rightarrow \mathbb{F}_q$$

$$S \mapsto \Pi_{x \in M} H(x)^{S_x}$$

where $H : \mathbb{P} \rightarrow \mathbb{F}_q$ is a hash function from the set $\mathbb{P}$ to the field $\mathbb{F}_q$ (q is a large prime power), M is a countable set for the multi-set S, and S_x is the multiplicity of x in S. Proven in [31], the set hash function $\mathbf{H}$ is collision-resistant as long as the discrete log assumption holds in $\mathbb{F}_q$ when H is modeled as a random oracle.

Searchable encryption. A searchable encryption scheme [6] comprises encryption, search, and (possibly) update algorithms. The encryption algorithm takes as input a secret key K, a document set D, and outputs a secure searchable index I and a sequence of ciphertexts C. The search algorithm takes as input an encrypted searchable index I, a secret key K, a keyword token t, and outputs the matched encrypted document set X. A dynamic SE scheme [7] includes the update algorithm, which takes as input K and a document f, and outputs an update token t' and an updated searchable index I'. In this paper, we follow the idea of [22] for its high efficiency on secure updates and for facilitating the file metadata anchoring process.

4 System Model

4.1 Architecture overview

As illustrated in Fig. 1, our proposed encrypted keyword search framework for decentralized storage consists of three types of actors, i.e., client, service peer, and an arbiter shard. Clients are the service requesters who want to outsource their personal files and later enjoy content search service, while service peers are individual nodes who are renting out their computing resources (i.e., storage and computation) for earning monetary rewards. Before a client can use our system for searchable and encrypted file storage, it has to establish connection with one or more service peers on the blockchain via grounding smart contracts [14]. Here, the connection between a client and a service peer can be established on their own or via a workforce market in decentralized storage platforms [2], and a peer in our system can switch its role in different contracts.

To securely outsource their personal files to contracted service peer(s), the files to be uploaded are encrypted. In the meantime, clients also label each file with an unique identity, and build a searchable index over the file identities to enable secure keyword search. Then, the encrypted files along with the built searchable index are sent to the contracted service peer, and metadata of the searchable index will be anchored on the blockchain as evidences. After completing the uploading process above, clients can construct and issue search tokens to the blockchain for conducting secure queries. Once noticing that a search token is issued, contracted service peer will retrieve the search token from the blockchain, and perform search operation using the previously received searchable index to obtain candidate search results, i.e., matched file identities. Then, the corresponding encrypted files are sent back to the requested client as query answer, and the service peer also records the metadata of the search result to the blockchain for 1) anchoring evidences and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing

2) triggering a time-locked payment (i.e., a smart contract enforced payment that will be issued after a period of time) for this search operation.

The above service payment will be issued to the service peer if and only if it faithfully provides correct search services. In particular, a two-layer authentication mechanism will be initiated to validate the returned search results. At a high level, validation of the returned search results will first be performed locally by the client during the lock time. Then, if the client detects incorrect results, he can issue a judgment request to the arbiter shard for fair judgments. Here, the arbiter shard is formed by a group of service peers that voluntarily serve to maintain the healthiness of the system, i.e., preventing dishonest clients from repudiating service fees. Specifically, basing on the evidences previously anchored on the blockchain, the arbiter shard can fairly judge the correctness of search results returned by the contracted service peer by re-executing the search operation in dispute. Next, accordingly, the payment will be halted if the judgment request from the client is accepted, i.e., confirming that incorrect result is returned. While no action will be made by the arbiter shard if the client is detected as dishonest, i.e., confirming that correct result is indeed returned, and the payment will be finalized after the lock time. Note that when both the clients and service peers behave faithfully without disputes, the arbiter shard will not be involved, and the contract will finalize payment for each query service after the lock time automatically.

The high-level workflow in our proposed encrypted and fair keyword search service consists of the following stages: 1) *Initialization.* The client establishes on the blockchain a smart contract defining the service connection between one or multiple service peers. 2) *File addition.* The client submits to the contracted service peer(s) the encrypted files and the encrypted index for later secure queries. 3) *Keyword search.* Client issues to the contracted service peer(s) the search token of a requesting keyword via the blockchain, the service peer(s) returns the result to the client, and the smart contract will automatically issue a time-locked payment for this search operation. 4) *Fair judgment.* If the client reports incorrect search results, the arbiter shard will fairly judge the correctness of the search result returned by the contracted service peer(s), and handle (i.e., proceed or halt) the time-locked payment accordingly.

4.2 Threat model

- *Dishonest clients and service peers.* We consider both the clients and the service peers can be dishonest [2], [12]. For the clients, they may deny the search services provided by the contracted service peers, and fraudulently repudiate the service fees [2]. For the service peers, we consider two aspects of malicious behaviors. On the one hand, they might try to learn from the client's private files and abuse them for their self-benefits [41]. On the other hand, they could be compromised by hackers, and later return incorrect search results back to the clients for frauds and scams [42].
- *Arbiter shard with byzantine fault tolerance.* Arbiter shard is introduced in our proposed design to facilitate the judgment process and halt time-locked payment if confirming that a service peer has provided unfaithful service. In our design, the arbiter shard is formed by a group of service peers that serving as a committee to make final judgment when a client issues judgment requests. Similar to the standard Byzantine threat model [21], we assume that more than γ (a fraction larger than $2/3$) of the active service peers in the arbiter shard are honest.
- *Standard public blockchain model.* Following the standard blockchain threat model in [39], blockchain in our proposed design is maintained by a set of validators, and is trusted for execution correctness and availability, but not for privacy. In this paper, we focus on protocol designs, and we do not consider potential program bugs [43] and future software updates [44] in Ethereum.

4.3 Security goals

Our security goals are to enforce the following two properties: (1) *Privacy*. Personal files and the issued keyword for search queries should be kept private along the entire search process. Both service peers and the arbiter shard should not be able to infer client's private files and issued keywords for search. (2) *Fairness*. It should be possible to fairly detect the exact wrongdoer if disputes happened between a client and a contracted service peer, and enforce fair payments. That is, if confirming that a contracted service peer has incorrectly conducted the requested search query, the service fee for this query should be repudiated. On the contrary, if confirming that the client has falsified the verification result, service fees for this query should be enforced.

5 Secure and Fair Keyword Search for Decentralized Storage with Efficient Updates

Before we illustrate our detailed construction, we first overview the techniques employed to address the aforementioned challenges respectively, elaborate the reasonings behind, and show how to integrate them seamlessly.

5.1 Design rationale

Our proposed design features a private and fair keyword search service atop decentralized storage platforms, with support for automatic payments and efficient file updates. To start with, in order to support expressive keyword search capability over encrypted files, we adopt the effective searchable encryption (SE) primitive, and ask each client to build an encrypted searchable index for the files to be uploaded. Intuitively, the client can later issue a search token (for a specific keyword) to whoever received the encrypted searchable index to securely retrieve corresponding file identifiers, so that the encrypted files corresponding to a keyword can be privately identified. So, here the first problem is how to facilitate the deployment of above encrypted search service in existing decentralized storage platforms.

We observe that decentralized storage platforms usually adopt a *contract-then-use* setting, where service peers in the platform are contracted before providing service to a client [2], [3]. Thus to enable encrypted keyword search service in this setting, we adopt the design choice of preserving

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing

file and index locality, where the encrypted files and their encrypted indexes are stored in the same contracted service peer. Particularly, this design choice brings three benefits. The first benefit is that the time cost of keyword search and file retrieval can be minimized because the interactions among service peers (to locate and retrieve corresponding encrypted files) are avoided. Meanwhile, the encrypted index in each service peer is not partitioned and stored by different service peers, and thus existing SE schemes can readily be applied in our service. At last, a client can also leverage the built-in service in existing decentralized storage platforms, e.g., "the publish/subscribe" system in Storj [2], to connect with eligible service peers (who are willing to provide both keyword search and storage services), and effectively initiate the required searchable storage service.

With the above design, we can enable effective encrypted keyword search service for encrypted decentralized storage platforms. However, there is no guarantee for service reliability, as service peers are recruited from the wild and it is very likely that some service peers may be dishonest in providing encrypted keyword search services later. At a first glance, it seems that we can simply provide monetary reward to the service peers as incentives, so that they are motivated to make real effort in each search operation. But this does not naturally enforce fair exchange between service and monetary rewards, and it is trivially understood that if we reward the service peers without verifying their works, the correctness of each search operation can be easily misplaced. Therefore, it is of immense significance that the monetary reward should be given to the contracted service peer if and only if it has faithfully provided a search service to the client, i.e., *pay-per-authenticated use*.

To achieve the fair exchange between correct service and monetary reward, the correctness of each returned search result is thus required to be verified. In the literature, bringing verifiability to search results in SE schemes can be achieved by letting the client to maintain a checklist (at local [23] or outsource via authenticated data structure [12]), so that the client can later check the correctness of each returned search result. But it is insufficient in our targeted decentralized storage setting, because client might also be dishonest, fraudulently denying correct search service to repudiate monetary rewards. Therefore, we seek solution from the emerging blockchain techniques [14], which can inherently guarantee fair exchange by having the client deposited, and automatically issuing monetary rewards to the contracted service peer if proving that correct search result is returned.

In order to leverage blockchain for issuing fair payments, a straightforward way is to let the contracted service peer to send the proof of a search operation to the blockchain for correctness checking, so that the blockchain can determine whether the contracted service peer has faithfully provided search service, and issues monetary reward accordingly. However, there are two disadvantages. Firstly, generating proof for an encrypted search operation can be very challenging and incur enormous computation overhead for the contracted service peers [18]. Secondly, on-chain operation for proof verification could incur highly uneconomical monetary cost that contracted service peer might be reluctant to embrace [19]. Instead, our idea is to off-load the task

Acronym	Definition
f	File
c	Encrypted file
w	Keyword
δ_f	Add token
τ_w	Search token
$\mathbf{H}$	(Multi)set hash function
θ	Search history
T_{client}	Client checklist
T_{peer}	Digest index
ζ_f	File index
ζ_w	Search index
$\mathbf{I}_w$	Posting list
$\mathbf{trans}$	Blockchain transaction
$ID(f)$	File identifier
h	Set hash
$\mathbb{H}$	Cryptographic hash function

TABLE 2: Glossary

of result correctness checking to a special committee called arbiter shard, which is formed by a group of voluntarily joined service peers serving to maintain service healthiness. Besides, we further reduce their work-load by crafting a two-layer authentication mechanism where client-side verification is used as the first layer authentication, so that the arbiter shard is involved (as the second layer authentication) only when a client reports incorrect search result.

At a high-level, our proposed mechanism works as follows. Firstly, when the contracted service peer returns the search result to the client, a time-locked payment will be made to the contracted service peer. Then, during the lock time, the client can locally check the validity of the returned search result for keyword w, i.e., $\mathbf{I}_w$, via the checklist T_{client} of search result digests [23]. Then, if the verification fails, the client can issue a judgment request to the arbiter shard for halting the time-locked payment. Later, the arbiter shard will ask the corresponding service peer to provide the encrypted index to each node in the arbiter shard for re-execution. According to the search result generated from the re-execution, a judgment can be made to determine whether the judgment request from the client is valid. Particularly, if the judgment request is accepted by the arbiter shard, the time-locked payment will be halted by the arbiter shard.

But because the re-execution is based on the encrypted index provided by the service peer in dispute, we have to validate the index before re-execution. Also, we need a customized design to check the search results returned by the contracted service peer using results generated from re-execution, so that fair judgment can be made at each arbiter node. At last, as the judgment process is performed independently at each node in the arbiter shard, we need to consider how to reach consensus on a final judgment result, i.e., whether to halt the payment. In our design, to deal with the first requirement, we leverage the blockchain to store metadata of the file addition process, so that nodes in the arbiter shard can later use them as evidences for verifying the provided encrypted index. While for the second requirement, our idea is to previously anchor the digest of each added file to the blockchain, and later ask the contracted ser-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing

vice peer to anchor the digest of each returned search result on the blockchain. In this way, when obtaining the search result from re-execution, each node in the arbiter shard can readily aggregate the corresponding file digests (through the incremental property), and then use the aggregated digest to compare with the one anchored by the contracted service peer for fair judgment. To support this customized design, we resort to a dynamic efficient SE scheme proposed in [15], and integrate it with incremental set hash [31] as an instantiation. Lastly, for consensus judgment in the arbiter shard, we follow the idea of [45] and use a voting mechanism for reaching consensus, and the result is that if the judgment result is accepted by the majority of nodes (i.e., a fraction $> 2/3$) in the arbiter shard, the time-locked payment will be halted. Now, we will illustrate our concrete scheme designs.

5.2 Ω_{fair}: reliable searchable encryption with fairness

Let F: $\{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ be a secure pseudo-random function (PRF), H: $\{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ be a key-based hash function, **H** be a collision-resistant $\mathrm{MSet-Mu-Hash}$ function [31], (Enc, Dec) be an CPA-secure (Chosen-Plaintext-Attack) encryption scheme, and $\mathbb{H}$ be a collision-resistant cryptographic hash function. Besides, the client, the contracted service peer, and each node in the arbiter shard have identities associated with their accounts on the blockchain, denoted as C, S, and A_i respectively. For clear illustration, we will assume a client C later will only contract with one service peer S for file storage and keyword search service.

Initialization. In the initialization phase of the proposed system, every participant (including client and service peer) generates a public/secret key pair, denoted as (pk, sk), and publish the public key pk through a key publishing mechanism like PGP [46]. We omit the security threats of PGP's key publishing mechanism and assume that each published public key is securely maintained by all participants. The secret key sk of every participant is stored secretly at local, and it is later used for generating transaction signatures.

Next, the client C interacts and makes agreement with a service peer, so as to establish a smart contract on the blockchain via a standard contract mining procedure [14]. Specifically, the contract records service information, which includes: 1) the account addresses of the client C and contracted service peer S on the blockchain; 2) the service duration (can refer to either real physical times or block indexes); 3) the service fees settled for each keyword search operation; Besides, two payment functions and a revoke function are programmed:

- Function deposit that subtracts money from the client's account and deposits it into the contract.
- Function search that freezes a pre-defined amount of money from the client's deposit for each keyword search operation, and sets up a t-time-locked payment.
- Function judge that revokes the time-locked payment for a performed search operation.

Note that function judge can only be triggered by nodes in the arbiter shard. Both the client C and contracted service peer S can learn about whether the duration of the established service is finalized (e.g., using the Greenwich Mean Time (GMT) as the reference for real physical time, or the consensus block state at local as the reference for block index).

Additionally, the client C initializes k_1, k_2, T_{client}, θ, and **H**, where k_1 and k_2 are the secret keys, T_{client} is an empty set to maintain set hashes for search result verifications, θ is an empty list to store search history, and **H** is a collision-resistant set hash function. In the meantime, the contracted service peer initializes the encrypted file index ζ_f, the search index ζ_w, the posting list $\mathbf{I}_w$ that stores the keyword search results, and an empty table T_{peer} to store set hashes. The set hash function **H** will be securely shared with S.

File addition. Given a file f to be uploaded, this process builds up the add token and anchors file metadata to the blockchain for fair judgment. Specifically, the client C first parses f into distinct keywords $\sigma = (w_1, ..., w_{len(\sigma)})$, where σ denotes the distinct word list of f. Then, using the secret key k_1 and a sequence of pseudorandom values $(s_1, ..., s_{len(\sigma)})$, the corresponding ciphertext of each distinct keywords parsed above can be constructed, as in Step 4 of Fig. 2. The generated ciphertexts for all keywords $\{c_i\}$ are then combined to create the encrypted file index c^*. In addition to the encrypted file index, the client also generate the file id $ID(f)$ from f, a reference list $\mathbf{x}$ that records the keywords searched before, and the file ciphertext c generated via $Enc_{k_2}(f)$, where k_2 is a secret key specifically for encryption. Accordingly, the add token for file f is constructed as $\delta_f = (ID(f), c^*, \mathbf{x}, c)$. Besides, the client also generates a digest h' via $\mathbf{H}(ID(f))$, where **H** is the set hash function, so as to incrementally update file f's digest (i.e., h') into the local checklist T_{client} for later result verification, as shown in Step 4c) in Fig. 2.

Next, to anchor metadata for this file addition process, the client hashes δ_f via $\mathbb{H}(\delta_f)$, where $\mathbb{H}$ can be a standard cryptographic hash function. Then, the digest h' and $\mathbb{H}(\delta_f)$ are combined to create an add transaction:

$$\mathbf{trans}_{add}^{\mathcal{C}} = (h', \mathbb{H}(\delta_f))$$

which is sent to the blockchain for anchoring metadata for this file addition process. In the meantime, the add token δ_f is sent to the contracted service peer S.

Before the file metadata is anchored on the blockchain, it has to be confirmed by the contracted service peer so as to prevent client from anchoring fraudulent metadata [47]. Specifically, once receiving δ_f, the contracted service peer S needs to independently generates h' through the shared set hash function **H** and the hash digest of δ_f via $\mathbb{H}(\delta_f)$. Then, the contracted service peer generates an add transaction of its generated h' and $\mathbb{H}(\delta_f)$:

$$\mathbf{trans}_{add}^{\mathcal{S}} = (h', \mathbb{H}(\delta_f))$$

and sends it to the blockchain. After receiving add transactions from both client and contracted service peer, i.e., $\mathbf{trans}_{add}^{\mathcal{C}}$ and $\mathbf{trans}_{add}^{\mathcal{S}}$, the blockchain will check whether they are identical. If $\mathbf{trans}_{add}^{\mathcal{C}}$ and $\mathbf{trans}_{add}^{\mathcal{S}}$ record the same metadata, blockchain will accept this file addition process, and the contracted service peer will update the file index ζ_f, the search index ζ_w, and the digest index T_{peer} accordingly with δ_f, as illustrated in Step 8 and 9 of Fig. 2. Otherwise, if

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing

Protocol: file addition

FOR CLIENT C:

1. $c \leftarrow Enc_{k_2}(f)$ // encrypt the file to be uploaded
2. Parse f for distinct keywords $\sigma = (w_1, ..., w_{len(\sigma)})$
3. Compute $h' = \mathbf{H}_{k_3}(ID(f))$
4. **for each** $w_i \in \sigma$:
   a) Generate a pseudorandom value s_i
   b) Compute $\tau_{w_i} = F_{k_1}(w_i)$
   c) **if** $T_{client}[\tau_{w_i}]$ exists,
      $h \leftarrow T_{client}[\tau_{w_i}]$, $h^* = h +_{\mathbf{H}} h'$
      else, $h^* = h'$
   d) Add $(\tau_{w_i}, h^*) \rightarrow T_{client}$
   e) **if** $\tau_{w_i} \in \theta$, add $\tau_{w_i} \rightarrow \mathbf{x}$
   f) $c_i \leftarrow H_{\tau_{w_i}}(s_i)||s_i$
5. Sort $c^* = (c_1, ..., c_{len(\sigma)})$ in lexicographic order and set $\delta_f = (ID(f), c^*, \mathbf{x}, c)$
6. Send $\mathbf{trans}^{\mathcal{C}}_{add} = (h', \mathbb{H}(\delta_f))$ to the blockchain
7. Send δ_f to the contracted service peer

FOR SERVICE PEER S:
Once receiving an add token δ_f from the client:

8. Compute $h' = \mathbf{H}_{k_3}(ID(f))$ and $\mathbb{H}(\delta_f)$
9. Send $\mathbf{trans}^{\mathcal{S}}_{add} = (h', \mathbb{H}(\delta_f))$ to the blockchain
10. **for each** $x_i \in \mathbf{x}$:
    a) Add $ID(f) \rightarrow \zeta_w[x_i]$
    b) $h \leftarrow T_{peer}[x_i]$, $h^* = h +_{\mathbf{H}} h'$
       $T_{peer}[x_i] \leftarrow h^*$
11. Add c to the leasing storage
12. Insert c^* to ζ_f

Fig. 2: File addition process between a client and a contracted service peer.

$\mathbf{trans}^{\mathcal{C}}_{add}$ and $\mathbf{trans}^{\mathcal{S}}_{add}$ are conflicted, the file addition process for file f aborts and δ_f will not be used by service peer S.

Keyword search. Given a keyword w to perform keyword search, this process generates the search token τ_w from w, retrieves search results, and records a search transaction on the blockchain for anchoring result metadata. In detail, the client C first generates the search token τ_w for keyword w via $\tau_w \leftarrow F_{k_1}(w)$, where F_{k_1} is the PRF function with key k_1. Then, the client sends the search token τ_w to the blockchain, and updates τ_w in the search history θ for facilitating list $\mathbf{x}$ construction in a new file, as in Step 4e) of Fig. 2.

After retrieving τ_w from the blockchain, the contracted service peer S processes τ_w with different indexes depending on whether τ_w has been searched before. On the one hand, if τ_w has been searched before, the contracted service peer can efficiently retrieve the search result $\mathbf{I}_w$ directly from the search index ζ_w and the set hash digest h_m from the digest index T_{peer}. On the other hand, if τ_w has not been searched before, the contracted service peer needs to linearly scan the file index ζ_f to retrieve the search result $\mathbf{I}_w$ and construct its integrity proof h_m accordingly. Explicitly, it scans every c^* in ζ_f and generates the search results by checking whether $H_{\tau_w}(r_i) = l_i$ for each $c_i \in c^*$, as in Step 8a) of Fig. 3, where r_i equals to the pseudorandom value s_i

Protocol: keyword search

FOR CLIENT C:

1. Given a keyword w
2. $\tau_w \leftarrow F_{k_1}(w)$
3. $\theta' = \theta \cup \{\tau_w\}$ // Renew the search history
4. Send τ_w to the blockchain

FOR SERVICE PEER S:
Once retrieving a search token τ_w from the blockchain:

5. Initialize result digest h_m
6. Parse ζ_w and check if $\zeta_w[\tau_w]$ exists,
7. **If** exists,
   Retrieve $\mathbf{I}_w \leftarrow \zeta_w[\tau_w]$ and $h_m \leftarrow T_{peer}[\tau_w]$, jump to step 10
8. **else, for each** $c^* \in \zeta_f$,
   a) **for each** $c_i \in c^*$ that is $i \in [1, len(c)]$, set $c_i = l_i||r_i$ and check if $H_{\tau_w}(r_i) = l_i$
   b) **If** $H_{\tau_w}(r_i) = l_i$,
      i) Retrieve related $ID(f)$ and h'
      ii) Insert related $ID(f)$ into $\mathbf{I}_w$
      iii) $h_m = h_m +_{\mathbf{H}} h'$
9. Create new entries $\zeta_w[\tau_w] \leftarrow \mathbf{I}_w$ and $T_{peer}[\tau_w] \leftarrow h_m$
10. Return the corresponding encrypted files $(c_1, ...c_n)$ and $\mathbf{I}_w$ back to the client
11. Send $\mathbf{trans}_{search} = (\tau_w, h_m)$ to the blockchain

BLOCKCHAIN:
Once receiving $\mathbf{trans}_{search}$ from the service peer:

12. Set up a lock time t // a system parameter
13. Issue a t-time-locked payment

Fig. 3: Keyword search process for a search token τ_w.

and l_i is the left k-bits of c_i. Besides, it aggregates each set hash digest of the retrieved $ID(f)$ to generate h_m.

In either case above, the retrieved search result $\mathbf{I}_w$ and corresponding file ciphertexts $\{c\}$ are returned to the client. In the meantime, the contracted service peer generates a search transaction:

$$\mathbf{trans}_{search} = (\tau_w, h_m)$$

to anchor the result metadata to the blockchain, and the blockchain will automatically issue a t time-locked payment to the contracted service peer. Note that after the keyword search process above, τ_w is now denoted as a searched token, and the search results $\mathbf{I}_w$ and integrity proof h_m will be updated to $\zeta_w[\tau_w]$ and $T_{peer}[\tau_w]$, respectively.

Once obtaining the search result, i.e., $\mathbf{I}_w$ and file ciphertexts $\{c\}$, the client decrypts each ciphertext via $Dec_{k_2}(c)$ and checks whether $\mathbf{I}_w$ matches the decrypted files. Then, the client verifies $\mathbf{I}_w$ through the local checklist T_{client} by generating a challenge set hash digest h^* via $h^* \leftarrow \sum_{ID \in \mathbf{I}_w} \mathbf{H}(ID)$, and comparing it with the one recorded in $T_{client}[\tau_w]$. Through the above client-side verification, incorrect search result is thus can be detected. Particularly, if the client has detected incorrect search result, a judgment request will be issued to the arbiter shard to initialize the fair judgment process, as illustrated below.

Fair judgment. Given a judgment request issued by the client, the process fairly judges the correctness of the disputing search result, and halts the time-locked payment

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing

if the client's judgment request is accepted. Specifically, once receiving a judgment request from the client C, the arbiter shard issues a response request to ask the contracted service peer S to provide corresponding encrypted indexes for judgment. The contracted peer then broadcasts all corresponding add tokens $\{\delta_f\}$ sent by the client to the arbiter shard. To fairly judge the correctness of the disputing search result, each node in the arbiter shard, i.e., A_i, first checks that each received add token δ_f matches its hash digest $\mathbb{H}(\delta_f)$ anchored on the blockchain. Then if the add tokens are verified, the challenge file index ζ_f^* is built from $\{\delta_f\}$ to process the corresponding search token τ_w anchored on the blockchain, and a search result set $\mathbf{I}_w^*$ is generated via linearly scanning ζ_f^* for matched file identifiers, as in the process *keyword search* for a search token that has not been searched before. With the generated search result $\mathbf{I}_w^*$, a challenge set hash digest h_c can thus be obtained by aggregating corresponding files' set hash digests on the blockchain, as in Step 6 and 7 of Fig. 4, so that h_c can be compared with the the contracted service peer's result digest h_m for judgment. Next, if h_c does not match h_m, the contracted service peer is denoted as dishonest, and a vote transaction (as a `halt` response) will be sent to the blockchain for halting the issued time-locked payment. Here to reach consensus among all nodes in the arbiter shard, a consensus-voting process will be launched.

In detail, each node A_i is assigned with one vote, and the threshold parameter γ (a fraction $> 2/3$) is used together with N, which is the arbiter shard size, to define a threshold number of votes for reaching consensus on a judgment while preserving byzantine fault tolerance against (possible) dishonest nodes in the arbiter shard. Hence, if a judgment request from the client has received more than $N \cdot \gamma$ votes, the time-locked payment will be halted.

Protocol: fair judgment

Once receiving a judgment request from the client:

FOR NODE A_i IN THE ARBITER SHARD:

1. Collect related add tokens $\{\delta_f\}$ from the contracted service peer
2. Check validity of the collected $\{\delta_f\}$ via $\{\mathbb{H}(\delta_f)\}$
3. $\zeta_f^* \leftarrow \{\delta_f\}$ // Build challenge file index
4. Retrieve τ_w from the blockchain
5. Re-execute keyword search process for τ_w in ζ_f^* to obtain search result $\mathbf{I}_w^*$, as in Step 8 of Fig. 3
6. Parse the blockchain for anchored set hashes $\{h'\}$ that correspond to $\mathbf{I}_w^*$.
7. Initialize h_c and $h_c \leftarrow \sum_{h \in \{h'\}} h$ // using $+_{\mathbf{H}}$
8. **if** $h_c = h_m$ anchored on the blockchain:
   send $(\texttt{halt}, \tau_w)$ to the blockchain

BLOCKCHAIN:

$counts \leftarrow \{\}$, $voters \leftarrow \{\}$

Upon receiving a `halt` response from an arbiter node A_i:

9. $\tau_w \leftarrow \texttt{halt}$
10. **if** $\mathsf{A}_i \notin voters$ **then**
    $voters \bigcup = \{\mathsf{A}_i\}$
    $counts[\tau_w] += 1$
    if $counts[\tau_w] > N \cdot \gamma$ **then**
    // if we have enough votes
    Halt the payment issued in Step 13 of Fig. 3

Fig. 4: Fair judgment process in case of disputes.

5.3 Security Guarantees

Confidentiality. To elaborate the confidentiality of files and query keywords in Ω_{fair}, we define three leakage functions, namely $\mathcal{L}_{add}$, $\mathcal{L}_{search}$, and $\mathcal{L}_{encrypt}$. First, $\mathcal{L}_{add} = (ID(f), len(\sigma), history(\sigma), \mathbf{H}(ID(f))$, where $ID(f)$ is the document identifier of f, σ is the unique keyword set of f, $\mathbf{H}(ID(f))$ is the set hash of $ID(f)$, and $history(\sigma)$ is a set which equals to $\{ID(f_i) : \forall w_i \in \sigma$ and $\tau_{w_i} \in \theta, ID(f_i) \in \zeta_w(\tau_{w_i})\}$. ζ_w is the search index maintained by the service peer and θ is the client's search history list. Second, $\mathcal{L}_{search} = (\mathsf{Access}(w), \tau_w)$. Here, $\mathsf{Access}(w)$ is the access pattern defined as a set that includes all returned search results $\{ID(f_i) : w \in f_i$ and $f_i \in \mathbf{f}\}$, where $\mathbf{f}$ is the set of all outsourced files and w is a query keyword. The search token τ_w elaborated here indicates the repeated query keywords, but it does not tell the content of keywords. Finally, $\mathcal{L}_{encrypt} = len(f)$ defines the encryption leakage, which includes only the length of f.

***Theorem* 1.** *Let* Π *be the algorithms of secure file add and encrypted search.* Π *is* $(\mathcal{L}_{add}, \mathcal{L}_{search}, \mathcal{L}_{encrypt})$*-secure against adaptive chosen-keyword attacks in the random oracle model if* (Enc, Dec) *is semantically secure, and* F, H *are secure PRF.*

Proof. We aim to prove that a PPT adversary cannot differentiate the views of search query processes in a real-world protocol and a simulated protocol. Here, the adversaries can conduct q query requests and p add requests adaptively. First, given $\mathcal{L}_{add}$ and $\mathcal{L}_{encrypt}$, a simulator can generate a dummy add token $\tilde{\delta}$, where the size of $\tilde{\delta}$ is identical to δ, but all ciphertexts in $\tilde{\delta}$ are random strings. Obviously, with the pseudo-randomness of the PRF and semantic security of the symmetric encryption scheme, the adversary cannot distinguish $\tilde{\delta}$ from δ, and hence cannot distinguish the built dummy file index $\tilde{\zeta}$ from ζ. Second, given $\mathcal{L}_{search}$, the simulator can simulate search tokens and ciphertexts adaptively. For every query request, if the keyword is not quested before, the simulator simulates the search token $\tilde{\tau}_w$ and randomly select $|\mathbf{I}_w|$ ciphertexts from $\tilde{\zeta}$ by replacing the PRFs with random oracles. Otherwise, it returns the same ciphertexts generated before. We observe that due to the pseudo-randomness of the PRFs, the adversary cannot distinguish between the tokens and search results in each query request, and thus will have negligible advantage to win the chosen keyword game later. We refer the readers to [22] for more details.

□

Soundness. To enable result verification on the client-side, we employ the set hash function to securely aggregate set hash digests of all files, and compare the aggregated digest with the one previously recorded in the client's local checklist. Given the collision-resistant assumption of our adopted set hash function $\mathbf{H}$, the probability that the contracted service returns an incorrect search result to the client while passing the client's verification is negligible. Since we can hardly find two different sets, i.e., $\mathbf{I}_1 \neq \mathbf{I}_2$, such that $\sum_{ID \in \mathbf{I}_1} \mathbf{H}(ID) = \sum_{ID \in \mathbf{I}_2} \mathbf{H}(ID)$ [31].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing

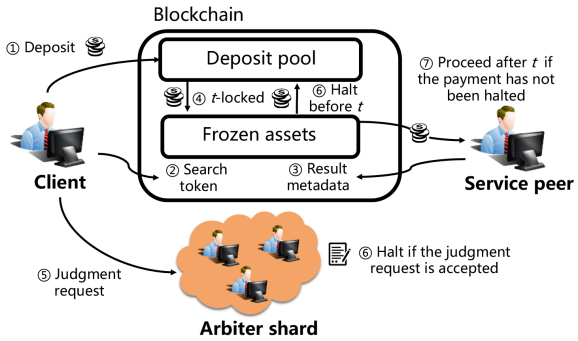


Fig. 5: Illustration of our design for achieving fair payments via the blockchain. t is a system parameter defining when an issued payment is finalized on the blockchain.

Judgment Fairness. Recall that we leverage the blockchain as an immutable log for anchoring both the metadata for the file addition process, and the search token and result metadata of the keyword search process. Therefore, when the client issues a judgment request to the arbiter shard, each arbiter node can check the validity of the provided add tokens, re-execute the disputing search operation, and fairly make a judgment for deciding whether to accept the client's request.

First, to authenticate the add tokens provided by the contracted service peer in dispute, we anchor hash digests of each add token on the blockchain. Specifically, two add transactions, i.e., $\textbf{trans}^{\mathcal{C}}_{add}$ from the client and $\textbf{trans}^{\mathcal{S}}_{add}$ from the contracted service peer respectively, are sent to the blockchain for metadata anchoring. The main reason for sending two similar transactions separately is to ensure that the file metadata to be anchored is confirmed by both the client and the contracted service peer [47]. Here, anchoring correct and undeniable file metadata as evidences is fundamental to our subsequent fair judgment process, because it enables each arbiter node to re-execute an authenticated keyword search operation and retrieve the trustworthy search result. With the trustworthy search result, each arbiter node can use the anchored set hash digest of each file to compute a challenge set hash digest h_c, as shown in Step 7 of Fig. 4, and then compare it with the digest h_m sent by the contracted service peer for judgment.

Second, note that the judgment is performed independently by each arbiter node, thus a consensus reaching process among all nodes in the arbiter shard is required to finalize a judgment result, i.e., whether or not to halt the t-time-locked payment issued to the contracted service peer. Here, we follow the standard Byzantine voting mechanism to assign each arbiter node with one vote, and leverage a threshold $N \cdot \gamma$ to identify a consensus-reached judgment. To ensure that the above consensus-reaching process can be processed successfully, the fraction of active honest nodes in the arbiter shard in our design is assumed to be than $2/3$, i.e., $\#honest > 2/3$, where $\#honest$ is the number of honest nodes in the arbiter shard, as illustrated in Section 4.2. This condition enforces that if the client's judgment request is valid, i.e., the contracted service peer has indeed provided incorrect search result, the locked payment will be halted, since honest arbiter nodes will vote to help the payment halting process pass the threshold. Besides, it also implies that if the client's judgment request is invalid, i.e., the contracted service peer has indeed provided correct search result, the locked payment will not be halted, since there will not be sufficient votes from the honest arbiter nodes. Therefore, our proposed design enforces a healthy ecosystem where a service peer will be fairly rewarded as long as it has faithfully provided search services. The entire workflow of our proposed fair payment design is presented in Fig. 5.

```
contract tlocked_searchpay {
   mapping (address => int) public balanceOf;
   mapping (address => int) ARBITER; // log arbiter nodes
   mapping (address => int) depositpool;
   address CLIENT, PEER, DEPOSIT;
   string token_now;
   int fee, t, threshold; // define parameters
   int T, votes;
   bool paysearch; // false will revoke the payment
   function deposit(int _value) returns (bool) {
     // authenticate message sender
     assert (msg.sender == CLIENT);
     // check if client has enough money
     assert (balanceOf[CLIENT] >= _value);
     balanceOf[CLIENT] -= _value;
     balanceOf[DEPOSIT] += _value;
     // add budget to our service
     depositpool[CLIENT] += _value;
   }
   function search(string token, string sethash) {
     // send search request
     if (msg.sender == CLIENT) {
      assert (depositpool[CLIENT] >= fee);
      token_now = token; // record the token to search
     }
     // claim money for a search operation
     if (msg.sender == PEER && token_now != None) {
   // check integrity then issue a t-locked payment
      assert (paysearch = isEqual(token_now, token));
      T = block.number + t; // current block number + t
      Alarm Clock Service(T, callback());
      depositpool[CLIENT] -= fee; // reduce budget
      token_now = None;
     }
   }
   function judge() returns (bool) {
     assert (ARBITER[msg.sender] != 0);
     votes += 1;
     if (votes > threshold) {
      paysearch = false; // revoke payment
      depositpool[CLIENT] += fee;
     }
   }
   function callback() returns (bool) {
     if (paysearch == true && block.number > T)
    // call public event Transfer to finalize the payment
       Transfer (DEPOSIT, PEER, fee);
   }
}
```

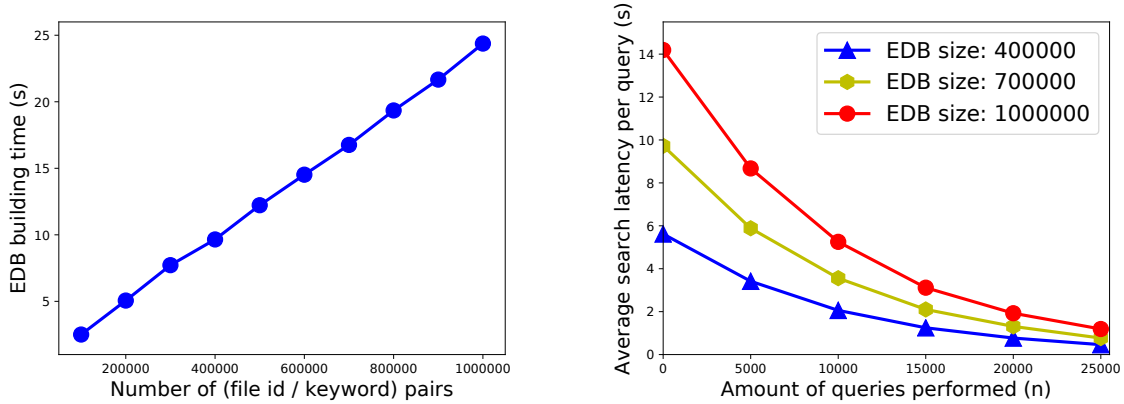
Fig. 6: Code sketch of search operation payment in a smart contract. Client will send "0" for the set hash input in `searchcharge` function. Here we assume an arbiter shard with 32 arbiter nodes for demonstration purpose.

6 Experimental Results

We implement the algorithms of client and service peer in Ω_{fair} using Python and construct the ethereum smart contract using Solidity[2], with a bit more than 2000 lines of codes. We adopt the Python cryptography library[3], and the Fernet symmetric encryption algorithm with a 256-byte key. The

2. On line at: https://solidity.readthedocs.io/en/develop/.
3. On line at: https://cryptography.io/en/latest/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing



(a) The EDB building time, as a function of the scale of keyword dataset.

(b) The search latency, as a function of the amount of queries performed.

Fig. 7: Experimental evaluation results on the update and search algorithm. The Google 10000 common keyword list is adopted and the largest census dataset is realized as a 1,000,000 record table of (w, id) pairs, where w is an independent keyword and id is the corresponding file identifier. The search latency scales down rapidly as more queries are performed and recorded on the search index, while the processing latency scales up as the size of the original dataset pairs increases.

hash function is the Python built-in implementation of SHA-256 [48]. The two key-based pseudo-random functions are implemented using HMAC-SHA256 from the cryptography library. The first parameter is treated as a key to the HMAC, and the second parameter is the input for the HMAC. For the set hash function, we implement it using the standard hash function SHA-256 as described above. The contract is programmed using the Solidity language that is designed to target the Ethereum Virtual Machine (EVM) [14]. We ran our experiment of the python programs on a desktop computer with a Quad-core CPU of 3.4 GHz, 16GB of RAM, a 256 GB SSD, and a Linux 16.04 operating system. The smart contract is tested with the Ethereum blockchain using a local simulated network *TestRPC* [49].

6.1 Update and Search Performance

We craft our test datasets for building file indexes and conducting queries using the Google's 10,000 most common English words (Google 10000) [50]. The dataset is realized by randomly choosing words within the keyword dataset and filling them into a set of files separately. The largest census dataset we use contains 1,000,000 records of (w, id) pairs, where w and id represent keyword and file identifier respectively. To better illustrate the performance result of the search algorithm, we further build up several smaller datasets, ranging from 100,000 to 900,000 records of (w, id) pairs, respectively.

Recall that to build an encrypted index for a new file f, we first parse the file for distinct keywords $\{w\}$. Then, together with the unique identifier id for the file, each (w, id) will be encrypted to generate a ciphertext in the file index δ_f. First, we evaluate the time of building an add token of a file with 300,000 distinct keyword, which is the same scale as in [15], and it takes less than 7 seconds to complete the add token construction. We further explore to scale the number of (w, id) pairs to create an encrypted database with multiple files. Fig. 7(a) plots the processing latency of building an add token, with varying number of (w, id) pairs. From the figure, we can observe that the processing latency grows as the (w, id) pairs to be processed increases. But we observe that the processing time of creating an encrypted database with one million of (w, id) pairs in our design is less than 25 seconds, which is practical to use and shows visible advantage over an basic inverted index based construction [8] (e.g., $\sim$ 50 times over the basic construction in [8] with the same testbed environment defined before). Note that although the encrypted database can be built with multiple files, the built encrypted files are uploaded one by one to the contracted service peer, i.e., one file's add token at each file addition process.

But adopting the above file index structure (in contrast to the inverted index like in [8]) to construct an encrypted database comes at the cost of increased search efficiency. Recall that if the queried keyword has not been searched before, retrieving related results requires a linear scan through the encrypted database, which also reflects the major operation (re-executing the search operation) latency of a fair judgment process. But if a searched keyword is given from the client, the result retrieval latency is optimal since we can readily locate the result in one specific entry of the search index ζ_w and the digest index T_{peer}. To evaluate the search performance, we choose three different scales of datasets and follow the idea of [15] to set up measurement points to record the mean search time for each 5,000 randomly chosen search tokens. But unlike experiments in [15], the query keyword chosen in our evaluation for is not weighted with any probability distribution, i.e., eliminating the query keyword frequencies, to show more general results. Fig. 7(b) plots the average search latency per 5000 queries when we have performed 5,000, 10,000, 15,000, 20,000, and 25,000 independent queries, with three different datasets, i.e., dataset each with 400,000, 700,000, and 1,000,000 (w, id) pairs. From the figure, we can learn that the average search latency per 5,000 queries scales down rapidly as more queries are performed before and indexed in the search index ζ_w, and thus can be retrieved optimally. We observe that after 25,000

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2877332, IEEE Transactions on Dependable and Secure Computing

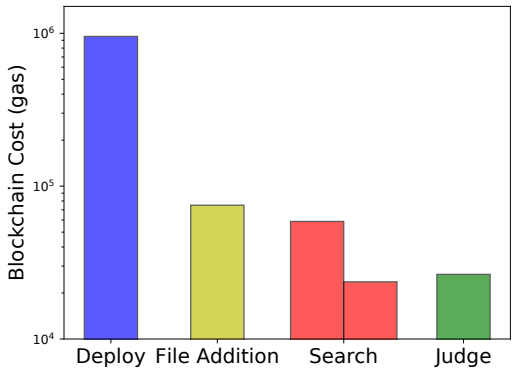


Fig. 8: Contract cost evaluation. The costs are presented by the gas usage in Ethereum, and the capital cost of contract deployment, file addition, search, and judge operation are \$0.76, \$0.59, \$0.64 and \$0.15 US dollar respectively, with an exchange rate of 1 *ether* = USD\$ 410 at the time of writing.

Number of (w, id) pairs	EDB size (MB)
400,000	20.99
700,000	36.01
1,000,000	51.46

TABLE 3: The built encrypted database size with three different number of (w, id) pairs.

of randomly selected queries, the average search latency is around 460ms in the smallest EDB, and it is expected that the search latency will become optimal in the long run when all independent keywords are queried and indexed. Note that we can also learn the operation latency of an arbiter node when re-executing a search operation from Fig. 7(b), and we can observe that linear scanning the largest data set (i.e., with 1,000,000 (w, id) pairs) takes around 14 seconds. We argue that this operation latency is acceptable because fair judgment is only executed when a client issues a judgment request, which we expect will not be frequently happened in real practice.

We also evaluate update cost of the search index ζ_w, digest index T_{peer}, the client's local checklist T_{client} when adding a new file. From the evaluation, we observe that the processing speed of the set hash function is efficient (around few milliseconds) [51], and with a large prime number less than $2^{32} - 1$, the size of a set hash digest is around 4 bytes. Thus even when we use all distinct keywords as in [50], the storage overhead for the local checklist on the client side is affordable (i.e., around 39MB).

6.2 Contract Construction and Evaluation

The smart contract in Ω_{fair} serves three purposes. First, it records the addresses of the client and the targeted service peer, the service information (e.g., service duration and fees), and the arbiter shard. Second, it stores the deposited cryptocurrenies and automatically triggers a t-locked payment for each search operation. Third, it records the halting votes sent by the arbiter nodes, and automatically halts the t-locked payment if there is enough votes (i.e., exceeding a pre-defined threshold).

Note that in Ethereum, every participant is represented by an address which is generated from the participant's public key at the setup stage. Therefore we record the addresses of the client, service peer, and arbiter nodes on the contract, so that the contract can automatically handle access authentication and money transferring. Also, the service fees for each search operation is recorded, and the duration of service is defined using the block number as a reference. To achieve the t-lock payment, we integrate a scheduled function call into our payment function for each search operation, such that the public Ethereum event *Transfer* will be executed after the lock-time t to finalize the payment. In particular, our prototype implementation employs an alarm clock service [52] that specifically targets this scheduling function call requirement of any Ethereum contract. During time t, the arbiter shard on the contract can jointly trigger the judge function on the blockchain to alter a flag tag, so that the payment finalization process will be automatically threw when the scheduled call triggers at time t. Note that on the Ethereum blockchain, the cost is evaluated in *gas*, which is associated with a *gas price* measured by the Ethereum currency named *ether*. In addition, Ethereum defines transaction cost and execution cost to measure storage and computation cost in the Ethereum state machine respectively.

Fig. 8 presents the cost evaluations of our implemented contract on Ethereum. For a file addition process, $(h', \mathbb{H}(\delta_f))$ tuples sent by two add transactions ($\mathbf{trans}^{\mathcal{C}}_{add}$ and $\mathbf{trans}^{\mathcal{S}}_{add}$) will incur storage cost on the blockchain. For a search operation, the contract triggers a t-time locked schedule call and freezes a pre-defined amount of money. It results in the first part of the blockchain cost elaborated in the figure. Then, if no judgment is requested by the client, or the arbiter shard judges that no further compensation is required, the payment is finalized by the public event *Transfer* when the scheduled call triggers at time t. It is represented by the second part of the search cost evaluation. The overhead of function judge evaluates the computation costs when the payment is revoked by the arbiter shard, which is triggered by a consensus voting process if the halting votes from the arbiter nodes exceed the pre-defined threshold $N \cdot \gamma$, where N is the arbiter shard size and γ is the threshold parameter. In addition to gas evaluation on the Ethereum blockchain, we further quantify the storage overhead for each transaction to handle, as shown in Table 4. We are aware that blockchain is an append-only structure, thus although a file can be deleted later by the client, the corresponding metadata anchored on the blockchain will not be erased. In future work, we will explore the leverage off-chain storage, e.g., [33], to further minimize the unnecessary storage overhead (when a file is deleted) on the blockchain.

Regarding the Ethereum blockchain's processing latency, it takes an average of 15 seconds for each transaction to be mined into the blockchain [14]. It indicates that there are delays when anchoring the metadata for a new file and issuing a new keyword trapdoor for query. We argue that this block confirmation latency is acceptable. In fact, as long as enough *transaction fees* [32] is included in transactions, the broadcast transactions will be mined and processed on the

Transactions	Blockchain storage cost
search token	20 bytes
result metadata	24 bytes
file addition	36 bytes

TABLE 4: The content storage cost for one transaction. A result metadata includes one HMAC digest τ_w of 20 bytes and and one set hash digest h_m of 4 bytes, with an overall estimated cost of 24 bytes. An add transaction for file addition includes one SHA-256 hash digest of 256 bits and one set hash digest, with an overall estimated cost of 36 bytes.

contract.

7 Discussions

Achieving Forward Security. The recently proposed file-injection attacks in dynamic SSE schemes highlight the need for forward security [23], [53]. In those attacks, the contents of past search queries can be revealed by injecting a few files. To preserve forward security, the information that the newly added files matching a past search query should be hided until a new search token is issued. A recent work [23] achieves forward security by using a trapdoor permutation function that makes the search token unlinkable to the add tokens. Here, following the similar architecture, our proposed scheme Ω_{fair} can be slightly revised to achieve forward security. Specifically, when building an add token for a new file, the reference list **x** as in Step 4e) of Fig. 2 should be removed. Then, we can use a trapdoor permutation π to construct a new search token $\tau^c_{w_i}$ as $\tau^c_{w_i} = \pi^{-1}_{usk}(\tau^{c-1}_{w_i})$, where $\tau^{c-1}_{w_i}$ is the current state of the search token constructed for keyword w_i and usk is the trapdoor permutation's secret key, and update the search token state for keyword w_i accordingly. The subsequent add token construction is the same as in Fig. 2. To perform search on a keyword w_p, the client will send the latest search token state for w_p, e.g., $\tau^c_{w_p}$ and the counter c to the contracted service peer. With the public key upk and c, the service peer can compute $\tau^{c-1}_{w_p} = \pi_{upk}(\tau^c_{w_p})$ to generate all related search tokens $\{\tau^i_{w_p}\}^c_{i=0}$, and use all of them to perform search operation as in Fig. 3. Note that if a search token generated from the trapdoor permutation is searched before, it can still be optimally retrieved from the search index ζ_w. We leave a detailed construction as our future work.

8 Conclusion

In this paper, we aim to bring secure and reliable content search over the encrypted files stored in emerging decentralized storage services. First, we study the problem of how to employ existing searchable encryption in decentralized storage, and propose to minimize the overhead by preserving file and index locality. Second, compared to a centralized setting, this new paradigm faces more severe threats, where both the client and service peer can be incentivized to conduct malicious behaviors. Hence, we need to bring fair payment for the query service, and we seek solutions from verifiable searchable encryption techniques and blockchain to ensure that the service peers will get the payment if and only if they faithfully conduct correct service. The proposed encrypted search functionality and fair payment mechanism are implemented using Python and on Ethereum smart contract, and the experiment results indicate the efficiency of our encrypted search service in the long run and the viability of our fairness design.

References

[1] C. Cai, X. Yuan, and C. Wang, "Towards trustworthy and private keyword search in encrypted decentralized storage," in *Proc. of ICC*, 2017.

[2] The Storj Project. On line at: https://storj.io/storj.pdf.

[3] The Sia Project. On line at: http://www.sia.tech.

[4] The Filecoin Project, "Filecoin." On line at: http://filecoin.io/filecoin.pdf.

[5] C. Gray, "Storj vs. dropbox: Why decentralized storage is the future." On line at: https://bitcoinmagazine.com/articles/storj-vs-dropbox-decentralized-storage-future-1408177107, 2014.

[6] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proc. of ACM CCS*, 2006.

[7] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. of ACM CCS*, 2012.

[8] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very large databases: Data structures and implementation," in *Proc. of NDSS*, 2014.

[9] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *Proc. of ACM/IFIP/USENIX Middleware*, 2003.

[10] P. Golle and I. Mironov, "Uncheatable distributed computations," in *Proc. of CT-RSA*, 2001.

[11] W. Du, J. Jia, M. Mangal, and M. Murugesan, "Uncheatable grid computing," in *Proc. of IEEE ICDCS*, 2004.

[12] R. Bost, P. Fouque, and D. Pointcheval, "Verifiable dynamic symmetric searchable encryption: Optimality and forward security," *IACR Cryptology ePrint Archive*, vol. 2016, p. 62, 2016.

[13] F. Chen, T. Xiang, X. Fu, and W. Yu, "Towards verifiable file search on the cloud," in *Proc. of IEEE CNS*, 2014.

[14] The Ethereum Project. On line at: https://ethereum.org.

[15] F. Hahn and F. Kerschbaum, "Searchable encryption with secure and efficient updates," in *Proc. of ACM CCS*, 2014.

[16] F. Tramèr, F. Zhang, H. Lin, J. Hubaux, A. Juels, and E. Shi, "Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge," in *Proc. of IEEE EuroS&P*, 2017.

[17] Y. Lu, Q. Tang, and G. Wang, "Zebralancer: Private and anonymous crowdsourcing system atop open blockchain," in *Proc. of IEEE ICDCS*, 2018.

[18] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Proc. of IEEE S&P*, 2016.

[19] D. C. Sánchez, "Raziel: Private and verifiable smart contracts on blockchains," *IACR Cryptology ePrint Archive*, 2017.

[20] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure multiparty computations on bitcoin," in *Proc. of IEEE S&P*, 2014.

[21] E. Kokoris-Kogias, E. C. Alp, S. D. Siby, N. Gailly, P. Jovanovic, L. Gasser, and B. Ford, "Hidden in plain sight: Storing and managing secrets on a public ledger," *IACR Cryptology ePrint Archive*, 2018.

[22] F. Hahn and F. Kerschbaum, "Searchable encryption with secure and efficient updates," in *Proc. of ACM CCS*, 2014.

[23] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *Proc. of ACM CCS*, 2016.

[24] Q. Wang, M. He, M. Du, S. S. M. Chow, R. W. F. Lai, and Q. Zou, "Searchable encryption over feature-rich data," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, pp. 496–510, 2018.

[25] J. Zhu, Q. Li, C. Wang, X. Yuan, Q. Wang, and K. Ren, "Enabling generic, verifiable, and secure data search in cloud services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, pp. 1721–1735, 2018.

[26] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Proc. of CRYPTO*, 2013.

[27] S. Kamara and T. Moataz, "Boolean searchable symmetric encryption with worst-case sub-linear complexity," in *Proc. of EUROCRYPT*, 2017.

[28] H. Li, Y. Yang, T. H. Luan, X. Liang, L. Zhou, and X. S. Shen, "Enabling fine-grained multi-keyword search supporting classified sub-dictionaries over encrypted cloud data," *IEEE Trans. Dependable Sec. Comput.*, vol. 13, no. 3, pp. 312–325, 2016.

[29] K. N. Xianrui Meng, Seny Kamara and G. Kollios, "Grecs: graph encryption for approximate shortest distance queries," in *Proc. of ACM CCS*, 2015.

[30] R. Cheng, J. Yan, C. Guan, F. Zhang, and K. Ren, "Verifiable searchable symmetric encryption from indistinguishability obfuscation," in *Proc. of ACM AsiaCCS*, 2015.

[31] D. E. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh, "Incremental multiset hash functions and their application to memory integrity checking," in *Proc. of ACM AsiaCCS*, 2003.

[32] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "Sok: Research perspectives and challenges for bitcoin and cryptocurrencies," in *Proc. of IEEE S&P*, 2015.

[33] M. Ali, J. C. Nelson, R. Shea, and M. J. Freedman, "Blockstack: A global naming and storage system secured by blockchains," in *Proc. of USENIX ATC*, 2016.

[34] The Skuchain Project. On line at: https://www.skuchain.com.

[35] The MedRec Project. On line at: https://www.pubpub.org/pub/medrec.

[36] The Bitcoin Project. On line at: https://bitcoin.org/en/.

[37] I. Bentov and R. Kumaresan, "How to use bitcoin to design fair protocols," in *Proc. of CRYPTO*, 2014.

[38] R. Kumaresan and I. Bentov, "Amortizing secure computation with penalties," in *Proc. of ACM CCS*, 2016.

[39] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Proc. of IEEE S&P*, 2016.

[40] The Maidsafe Project. On line at: https://maidsafe.net.

[41] S. Kamara, "Encrypted search," *ACM Crossroads*, vol. 21, no. 3, pp. 30–34, 2015.

[42] M. Mimoso, "Few victims reporting ransomware attacks to fbi." On line at: https://threatpost.com/few-victims-reporting-ransomware-attacks-to-fbi/126510/, 2017.

[43] J. I. Wong, "A coding error led to $30 million in ethereum being stolen." On line at: https://qz.com/1034321/ethereum-hack-a-coding-error-led-to-30-million-in-ethereum-being-stolen/.

[44] M. Gubik, "Proof of stake faq." On line at: https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ.

[45] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proc. of SOSP*, 2017.

[46] P. R. Zimmermann, *The Official PGP User's Guide*. Cambridge, MA, USA: MIT Press, 1995.

[47] E. Cecchetti, F. Zhang, Y. Ji, A. E. Kosba, A. Juels, and E. Shi, "Solidus: Confidential distributed ledger transactions via PVORM," in *Proc. of ACM CCS*, 2017.

[48] FIPS 180-3, Secure Hash Standard (SHS). Federal Information Processing Standards Publications (FIPS), October, 2008.

[49] Ethereumjs-testrpc. Online at: https://www.npmjs.com/package/ethereumjs-testrpc.

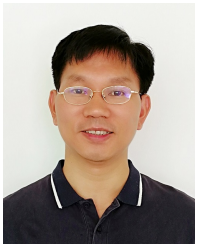
[50] 10,000 most common English words. Online at: Onlineat:https://github.com/first20hours/google-10000-english.

[51] D. E. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas, "Towards constant bandwidth overhead integrity checking of untrusted data," in *Proc. of IEEE S&P*, 2005.

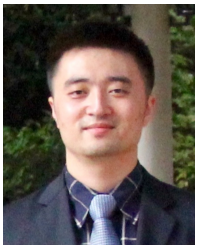
[52] Ethereum Alarm Clock. On line at: http://www.ethereum-alarm-clock.com/.

[53] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *Proc. of ACM CCS*, 2017.

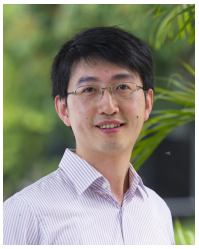
Chengjun Cai Chengjun Cai is a first year Ph.D. student in City University of Hong Kong. He received his B.S. degree in Computer Science and Technology from Jinan University in 2016. He was a research assistant in City University of Hong Kong. His research interests include distributed system security and privacy-enhancing technologies.



Jian Weng Jian Weng is a professor and the Executive Dean with College of Information Science and Technology in Jinan University. He received B.S. degree and M.S. degree at South China University of Technology in 2001 and 2004 respectively, and Ph.D. degree at Shanghai Jiao Tong University in 2008. His research areas include public key cryptography, cloud security, blockchain, etc. He has published 80 papers in international conferences and journals such as CRYPTO, EUROCRYPT, ASIACRYPT, TCC, PKC, CT-RSA, IEEE TDSC, etc. He also serves as associate editor of IEEE Transactions on Vehicular Technology.



Xingliang Yuan Xingliang Yuan is a Lecturer at the the Faculty of Information Technology, Monash University, Australia. He received his B.S. degree from Nanjing University of Posts and Telecommunications in 2008, the M.S. degree from Illinois Institute of Technology in 2009, both in Electrical Engineering, and the Ph.D. degree in Computer Science from City University of Hong Kong in 2016. His research interests include cloud computing security, secure networked systems, and hardware security.



Cong Wang Cong Wang has been an Associate Professor at the Department of Computer Science, City University of Hong Kong, since the Summer of 2012. He received his PhD in the Electrical and Computer Engineering from Illinois Institute of Technology, USA, M.Eng in Communication and Information System, and B.Eng in Electronic Information Engineering, both from Wuhan University, China. His current research interests include data and computation outsourcing security in the context of cloud computing, network security in emerging Internet architecture, multimedia security and its applications, and privacy-enhancing technologies in the context of big data and IoT. He has published frequently in peer-reviewed journals and conferences, including IEEE JSAC, IEEE TIFS, IEEE TMM, IEEE INFOCOM, ACM Multimedia, AsiaCCS, etc. His H-index is 25, and his total citation has exceeded 12,800, according to Google Scholar (as of Sep. 2017).

He has been one of the Founding Members of the Young Academy of Sciences of Hong Kong since September 2017. He received the President's Awards, City University of Hong Kong in January 2017. He is a co-recipient of the Best Student Paper Award of IEEE ICDCS 2017, the Best Paper Award of IEEE MSN 2015 and CHINACOM 2009. His research has been supported by multiple government research fund agencies, including National Natural Science Foundation of China, Hong Kong Research Grants Council, and Hong Kong Innovation and Technology Commission. He has been serving as the TPC co-chairs for a number of IEEE conferences/workshops. He is a member of IEEE and ACM.