2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00091

Authenticated Keyword Search in Scalable Hybrid-Storage Blockchains

Ce Zhang[†], Cheng Xu[†], Haixin Wang, Jianliang Xu, Byron Choi

Department of Computer Science, Hong Kong Baptist University, Hong Kong
{cezhang, chengxu, hxwang, xujl, bchoi}@comp.hkbu.edu.hk

***Abstract*—Blockchain has emerged as a promising solution for secure data storage and retrieval for decentralized applications. To scale blockchain systems, a prevailing approach is to employ a hybrid storage model, where only small meta-data are stored on-chain while the raw data are outsourced to an off-chain storage service provider. The key issue for query processing in such a system is the design of *gas*-efficient authenticated data structure (ADS) to authenticate the query results. In this paper, we study novel ADS schemes for authenticated keyword search in hybrid-storage blockchains. We first propose the Suppressed Merkle inverted (Merkle^{inv}) index, which maintains only a partial ADS structure on-chain that can be securely updated with a logarithm-sized cryptographic proof. Moreover, we propose a Chameleon inverted (Chameleon^{inv}) index that leverages the chameleon vector commitment to achieve a constant maintenance cost. It is further optimized with Bloom filters to enhance the query and verification performance. We prove the security of the proposed ADS schemes and evaluate their performance using real datasets on the Ethereum platform. Experimental results show that, compared to a baseline solution, the proposed Merkle^{inv} and Chameleon^{inv} indexes reduce the average on-chain maintenance cost from US$10.39 down to US$2.50 and US$0.24, respectively, without sacrificing much the query performance.**

I. Introduction

Blockchain is a distributed ledger collectively maintained by a network of mutually untrusted nodes. With the hash chain technique and the consensus protocol, the data stored in a blockchain are ensured to be immutable and tamper-resistant [1]. As such, blockchain has been considered a promising solution for secure data storage and retrieval, also known as *e-notarization*, in many decentralized applications, such as IoT, healthcare, supply chain, and legal document management [2].

Despite of its evident advantages, blockchain is known to have limited storage scalability since data needs to be replicated across the entire network. Storing raw data, such as documents, images, and PDF files, directly into the blockchain would increase the storage burden, leading to longer transaction delay and lower system throughput [3]. To tackle this issue, a common approach is combining the blockchain with off-chain storage to form a hybrid-storage blockchain [4], [5].[1] As shown in Fig. 1, the raw data are sent to an off-chain storage service provider (SP) for management, while their cryptographic hashes are stored on-chain to uphold data integrity. To support integrity-assured data retrieval in such a hybrid-storage blockchain, authenticated query processing can be applied. The basic idea is to let both the smart contract and the off-chain SP maintain an authenticated data structure (ADS). For each query, the SP computes not only the query results but also a cryptographic proof known as verification object (VO) to attest to those results. With the VO from the SP and the authenticated information retrieved from the blockchain, the client can verify the result integrity.

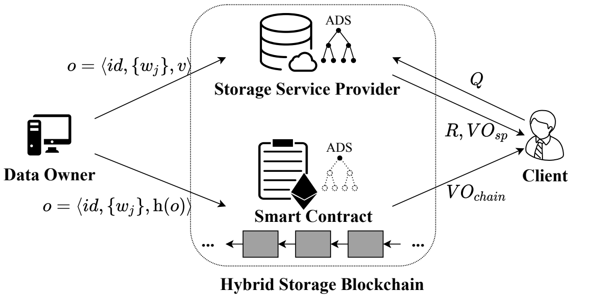
ADS
Storage Service Provider
$o = \langle id, \{w_j\}, v\rangle$
Data Owner
$o = \langle id, \{w_j\}, \mathrm{h}(o)\rangle$
ADS
Smart Contract
...
...
Hybrid Storage Blockchain
Q
R, VO_{sp}
VO_{chain}
Client

Fig. 1: Architecture of a Hybrid-Storage Blockchain

Unlike authenticated query processing in traditional outsourced databases that mainly focuses on query efficiency [7]–[11], a key challenge here is that the ADS should be designed to be *update efficient*. In other words, the ADS should be able to be efficiently maintained by the smart contract, in terms of both computation and storage costs. Ethereum [12], a pioneering smart contract platform, employs a unique *gas* model to measure the smart contract's execution cost. Table I shows the gas costs of the major smart contract operations. As can be seen, the costs incurred for different operations differ dramatically, with the operation of storing data into the blockchain state being the most expensive one. In a pioneering study [13], a gas-efficient ADS named GEM^2-tree has been proposed. However, the GEM^2-tree has two limitations. First, it is developed for range queries and cannot be directly used to support other more complex queries such as keyword search and similarity search. Second, as we will elaborate later on, it still incurs a relatively high gas cost due to storing many intermediate data in the blockchain state.

Motivated by the popularity of keyword search in data retrieval, in this paper, we study novel ADS schemes for authenticated keyword search in hybrid-storage blockchains. We start by introducing a baseline ADS called Merkle inverted (Merkle^{inv}) index. The Merkle^{inv} index is an inverted index in which each keyword corresponds to a Merkle B-tree (MB-tree) [7] that indexes the corresponding object IDs. With the

[†] These authors have contributed equally to this work.

[1] Another advantage is that the data stored off-chain can be easily deleted when needed, which complies with the "right to be forgotten" under the EU General Data Protection Regulation (GDPR) [6].

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00091

index, a keyword search represented by a Boolean expression in the disjunctive normal form (DNF) (e.g., ("COVID-19"$\wedge$"Vaccine")$\vee$("SARS-CoV-2"$\wedge$"Vaccine")) is seen as the union of the results from each conjunctive component. Each conjunctive component can then be processed as authenticated join queries over the query keywords' MB-trees. However, maintaining the complete Merkle^{inv} index on-chain by the smart contract is prohibitively expensive due to the excessive storage operations.

To address this issue, we first propose the Suppressed Merkle^{inv} index that can reduce the smart contract maintenance cost without compromising the query efficiency. The main idea is that the smart contract maintains only the root digest of each MB-tree in the Merkle^{inv} index. This comes with the observation that for the on-chain ADS, only the root digests are used during the authenticated keyword search. To update the root digests upon arrival of a new object, however, the SP needs to send an *update proof*, consisting of the authenticated information to securely compute the new root digests, to the smart contract. The maintenance of the Suppressed Merkle^{inv} index requires constant (costly) storage operations and logarithmic (cheap) memory operations, achieving an overall lower gas cost.

We are not stopped here. To further reduce the ADS maintenance cost, we propose the Chameleon inverted (Chameleon^{inv}) index that leverages the chameleon vector commitment (CVC) to achieve a constant maintenance cost. Using the CVC, one can publicly verify the data stored in a vector. Moreover, the data owner can update the vector without changing its commitment by using a secret trapdoor. With these properties, a Chameleon tree can be built with an invariant root commitment. In the Chameleon^{inv} index, each keyword corresponds to a Chameleon tree. The smart contract maintains the root commitments with some auxiliary data of the Chameleon^{inv} index and the SP maintains the complete index for efficient query processing. In addition, to enhance the query and verification performance, we propose an optimized Chameleon^{inv*} index, in which the smart contract maintains an additional Bloom filter for every group of objects in each Chameleon tree. The Bloom filters help to facilitate the testing of non-existing objects, thus achieving better query and verification performance. On the other hand, due to the additional maintenance of the Bloom filters, the Chameleon^{inv*} index has a slightly higher maintenance cost. Table II summarizes the performance comparison of the aforementioned ADS schemes. Clearly, our proposed ADS schemes obtain a lower maintenance cost while retaining the same level of query and verification performance.

To summarize, this paper's contributions are as follows:

- For the first time in the literature, we study the problem of authenticated keyword search in scalable hybrid-storage blockchains.
- We propose the Suppressed Merkle^{inv} index that can significantly reduce the ADS maintenance cost in terms of the smart contract's gas consumption.
- We develop the Chameleon^{inv} index to further reduce the ADS maintenance cost to a constant level while still supporting efficient queries. To enhance the query and verification performance, an optimized Chameleon^{inv*} index is also proposed.

TABLE I: Ethereum Gas Cost (A full list is available in [12])

Operation	Gas Used	Cost in US$[2]	Explanation
C_{sload}	200	6.87×10^{-4}	load a word from storage
C_{sstore}	20,000	6.87×10^{-2}	save a word to storage
$C_{supdate}$	5,000	1.72×10^{-2}	update a word to storage
C_{mem}	3	1.03×10^{-5}	access a word in memory
C_{hash}	$30 + 6 \cdot x$	$(1.03 + 0.21 \cdot x) \times 10^{-4}$	hash a x-word message
C_{tx}	$21,000$	7.21×10^{-2}	execute a transaction
C_{txdata}	68	2.34×10^{-4}	transact a byte of data

TABLE II: Performance Comparison of ADS Schemes

ADS	Maintenance Gas Cost	Authenticated Keyword Search Cost	
		Query Process	Result Verification
Merkle^{inv}	$O(L \cdot C_1 \log n)$	$O(k \log n)$	$O(k \log n)$
Suppressed Merkle^{inv}	$O(L \cdot C_1 + L \cdot C_2 \log n)$	$O(k \log n)$	$O(k \log n)$
Chameleon^{inv}	$O(L \cdot C_1)$	$O(k \log n)$	$O(k \log n)$
Chameleon^{inv*}	$O(L \cdot C_1)$	$O(k' \log n)$	$O(k' \log n)$

Note: L is the maximum number of keywords of an object; n is the total number of objects; C_1 and C_2 denote the costs of storage operations (e.g., C_{sload}, C_{sstore}, $C_{supdate}$) and memory operations (e.g., C_{mem}, C_{hash}, C_{txdata}), respectively, and $C_1 > C_2$; k denotes the total number of matching and boundary objects for the authenticated join processing, k' is the optimized number of k with the help of Bloom filters, and $k' < k$.

- We conduct theoretical analysis and empirical evaluation to validate the proposed ADS schemes. The experiments using real datasets on the Ethereum platform show that, without sacrificing much the query performance, the proposed ADS schemes reduce up to 76% and 98% of the gas cost as compared to a baseline solution.

The rest of the paper is organized as follows. Section II gives the blockchain background and the problem formulation, which is followed by some preliminaries in Section III. Section IV and Section V present our Suppressed Merkle^{inv} index and Chameleon^{inv} index, respectively. Section VI analyzes the security of the proposed schemes. Section VII reports the experimental results. Section VIII surveys the related works. Finally, we conclude our paper in Section IX.

II. Background and Problem Formulation

In this section, we first provide some basic knowledge about blockchain technology. Then, we present the problem formulation in three aspects: system model, threat model, and problem statement.

A. Blockchain and Smart Contract

Blockchain is an append-only data structure consisting of a series of blocks. To ensure its immutability and tamper-resistance, each block is chained by a cryptographic hash pointer (as shown in Fig. 2). Furthermore, there is a consensus proof embedded in each block to ensure that the entire network keeps the same copy of the blocks. For example, in the Proof of Work (PoW) consensus [1], a nonce computed by solving some cryptographic hash puzzle is used as the consensus proof. While first-generation blockchains (e.g., Bitcoin [1]) are designed specifically for cryptocurrencies, second-generation blockchains (e.g., Ethereum [12]) introduce the concept of

[2] The cost in US$ is calculated based on an average gas price of 15 Gwei and the Ether price of US$229 at of June 15, 2020.

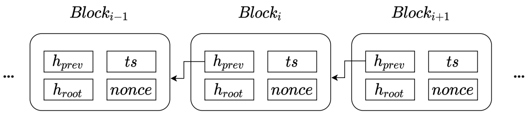

Fig. 2: Blockchain Header Structure

smart contract. A smart contract is a user-defined Turing-complete program executed by the blockchain virtual machines. The program can interact with the data stored in the blockchain with integrity assurance, which is guaranteed by the blockchain consensus protocol. Users can deploy and execute a smart contract by sending a transaction to the blockchain network. Since the blockchain network needs to spend computation and storage resources for the smart contract deployment and execution, a transaction fee, denominated in *gas*, is charged for each deployment and execution. As shown in Table I, in general, the storage operations are a multitude more expensive than the in-memory data access and computation operations. In order to prevent computation-intensive and non-stop smart contracts, a *gasLimit* (e.g., 8,000,000 in [12]) is set. The deployment or execution of a smart contract will be terminated if the total gas consumption exceeds the gasLimit.

B. System Model

As shown in Fig. 1, our system involves four parties: a data owner (DO), a blockchain with smart contract functionality, an off-chain storage service provider (SP), and query clients. The blockchain itself and the SP are components of the hybrid-storage blockchain. Each data object is modeled as a tuple $o_i = \langle id, \{w_j\}, v\rangle$, where id denotes the object's ID, $\{w_j\}$ $(1 \leq j \leq m)$ denotes a set of keywords associated with the object, and v is the content of the object. We assume that the data objects are appended to the blockchain in a streaming way. As such, the total number of objects is unbounded. Moreover, once stored in the system, the objects are immutable. Upon arrival of a new object o_i, the DO sends o_i to the SP but only the meta-data $\langle id, \{w_j\}, \mathsf{h}(o_i)\rangle$ to the blockchain, where $\mathsf{h}(\cdot)$ is a cryptographic hash function. In this way, we reduce the on-chain storage cost while ensuring the data integrity.

In order to support authenticated query processing under the hybrid-storage blockchain, an *authenticated data structure* (ADS) is maintained by both the smart contract and the SP. During the ADS maintenance, the DO invokes the smart contract to update the on-chain ADS. Meanwhile, the ADS in the SP is updated synchronously with the latest confirmed one on the blockchain. The digests of the ADS become the *authenticated information* that is shared by both the smart contract and the SP.

In this paper, we focus on keyword search queries. The query input is a monotone Boolean expression over the queried keywords. For simplicity, we assume that it is expressed in the disjunctive normal form (DNF) as $Q = q_1 \vee q_2 \vee \cdots \vee q_n$, where $q_i = w_1 \wedge w_2 \wedge \cdots \wedge w_l$. The client wants to retrieve every data object o_i that satisfies the query condition, i.e., $R = \{o_i = \langle id, \{w_j\}, v\rangle \mid Q(\{w_j\}) = true\}$.

The authenticated query processing works as follows. The client sends a keyword search request to the SP, which uses the ADS to compute the results as well as a verification object (VO_{sp}) that can be used by the client to verify the results. Both the results and VO_{sp} are returned to the client. During the result verification, the client first retrieves the authenticated digests (VO_{chain}) from the blockchain. Then, the client can verify the correctness of the results by combining VO_{chain} and VO_{sp}.

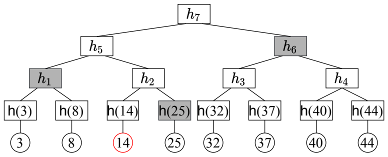

Fig. 3: Merkle Hash Tree

C. Threat Model

In our system, the off-chain SP is not fully trusted and can behave arbitrarily. It may add, delete, or modify the query results intentionally or unintentionally. To ensure the integrity of the query results, the client needs to check the following two security criteria:

- **Soundness**: All of the returned results satisfy the query condition and are originated from the DO;
- **Completeness**: No valid result is missing.

Regarding the blockchain, we assume that the adversary cannot gain any advantage in attacking the consensus protocol and that the execution integrity of the smart contract is guaranteed.

D. Problem Statement

With the above system model and threat model, the problem we study in this paper is how to design the ADS that can be efficiently maintained by the on-chain smart contract, in terms of the gas cost, while effectively supporting authenticated keyword search in hybrid-storage blockchains. In the following, we provide some preliminaries in Section III and then present a baseline solution, followed by two gas-efficient ADS schemes in Section IV and Section V.

III. Preliminaries

In this section, we introduce some cryptographic primitives and the authenticated query processing method on which our proposed ADS schemes build.

A. Cryptographic Primitives

Cryptographic Hash Function: A cryptographic hash function maps an arbitrary-length message m to a fixed-sized digest $\mathsf{h}(m)$. It has two important properties: (i) collision resistance, i.e., it is hard to find two different messages m_1 and m_2 such that $\mathsf{h}(m_1) = \mathsf{h}(m_2)$; (ii) irreversibility, i.e., given a digest h, it is hard to find a message m such that $\mathsf{h}(m) = h$.

Merkle Hash Tree (MHT): An MHT is a data structure for data authentication and verification [14]. Fig. 3 shows an example of MHT with eight data objects. Each leaf node stores the digest of the indexed object. Each internal node stores a hash that is derived from its two child nodes, e.g., $h_5 = \mathsf{h}(h_1||h_2)$, where '$||$' denotes the concatenation operation. The root hash is signed by the data owner and made public. The MHT can be used to authenticate any subset of the data objects stored in the leaf nodes in logarithmic complexity. For example, to authenticate the object whose value is 14, a

Merkle proof consisting of $\{\mathsf{h}(25), h_1, h_6\}$ (shaded in Fig. 3) is returned for verification. A verifier can use the Merkle proof and the object's value 14 to reconstruct the root hash and compare it with the public signed root hash. If they match, it is proved that the object exists in the MHT and has not been tampered with.

The MHT is a binary tree. To support authenticated queries in relational databases, it has been extended to multi-way Merkle B-tree (MB-tree), which follows the B^+-tree structure and augments each index entry with a corresponding hash [7].

Vector Commitment (VC): A VC maps a vector of messages ($\langle m_1, m_2, \cdots, m_q \rangle$) to a fixed-sized commitment, which can be opened at a specific position (e.g., proving m_i is the i^{th} committed message) [15]. A VC scheme consists of several polynomial-time algorithms:

- $\mathsf{Gen}(1^\lambda, q)$: The key generation function returns the public parameters pp with the input of the security parameter λ and the vector size q.
- $\mathsf{Com}_{\mathsf{pp}}(\langle m_1, m_2, \cdots, m_q \rangle, r)$: The commitment function takes a vector of q messages and a random number r as input and outputs a commitment c with some auxiliary information aux.
- $\mathsf{Open}_{\mathsf{pp}}(i, m, \mathsf{aux})$: The opening function returns a proof π *iff* m is the i^{th} message in a vector w.r.t. c.
- $\mathsf{Ver}_{\mathsf{pp}}(c, i, m, \pi)$: The verification function returns 1 *iff* π passes the verification that c is computed based on a sequence of messages with m at position i.

Chameleon Vector Commitment (CVC): A CVC is a trapdoor vector commitment scheme [16]. A user who owns a private trapdoor can update a message in a vector without changing the vector's commitment. The key generation function $\mathsf{CGen}(1^\lambda, q)$ is slightly different from the $\mathsf{Gen}(1^\lambda, q)$ with an additional output of trapdoor td. Meanwhile, it supports an additional collision finding algorithm:

- $\mathsf{CCol}_{\mathsf{pp}}(c, i, m, m', \mathsf{td}, \mathsf{aux})$: The collision finding algorithm computes a new aux' such that aux' corresponds to a vector of q messages with m' instead of m at position i and the commitment remains to be c.

After finding the collision, the user gets the updated aux', which can be used as the input of $\mathsf{Open}_{\mathsf{pp}}(i, m', \mathsf{aux}')$ to compute a proof π'. Then, a verifier can use $\mathsf{Ver}_{\mathsf{pp}}(c, i, m', \pi')$ to verify that m' is the i^{th} message stored in c's vector.

B. Authenticated Join Processing with the MB-tree

The authenticated join processing aims to join multiple tables with result integrity assurance. Yang *et al.* [8] proposed a sort-merge-join like algorithm. Consider two tables, each indexed by an MB-tree, denoted by $\mathcal{T}_R$ and $\mathcal{T}_S$, respectively.[3] The join is processed in rounds. We use Fig. 4 as an example to illustrate the detailed procedure. We start from the tree $\mathcal{T}_R$ with its first object r_1 as a target. Its matching and boundary objects in the tree $\mathcal{T}_S$ are s_3 and s_4. In the next round, we switch the roles of $\mathcal{T}_S$ and $\mathcal{T}_R$. The target changes to s_4 and the boundary object r_2 is retrieved. The role switching continues until reaching the target r_5 with its left boundary s_{12}, which is the last leaf node of $\mathcal{T}_S$. For result verification, all of the matching and boundary objects, together with their Merkle proofs (shaded in Fig. 4), are added to the VO. During the verification, the client can reconstruct the root hash of the tree $\mathcal{T}_R$ by computing $\mathsf{h}(\mathsf{h}(\mathsf{h}(r_1)||\,\mathsf{h}(r_2))||\,\mathsf{h}(\mathsf{h}(r_3)||\,\mathsf{h}(r_4))||\,\mathsf{h}(\mathsf{h}(r_5)||h_{r_6}))$ and verify it against the signed one. This ensures the integrity of $\mathcal{T}_R$'s matching and boundary objects. The verification for the tree $\mathcal{T}_S$ is similar. After that, the client checks that each round's target is the previous round's right boundary object and also resides within its corresponding boundary, which ensures that no valid result is missing. Specifically, the target r_1 in $\mathcal{T}_R$ is checked against s_3 and s_4 in $\mathcal{T}_S$, and r_1 is found to be one of the join results. The right boundary s_4 is the target for the next round, which is checked against r_2 in $\mathcal{T}_R$. The checking terminates when we reach the left boundary s_{12} in $\mathcal{T}_S$ with the target r_5.

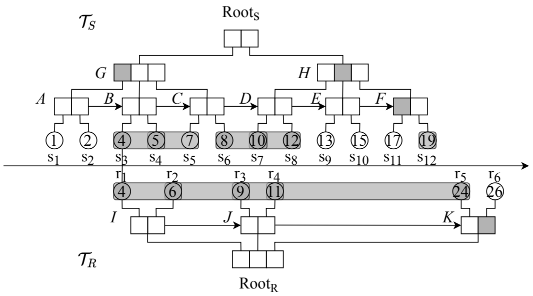

Fig. 4: Authenticated Join Processing of Two MB-trees

IV. Suppressed Merkle^{inv} Index

In this section, we first introduce the baseline solution, Merkle^{inv} index, and show that it suffers from a high on-chain maintenance cost. To tackle this issue, we propose a new ADS, called Suppressed Merkle^{inv} index. It can be efficiently maintained by the smart contract, while achieving the same query efficiency as the baseline solution.

A. Baseline Solution

To process keyword search, a commonly used data structure is the inverted index [17]. It consists of a dictionary of keywords, each pointing to a list of objects that contain the keyword. Fig. 5 shows an example of inverted list. In order to support authenticated keyword search, a baseline solution is to build an MB-tree for each keyword's object list. We call this ADS the Merkle inverted (Merkle^{inv}) index. In detail, the search key of each MB-tree is the object ID. Without loss of generality, we assume that the object ID is a monotonically increment attribute (e.g., using the transaction timestamp as ID). This ensures that the object list is ordered for optimizing the insertion and query processing. When a new object o_i is added to the Merkle^{inv} index, its object ID and hashed value $\langle o_i.id, \mathsf{h}(o_i) \rangle$ will be inserted to the MB-tree of each associated keyword. The Merkle^{inv} index is maintained by both the on-chain smart contract and the off-chain SP. Recall that a keyword search query is represented in DNF. The SP can process each conjunctive component as an authenticated join query over the keywords' MB-trees and combine the results of all conjunctive components. For ease of illustration, we will

[3] For join process with multiple trees, the two trees with the smallest sizes are joined first; then, the intermediate join results are used as targets to find the matching and boundary objects from other trees.

Keyword ID	Keyword w		Object List for w
1	COVID-19	$\mapsto$	1, 2, 4, 5, 7, 8, 10, 12, 13, 15, 17, 19
2	Symptom	$\mapsto$	4, 6, 9, 11, 24, 26
3	SARS-CoV-2	$\mapsto$	1, 3
4	Vaccine	$\mapsto$	4, 5, 8

Fig. 5: Inverted Index Example

only focus on how to process a conjunctive component of the query in the rest of the paper.

We use Fig. 4 and Fig. 5 to give an example of authenticated keyword search with the Merkle^{inv} index. The MB-trees $\mathcal{T}_S$ and $\mathcal{T}_R$ in Fig. 4 are built upon the object lists of keywords "COVID-19" and "Symptom" in Fig. 5. Consider a keyword search query Q = "COVID-19" $\wedge$ "Symptom". Using the algorithm discussed in Section III-B, the SP processes it as an authenticated join query over $\mathcal{T}_S$ and $\mathcal{T}_R$, and obtains the result r_1. It also generates a VO (denoted as VO_{sp}) for result verification. To verify the query results, the same process in Section III-B is applied except that the root hashes of $\mathcal{T}_S$ and $\mathcal{T}_R$ (denoted as VO_{chain}) are retrieved from the blockchain.[4]

Cost Analysis. To analyze the on-chain maintenance cost of the Merkle^{inv} index, we first consider the cost of adding a new object to a single keyword's MB-tree. We assume that the MB-tree's node capacity is the same as the granularity of blockchain data access. Suppose that the MB-tree has a fan-out of F and the current number of objects is n. An insertion to the MB-tree requires fetching the corresponding leaf node and storing the inserted object, which consume $O(\log_F n \cdot C_{sload} + C_{sstore})$ gas. Then, the hashes of the $\log_F n$ ancestor nodes are updated, each requiring $O(F \cdot C_{sload} + C_{hash} + C_{supdate})$ gas. Furthermore, in the worst case, there could be up to $\log_F n$ node splits during the insertion. For each node split, a new node will be created, followed by the key distribution and the update of the corresponding hash values. The creation of a new node costs $2C_{sstore}$ to store the node content and its hash value. The rest of the operations consumes $O(F \cdot C_{sload} + C_{supdate})$ gas. In total, the worst-case maintenance cost for a single keyword's MB-tree is as follows:

$$C_{\text{MI}}^{\text{insert}}(n) = \log_F n \times \Big(2C_{sstore} + 2C_{supdate} + (2F+1)C_{sload} + C_{hash}\Big) + C_{sstore}$$

For a new object with L keywords, the total cost is $L \cdot C_{\text{MI}}^{\text{insert}}(n)$. Apparently, the gas cost is logarithmic with respect to the number of the objects. Moreover, the coefficient of the logarithmic term contains the expensive storage operations, i.e., C_{sstore}, $C_{supdate}$, and C_{sload}. As such, maintaining the complete Merkle^{inv} index on-chain is inefficient in terms of the gas cost.

B. Overview of the Suppressed Merkle^{inv} Index

As shown in the previous section, only the on-chain root hashes are needed during the authenticated keyword search. It is redundant to maintain the complete MB-trees on-chain. Therefore, in this section, we propose the Suppressed Merkle^{inv} index to reduce the smart contract maintenance cost.

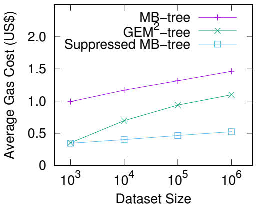

Fig. 6: Average Gas Cost for Index Maintenance (DBLP)

The general idea of the Suppressed Merkle^{inv} index is to *fully* suppress the on-chain MB-trees. That is, we store only the root hashes of the MB-trees to minimize the maintenance cost. The SP, on the other hand, still maintains the complete structures of the MB-trees to support efficient query processing. This idea is inspired by the GEM^2-tree proposed in [13], which maintains a series of suppressed MB-trees that are gracefully merged into a materialized MB-tree. However, as the materialized MB-tree is large and still needs to be maintained by the smart contract, the cost saving of GEM^2-tree is limited as shown in Fig. 6.[5] In contrast, by maintaining only the root hash on-chain, the suppressed MB-tree saves more maintenance costs and is thus applied to the Suppressed Merkle^{inv} index.

A key issue is how the smart contract can maintain the root hash of each MB-tree without knowing the complete structure. Our idea is to ask the off-chain SP to construct an *update proof*, denoted as $UpdVO$, during a new object's insertion. Consisting of enough authenticated information, the $UpdVO$ is then sent to the smart contract to securely update the root hashes. In the following, we give the details on how to construct the $UpdVO$ (by the SP) and how to use it to update the MB-trees' root hashes (by the smart contract).

C. Details of the Suppressed Merkle^{inv} Index

Without loss of generality, we consider the insertion operation on a single MB-tree in the inverted index. As the SP is not fully trusted, the $UpdVO$ should include the Merkle proof corresponding to the tree path of the leaf node to be updated. Since the object IDs are monotonically increment, the new insertion will always go to the right-most leaf node. Thus, the $UpdVO$ consists of the right-most branch of the MB-tree. Algorithm 1 describes the procedure of constructing the $UpdVO$ by the SP. It traverses the tree in a top-down fashion starting from the root node. For each internal node, the child hashes except the right-most one are appended to the $UpdVO$ (lines 2-7). Finally, when we reach the leaf node, the hashes of all existing objects and the new object are added to the $UpdVO$ (lines 8-13).

Example. *Consider Fig. 4 as an example. Suppose a new object s_{13} with id $= 23$ is going to be added to the MB-tree $\mathcal{T}_S$. The object will be inserted into the leaf node F. By invoking Algorithm 1, the $UpdVO$ computed by the SP consists of: (i) $\langle h_G \rangle$; (ii) $\langle h_D, h_E \rangle$; (iii) $\langle h_{s_{11}}, h_{s_{12}} \rangle$; (iv) $h_{s_{13}}$.*

Upon receiving the $UpdVO$ from the SP, the smart contract first verifies the hash of the new object with respect to the one sent by the DO. Next, the smart contract checks the other

[4] For non-existing keywords, their digests are empty in VO_{chain} and can be checked by the client for ensuring result completeness.

[5] The gas costs plotted in Fig. 6 were measured on the Ethereum platform under the default experiment settings specified in Section VII.

```
Algorithm 1 GenUpdVO(MB-tree $\mathcal{T}$, $o$) by SP
Input Keyword's MB-tree $\mathcal{T}$, new object $o$
Output Update proof $UpdVO$
$curnode \leftarrow \mathcal{T}.root$;
while $curnode$ is not a leaf do
    $UpdVO.append$("$\langle$");
    for $i$ in $[0, curnode.fanout - 2]$ do
        $UpdVO.append(h_{curnode.child[i]})$;
    $UpdVO.append$("$\rangle$"); $curnode \leftarrow curnode.child.last$;
if $curnode$ is a leaf then
    $UpdVO.append$("$\langle$");
    for each $h_o$ in $curnode$ do
        $UpdVO.append(h_o)$;
    $UpdVO.append$("$\rangle$"); $UpdVO.append(\mathsf{h}(o))$;
return $UpdVO$
```

update information sent by the SP. More specifically, a root hash is reconstructed from the hashes in the $UpdVO$ in a bottom-up manner and compared with the one stored on-chain. If they match, the integrity of the $UpdVO$ is proved.

After the verification of the $UpdVO$, the smart contract uses it to update the root hash of the MB-tree using Algorithm 2. Similar to the MB-tree's insertion, the leaf node's hash is firstly recomputed (lines 5-10), followed by recomputing the hashes of all the nodes in the Merkle path level by level up to the root (lines 11-23). Note that a node may be split due to overflow during the insertion. We use the in-memory variable h' to record the updated hash value and h'_1, h'_2 to record the two split nodes' hash values. If the current node is not split (lines 5-6 and lines 13-17), the updated node hash is stored in h'. Otherwise, the two split nodes' hashes are computed and stored in h'_1 and h'_2, respectively (lines 7-10 and lines 18-21). Finally, the updated root hash is stored on-chain by the smart contract. Once successful, the smart contract emits an event to indicate the completion of the update operation (lines 23-24). After receiving the event, the SP can use the updated ADS for future query processing.

Example. *We continue the example above to illustrate the UpdVO verification and the root hash update during the insertion. The smart contract first verifies the integrity of the UpdVO by recomputing the root hash* $\mathsf{h}(h_G||\,\mathsf{h}(h_D||h_E||\,\mathsf{h}(h_{s_{11}}||h_{s_{12}})))$ *and comparing it with the one stored on-chain. Also,* $h_{s_{13}}$ *is verified against the hash of* s_{13} *sent by the DO. Then, the smart contract proceeds to update the root hash for the new object* s_{13}*. We assume that the fan-out is 4. Since the current leaf node F is not full, it is not split. The node F's hash is updated to* $h'_F = \mathsf{h}(h_{s_{11}}||h_{s_{12}}||h_{s_{13}})$ *after the insertion of* s_{13}*. For the node H, its hash is updated to* $h'_H = \mathsf{h}(h_D||h_E||h'_F)$*. Finally, the root hash is updated to* $\mathsf{h}(h_G||h'_H)$*. Fig. 7 shows the updated MB-tree after the insertion of* s_{13}*, where the shaded parts indicate the updated hashes.*

Cost Analysis. We now estimate the smart contract maintenance cost of the Suppressed Merkle^{inv} index. We first consider the maintenance cost of updating the MB-tree for a single keyword. Assume that the keyword's MB-tree contains n objects and the fan-out is F. For the $UpdVO$,

```
Algorithm 2 Insert($UpdVO$) by Smart Contract
Input Update proof $UpdVO$
1: Verify $UpdVO$;
2: if verification failed then Event("Invalid $UpdVO$"); Abort;
3: $valueh \leftarrow UpdVO[0]$; $l \leftarrow len(valueh)$;
4: $split \leftarrow false$;
5: if $l < fanout - 1$ then
6:     $h' \leftarrow \mathsf{h}(valueh[0, l]||\,\mathsf{h}(o))$;
7: else
8:     $split \leftarrow true$; $half \leftarrow \lceil (fanout + 1)/2 \rceil$;
9:     $h'_1 \leftarrow \mathsf{h}(valueh[0, half])$;
10:    $h'_2 \leftarrow \mathsf{h}(valueh[half + 1, l]||\,\mathsf{h}(o))$;
11: for $i$ in [1, $len(UpdVO)$-1] do
12:    $childh \leftarrow UpdVO[i]$; $l \leftarrow len(childh)$;
13:    if $split = false$ then
14:        $h' \leftarrow \mathsf{h}(childh[0, l-1]||h')$;
15:    else
16:        if $l < fanout$ then
17:            $split \leftarrow false$; $h' \leftarrow \mathsf{h}(chidlh[0, l-1]||h'_1||h'_2)$;
18:        else
19:            $split \leftarrow true$; $half \leftarrow \lceil (fanout + 1)/2 \rceil + 1$;
20:            $h'_1 \leftarrow \mathsf{h}(childh[0, half])$;
21:            $h'_2 \leftarrow \mathsf{h}(childh[half + 1, l-1]||h'_1||h'_2)$;
22: if $split = false$ then $SC.w.h_{root} \leftarrow h'$;
23: else $SC.w.h_{root} \leftarrow \mathsf{h}(h'_1||h'_2)$;
24: Event("Successful update for $UpdVO.\mathsf{h}(o)$");
```

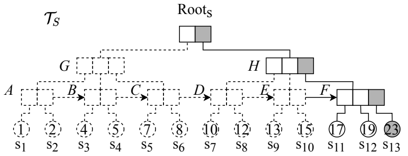

Fig. 7: Updated MB-tree $\mathcal{T}_S$ after Inserting s_{13}

we include at most F hashes for each level of the tree, requiring a total size of $O(\log_F n \cdot F \cdot |h|)$, where $|h|$ is the size of a hash value. As the $UpdVO$ is sent by the SP to the smart contract, the corresponding transaction cost is $O(\log_F n \cdot F \cdot |h| \cdot C_{txdata})$. The integrity verification of the $UpdVO$ requires the computation of the hash for each tree level and the checking of the root hash and the new object's hash, which cost $O(\log_F n \cdot (C_{hash} + F \cdot C_{mem}))$ and $2C_{sload}$ gas, respectively. To update the MB-tree's root hash, the hashes of $\log_F n$ intermediate nodes are re-computed, each costing $O(F \cdot C_{mem} + C_{hash})$ gas. Furthermore, up to $\log_F n$ node splits may occur during the insertion. Each node split takes extra $C_{hash} + C_{mem}$ gas compared with the case without node splits. Finally, the smart contract needs to update the root hash on-chain, which costs $C_{supdate}$ gas. In all, the worse-case maintenance cost for a single keyword's MB-tree is:

$$\begin{aligned} C_{\text{SMI}}^{\text{insert}}(n) = & \log_F n \cdot (F \cdot |h| \cdot C_{txdata} + 3C_{hash} + (2F+1)C_{mem}) \\ & + 2C_{sload} + C_{supdate} \end{aligned}$$

For a new object with L keywords, the total cost is $L \cdot C_{\text{SMI}}^{\text{insert}}(n) + C_{tx}$, where C_{tx} is the basic cost of the transaction of sending the $UpdVO$ from the SP to the smart contract. It is worthwhile to note that the coefficient of the logarithmic term in the cost function only contains the cheap operations, i.e., $C_{txdata}, C_{hash}, C_{mem}$, and that the costly operations like $C_{supdate}$ and C_{sload} are with a constant coefficient only. Therefore, the Suppressed Merkle^{inv} index achieves a lower

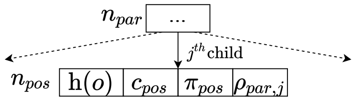

Fig. 8: Example of a Chameleon Tree Node

maintenance cost compared with the Merkle^{inv} index.

V. Chameleon^{inv} Index

Owing to the MB-tree structure, the Suppressed Merkle^{inv} index still has a logarithmic maintenance cost with respect to the memory operations. As shown in Section III-A, the chameleon vector commitment (CVC) has a nice property that one can update a vector without changing its digest using a secret trapdoor. Thus, inspired by the CVC [16], in this section, we propose a Chameleon inverted (Chameleon^{inv}) index, which has a constant maintenance cost and supports efficient authenticated keyword search.

A. Overview of the Chameleon^{inv} Index

To start with, we first present a Chameleon tree for indexing the objects of a single keyword. A Chameleon tree is a q-ary tree in which each node corresponds to a data object o and a node position pos, which is assigned by counting the nodes with the order of left to right and top to bottom. The root is a special node without a corresponding object, and it contains a CVC commitment computed by $\mathsf{Com}_{\mathsf{pp}}(\underbrace{\langle 0,...,0\rangle}_{q+1}, PRF(sk,w))$, where $\underbrace{\langle 0,...,0\rangle}_{q+1}$ is a constant vector, $PRF(\cdot)$ is a keyed pseudorandom function, sk is a private key owned by the DO, and w is the keyword. For each non-root node at position pos, as shown in Fig. 8, it consists of four components: $\{\mathsf{h}(o), c_{pos}, \pi_{pos}, \rho_{par,j}\}$, where $\mathsf{h}(o)$ is the object hash, c_{pos} is a CVC commitment computed by $\mathsf{Com}_{\mathsf{pp}}(\underbrace{\langle 0,...,0\rangle}_{q+1}, PRF(sk,pos||w))$, π_{pos} is the CVC proof proving that $\mathsf{h}(o)$ is the first element stored in an updated vector of c_{pos}, and $\rho_{par,j}$ proves that this node is linked to the j^{th} child of the parent node at position par (how to generate π_{pos} and $\rho_{par,j}$ will be discussed in Section V-B). With the child-parent relationship, the membership of o can be verified recursively in a bottom-up fashion, which will be elaborated in Section V-C.

Note that the commitment of each node is pre-determined by a constant vector, keyword w, and node position pos. The commitment remains unchanged, while the vector can be updated using a trapdoor during object insertions. Meanwhile, a new node with the inserted object is always linked to its parent node from left to right. With these two properties, given the total count of objects, cnt, the structure of a Chameleon tree is decided. The value of cnt needs to be maintained on-chain, which incurs a constant cost, in order to prevent an adversary from returning the query results based on an outdated index.

With the Chameleon tree, we can build the Chameleon^{inv} index, in which each keyword corresponds to a Chameleon tree. For each Chameleon tree, the smart contract only needs to maintain the (invariant) root commitment and the total object count cnt, both of which act as the authenticated information and are used during authenticated keyword search.

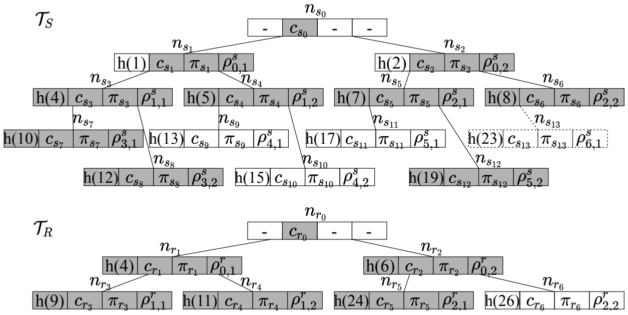
$\mathcal{T}_S$
n_{s_0}
- c_{s_0} - -
n_{s_1}
h(1) c_{s_1} π_{s_1} $\rho^s_{0,1}$
n_{s_2}
h(2) c_{s_2} π_{s_2} $\rho^s_{0,2}$
n_{s_3}
h(4) c_{s_3} π_{s_3} $\rho^s_{1,1}$
n_{s_4}
h(5) c_{s_4} π_{s_4} $\rho^s_{1,2}$
n_{s_5}
h(7) c_{s_5} π_{s_5} $\rho^s_{2,1}$
n_{s_6}
h(8) c_{s_6} π_{s_6} $\rho^s_{2,2}$
n_{s_7}
h(10) c_{s_7} π_{s_7} $\rho^s_{3,1}$
n_{s_9}
h(13) c_{s_9} π_{s_9} $\rho^s_{4,1}$
$n_{s_{11}}$
h(17) $c_{s_{11}}$ $\pi_{s_{11}}$ $\rho^s_{5,1}$
$n_{s_{13}}$
h(23) $c_{s_{13}}$ $\pi_{s_{13}}$ $\rho^s_{6,1}$
n_{s_8}
h(12) c_{s_8} π_{s_8} $\rho^s_{3,2}$
$n_{s_{10}}$
h(15) $c_{s_{10}}$ $\pi_{s_{10}}$ $\rho^s_{4,2}$
$n_{s_{12}}$
h(19) $c_{s_{12}}$ $\pi_{s_{12}}$ $\rho^s_{5,2}$
$\mathcal{T}_R$
n_{r_0}
- c_{r_0} - -
n_{r_1}
h(4) c_{r_1} π_{r_1} $\rho^r_{0,1}$
n_{r_2}
h(6) c_{r_2} π_{r_2} $\rho^r_{0,2}$
n_{r_3}
h(9) c_{r_3} π_{r_3} $\rho^r_{1,1}$
n_{r_4}
h(11) c_{r_4} π_{r_4} $\rho^r_{1,2}$
n_{r_5}
h(24) c_{r_5} π_{r_5} $\rho^r_{2,1}$
n_{r_6}
h(26) c_{r_6} π_{r_6} $\rho^r_{2,2}$

Fig. 9: Example of Two Chameleon Trees

Algorithm 3 Setup(q, w) by DO

```
1: function SETUP($q$, $w$)
2:     Generate PRF key $sk \leftarrow \{0,1\}^\lambda$;
3:     Compute a key pair of CVC $(\mathsf{pp}, \mathsf{td}) \leftarrow \mathsf{CGen}(1^\lambda, q+1)$;
4:     Set $cnt \leftarrow 0$; Compute $r_0 \leftarrow PRF(sk, 0||w)$;
5:     Compute $(c_0, aux) \leftarrow \mathsf{Com}_{\mathsf{pp}}(\langle 0,...,0\rangle, r_0)$;
6:     Send $\langle w, c_0\rangle$ to the blockchain and the SP;
```

B. Maintenance of the Chameleon^{inv} Index

For ease of illustration, we focus on the operation to add a new object to the Chameleon tree of a single keyword in the inverted index.

Algorithm 3 describes the setup procedure, which is run by the DO to create the initial Chameleon tree. It starts with generating the necessary keys, including the PRF key sk, the CVC trapdoor td, and its public parameter pp (lines 2-3). Note that sk and td need to be kept secret by the DO, otherwise they may be used by an adversary to tamper with the index. After the key generation, the commitment of the root node, c_0, is computed (lines 4-5). Finally, the keyword w and the root commitment c_0 are sent to both the blockchain and the SP (line 6).

The procedure to insert a new object o into the Chameleon tree is described in Algorithm 4. It consists of two steps: (i) creating a new node for o, and (ii) linking the new node to its corresponding parent node. In the first step, a new node's position pos is assigned and the pre-determined commitment c_{pos} is computed (lines 2-3). Then, the DO finds the collision of c_{pos} using the trapdoor td and updates the vector's first element to the object's hash $\mathsf{h}(o)$ (line 4). After that, a proof π_{pos} is generated (line 5). In the second step, the DO first locates the parent node n_{par} and the corresponding index j in the CVC (line 6). Then, n_{par}'s pre-determined commitment c_{par} is computed and a collision is found to update its $(j+1)^{th}$ element to c_{pos} (lines 7-8). Next, a corresponding proof $\rho_{par,j}$ is computed to attest to that n_{pos} is the j^{th} child of n_{par} (line 9). We denote $\langle c_{pos}, \pi_{pos}, \rho_{par,j}\rangle$ as the *insertion proof* of the new object o. Finally, the DO sends the insertion proof to the SP and the updated cnt value to the blockchain (lines 10-11).

Example. *Fig. 9 shows two Chameleon trees with $q = 2$ (the object ID is used to denote the object for simplicity). We give an example of inserting a new object o with id $= 23$ to the Chameleon tree $\mathcal{T}_S$. The total object count cnt is incremented from 12 to 13, which is also o's node position. We create a*

Algorithm 4 Insert($sk, \mathsf{td}, cnt, w, o$) by DO

```
function INSERT($sk, \mathsf{td}, cnt, w, o$)
    $cnt \leftarrow cnt + 1$; $pos \leftarrow cnt$; $r_{pos} \leftarrow PRF(sk, pos||w)$;
    $(c_{pos}, aux_{pos}) \leftarrow \mathsf{Com}_{\mathsf{pp}}(\langle 0, ..., 0 \rangle, r_{pos})$;
    $aux'_{pos} \leftarrow \mathsf{CCol}_{\mathsf{pp}}(c_{pos}, 1, 0, \mathsf{h}(o), \mathsf{td}, aux_{pos})$;
    $\pi_{pos} \leftarrow \mathsf{Open}_{\mathsf{pp}}(1, \mathsf{h}(o), aux'_{pos})$;
    $(par, j) \leftarrow$ getPar($pos$); $r_{par} \leftarrow PRF(sk, par||w)$;
    $(c_{par}, aux_{par}) \leftarrow \mathsf{Com}_{\mathsf{pp}}(\langle 0, ..., 0 \rangle, r_{par})$;
    $aux'_{par} \leftarrow \mathsf{CCol}_{\mathsf{pp}}(c_{par}, j + 1, 0, c_{pos}, \mathsf{td}, aux_{par})$;
    $\rho_{par,j} \leftarrow \mathsf{Open}_{\mathsf{pp}}(j + 1, c_{pos}, aux'_{par})$;
    Send $\langle cnt, o, \mathsf{h}(o), c_{pos}, \pi_{pos}, \rho_{par,j} \rangle$ to the SP;
    Send $\langle w, cnt \rangle$ to the blockchain;
```

Algorithm 5 Authenticated Join($\mathcal{T}_R$, $\mathcal{T}_S$) by SP

```
Input Two Chameleon trees $\mathcal{T}_R$, $\mathcal{T}_S$
Output Verification object $VO_{sp}$
$pos_{start} \leftarrow 1$;
loop
    $target \leftarrow \mathcal{T}_R[pos_{start}]$; $VO_{sp}.add(target, \Pi_R(target))$;
    $(pos_l, pos_u) \leftarrow$getPos($\mathcal{T}_S$, $target.id$);
    $VO_{sp}.add(\mathcal{T}_S[pos_l], \Pi_S(\mathcal{T}_S[pos_l]))$;
    if $pos_u \neq +\infty$ then
        $VO_{sp}.add(\mathcal{T}_S[pos_u], \Pi_S(\mathcal{T}_S[pos_u]))$;
        $pos_{start} \leftarrow pos_u$;
    else $VO_{sp}.add(+\infty, \mathcal{T}_S.cnt)$; break;
    Switch the roles of $\mathcal{T}_R$ and $\mathcal{T}_S$;
```

new node for o. With $pos = 13$, the node's commitment $c_{s_{13}}$ is computed, the parent node position $par = 6$ and the element index $j = 1$ are determined. Next, the two proofs, $\pi_{s_{13}}$ and $\rho^s_{6,1}$, are computed by finding the collisions of $c_{s_{13}}$ and c_{s_6}. The insertion proof, consisting of $\langle c_{s_{13}}, \pi_{s_{13}}, \rho^s_{6,1} \rangle$, is then sent to the SP. The count $cnt = 13$ is sent to the blockchain as part of the authenticated information.

Cost Analysis. To analyze the smart contract maintenance cost of the Chameleon^{inv} index, we first consider the cost of adding a new object to a single Chameleon tree. While the root commitment is invariant, the smart contract only needs to update the object count, which costs $C_{supdate}$ gas. Therefore, the maintenance cost of the Chameleon tree is simply:

$$C^{\text{insert}}_{\text{Chameleon}} = C_{supdate}$$

For an object with L keywords, the total maintenance cost is $L \cdot C^{\text{insert}}_{\text{Chameleon}}$. Apparently, this cost is constant.

C. Authenticated Keyword Search with Chameleon^{inv} Index

In this section, we discuss how to process authenticated keyword search with the Chameleon^{inv} index. Similar to the one discussed in Section IV, a keyword search query is transformed to join the query keywords' Chameleon trees for each conjunctive component of the query condition.

As a building block, we start by introducing the authenticated membership test on the Chameleon tree. To prove the existence of an object at position pos in the Chameleon tree, the SP can generate a *membership proof*, Π, by including the insertion proofs of the object at pos and all its ancestor nodes except the root. For example, in Fig. 9, the membership proof of s_3 in the tree $\mathcal{T}_S$ consists of the insertion proofs of s_3 and s_1, i.e., $\{c_{s_3}, \pi_{s_3}, \rho^s_{1,1}, c_{s_1}, \rho^s_{0,1}\}$. The membership verification is processed recursively in a bottom-up fashion. In the above example, one can use π_{s_3} to prove that s_3 is stored in n_{s_3} and $\rho^s_{1,1}$ to prove that n_{s_3} is the first child of n_{s_1}. Finally, with $\rho^s_{0,1}$ and the root commitment c_{s_0} retrieved from the blockchain, n_{s_1} is proved that it is indeed the first child of the root and the verification is completed.

Algorithm 6 Result Verification(VO_{chain}, VO_{sp}) by Client

```
Input $VO_{sp}$ from SP, $VO_{chain}$ from blockchain
Output Whether the verification is passed
Retrieve $\langle c_0, cnt \rangle$ of the joined trees as $VO_{chain}$ from the
  blockchain;
for each round in $VO_{sp}$ do
    Check the position of this round's target is 1 or equals the
      previous round's boundary position;
    Verify two boundary objects with their $\Pi$ w.r.t. its $c_0$;
    Verify the target with its $\Pi$ and add the target as a result when
      the target's ID equals the left boundary's ID;
    Verify the target's ID being in the two boundary IDs;
    if the round is the last round then
        Check the termination position $\geq cnt$;
```

To process authenticated keyword search, without loss of generality, let us consider the case of joining two Chameleon trees, which is described in Algorithm 5. The overall procedure is similar to the MB-trees' join explained in Section III-B. The major difference is that the Chameleon tree is indexed by each object's node position, rather than the object ID. To facilitate join processing, the SP builds a hash map locally to keep track of the mappings between the object IDs and their positions for each Chameleon tree. Similar to the MB-tree, the join query is processed in rounds between the two trees. For each round, the membership proofs of (i) the target and (ii) the matching and boundary objects are added to the VO_{sp}. The two trees switch their roles to find the matching objects until one tree's end is reached.

Algorithm 6 shows the procedure of the client's verification process, which is also similar to that of the MB-tree join. First of all, the client retrieves the root commitment c_0 and count cnt (denoted as VO_{chain}) for each relevant Chameleon tree. Then, the client checks the VO_{sp} for each round of join processing. Specifically, for each round, the client (i) checks the position of this round's target is 1 or equals the previous round's boundary position; (ii) verifies the integrity of the two boundary objects using their membership proofs; (iii) verifies the integrity of the target object with its membership proof and adds the object as a result if its ID matches the left boundary's ID; (iv) verifies that the target object's ID is between the boundary objects' IDs. For the last round, the client additionally (v) checks the termination position is no smaller than the object count cnt. The checking of (ii) and (iii) ensures the results' soundness, while that of (i), (iv), and (v) guarantees the results' completeness.

Example. *We use Fig. 9 to give an example of the authenticated keyword search with the query Q = "COVID-19" $\wedge$ "Symptom". The Chameleon trees $\mathcal{T}_S$ and $\mathcal{T}_R$ correspond to the keyword "COVID-19" and "Symptom", respectively. At first, the first object of $\mathcal{T}_R$, r_1, with its membership proof $\Pi_R(r_1)$ is inserted to VO_{sp}. Here, $\Pi_R(r_1)$ consists of the insertion proof of the node r_1. Then, r_1 is used as a target and its matching and boundary objects, s_3 and s_4, with their* *membership proofs are added to VO_{sp}. For the next round, the roles of $\mathcal{T}_R$ and $\mathcal{T}_S$ switch and the target changes to s_4, which is the previous round's right boundary. The process continues until we reach the end of $\mathcal{T}_S$ with the target r_5 and its boundary s_{13}. The components in the membership proofs are shaded in Fig. 9.*

For result verification, after retrieving the root commitments c_{r_0}, c_{s_0} and their cnt values from the blockchain, the client verifies r_1 with its proof $\Pi_R(r_1)$. Also, the matching and boundary objects, s_3 and s_4, are verified with their membership proofs. After checking r_1 with its left boundary s_3, the client knows that r_1 is a result. For the next round, s_4 is checked against its boundary object r_2. The verification goes on and finally after checking the termination position 13 *is no smaller than the tree's cnt value, the whole process is completed.*

D. Chameleon^{inv*} Index

One disadvantage of the Chameleon^{inv} index is that the verification of the commitments (in the membership proofs) involves heavy cryptographic pairing operations, which are several orders of magnitude slower than the cryptographic hash function [16]. As a result, the client's verification cost is high when using the Chameleon^{inv} index. To reduce the verification cost, we propose an optimized Chameleon^{inv*} index, in which a Bloom filter is created for every b objects in each Chameleon tree and used to efficiently prove the non-existence of objects. Specifically, during the insertion of a new object, the object's ID is inserted to the current Bloom filter if it is not full; otherwise a new Bloom filter is created. We also set a range for each filter with the smallest and largest IDs. The Bloom filters are maintained by the smart contract on-chain to ensure their integrity. Nevertheless, compared with the Chameleon^{inv} index, the smart contract cost does not increase too much since the maintenance of a Bloom filter incurs only a constant cost of $C_{supdate} + C_{sstore}/b + C_{sload}$. Thus, we have

$$C^{\text{insert}}_{\text{Chameleon*}} = 2C_{supdate} + C_{sstore}/b + C_{sload}$$

The Bloom filters can not only improve the verification performance, but also the query performance since the unmatched objects can be efficiently tested in the join algorithm. As such, the SP also maintains the same set of filters. The authenticated join of the Chameleon* trees is slightly different from Algorithm 5. When joining for a target object, we use the corresponding Bloom filter in the second index tree to test whether a matching object exists. If so, we proceed as usual. Otherwise, the consecutive object is set as the target to continue the join process. The verification of the authenticated join with the Chameleon* tree is similar to Algorithm 6 except that some unmatched objects can be quickly verified with the Bloom filters retrieved from the blockchain.

VI. Security Analysis

In this section, we analyze the security of the proposed ADS algorithms. We start by presenting the formal definitions of our security notions.

Definition 1 (ADS Maintenance Security)**.** *We say an ADS maintenance algorithm is secure if the success probability of any polynomial-time adversary is negligible in the following experiment:*

- *an adversary $\mathcal{A}$ selects two different datasets $\mathbb{D}_{\mathit{ideal}}$ and $\mathbb{D}_{\mathit{real}}$;*
- *the ADS maintenance algorithm constructs ADS and its corresponding authenticated information $VO_{chain,\mathit{ideal}}$ based on $\mathbb{D}_{\mathit{ideal}}$ with an honest party acted as the SP;*
- *the ADS maintenance algorithm constructs ADS and its corresponding authenticated information $VO_{chain,\mathit{real}}$ based on $\mathbb{D}_{\mathit{real}}$ with the adversary $\mathcal{A}$ acted as the SP;*
- *the adversary succeeds if $VO_{chain,\mathit{ideal}} = VO_{chain,\mathit{real}}$.*

This above definition ensures that the chance for a malicious SP to forge incorrect authenticated information is negligible.

Definition 2 (Authenticated Query Processing Security)**.** *We say an authenticated keyword query algorithm is secure if the success probability of any polynomial-time adversary is negligible in the following experiment:*

- *an adversary $\mathcal{A}$ selects a dataset $\mathbb{D}$;*
- *the ADS maintenance algorithm constructs ADS and its corresponding authenticated information VO_{chain} based on $\mathbb{D}$;*
- *$\mathcal{A}$ outputs a tuple of range query Q, result R, and VO_{sp};*
- *the adversary succeeds if VO_{sp} passes the verification w.r.t. the VO_{chain} and satisfies the condition: $\{r_i | r_i \notin Q(\mathbb{D}) \wedge r_i \in R\} \neq \varnothing \vee \{r_j | r_j \in Q(\mathbb{D}) \wedge r_j \notin R\} \neq \varnothing$.*

The above definition states that the chance for a malicious SP to convince the user of an incorrect or incomplete answer is negligible. We now show that our proposed algorithms indeed satisfy the desired security requirements.

Theorem 1. *Our proposed ADS maintenance algorithms are secure with respect to Definition 1 if the underlying cryptographic primitives and blockchain smart contract executions are secure.*

Proof. This theorem is clearly true for both the Merkle^{inv} index and the Chameleon^{inv} index, since the adversary does not participate in the computation for the on-chain authenticated information. Next, we prove that this theorem is also true for the Suppressed Merkle^{inv} index by contradiction. Recall that the authenticated information is computed by the smart contract with the input of the update proof $UpdVO$. A successful attack would mean either the smart contract execution is not secure or the input is tampered with. The first case is a direct contradiction to the assumption. For the second case, a tampered update proof (i.e., $UpdVO_{\text{ideal}} = UpdVO_{\text{real}}$) that passes the smart contract check means that (i) there exist two MB-trees with different objects but the same root hash, or (ii) there exists another object whose hash is the same as that of the new object. In either case, it implies a successful collision of the underlying hash function, which contradicts to our assumption. □

Theorem 2. *Our proposed authenticated query algorithms are secure with respect to Definition 2 if the underlying cryptographic primitives are secure.*

Proof. We omit the proof for the Suppressed Merkle^{inv} index as its query algorithm is identical to that of the Merkle^{inv}

index. We prove that this theorem is also true for the Chameleon^{inv} index by contradiction.

Case 1: $\{r_i | r_i \notin Q(\mathbb{D}) \wedge r_i \in R\} \neq \varnothing$. This means that a tampered r_i is returned, which is not in the genuine result set. For the case where all queried keywords exist, once r_i passes the client verification, it means that the SP can get two different vectors with the same commitment since the client needs to verify r_i's membership proof and compare the root commitment from VO_{chain}. However, this contradicts to the assumption that without the trapdoor td, the SP cannot find a valid collision of a CVC. For the case where the query contains a non-existing keyword, the genuine result should be empty. But if r_i is returned and passes the verification, this means that the client can find a digest that corresponds to the non-existing keyword in VO_{chain}. This contradicts to the assumption that the smart contract is trustworthy and would not forge a digest for the non-existing keyword.

Case 2: $\{r_j | r_j \in Q(\mathbb{D}) \wedge r_j \notin R\} \neq \varnothing$. This means that a valid result r_j is missing from R. Since the client successfully checks these conditions: (i) the soundness of the target and the boundary objects, (ii) the target's ID residing the boundary objects' IDs, and (iii) the target being the previous round's right boundary, the missing r_j means that the SP can find the collision of the target's or the boundary objects' commitments with some other object's hash. This contradicts to the assumption that the SP cannot find a valid collision of a CVC without the trapdoor td. Also, note that if the SP uses an out-of-date cnt as the termination boundary to remove some results, the client can detect it since it does not match the one from the VO_{chain}. □

VII. Performance Evaluation

In this section, we evaluate the performance of the proposed ADS schemes on the Ethereum blockchain platform.

A. Experiment Setup

We use two datasets, namely DBLP[6] and Twitter[7], for performance evaluation. The DBLP dataset consists of 5 million paper entries with titles, authors, and affiliations. The twitter dataset on the other hand contains 1.5 million tweets. For both datasets, an incremental 32-bit identifier is assigned for each data object. Moreover, we extract the keywords for each data object by removing stop words from its content.

For the proposed ADS schemes, the following settings are adopted. Since the word size in the Ethereum is 32 bytes, the fan-out of the MB-trees in the Merkle^{inv} index is set to 4, which is the maximum of F satisfying $(F-1)\cdot l_d + F \cdot l_p + l_p <$ 32 bytes, where l_d and l_p are the sizes of a delimiter (4 bytes) and a pointer (3 bytes), respectively. We choose the same fan-out for the other ADSs for the sake of fair comparison. For the Chameleon^{inv*} index, we fix the length of a Bloom filter to 256 bits, which is identical to the word size in the Ethereum. By default, the maximum number of the objects allowed to be inserted into a Bloom filter, b, is set to 30.

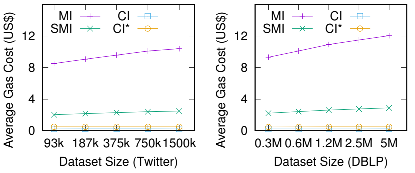

Fig. 10: Gas Consumption vs. Dataset Size

TABLE III: Gas Cost Breakdown in US$ (Twitter)

ADS	**Write Cost** $(C_{sstore}, C_{supdate})$	**Read Cost** (C_{sload})	**Others** $(C_{txdata}, C_{hash}$, etc.)	**Total**
MI	7.44	1.19	1.76	10.39
SMI	0.13	0.01	2.36	2.50
CI	0.13	0.00	0.11	0.24
CI*	0.28	0.01	0.21	0.50

In the experiments, a private Ethereum network is deployed using Geth and the block *gasLimit* is set to 8,000,000.[8] The smart contract is written in the Solidity language. For the SP, a server equipped with Dual 10-Core Intel Xeon E5-2630 v4 2.2GHz CPU and 256G RAM, running CentOS 7, is used. For each of the DO and the client, a desktop computer with Intel Core i7-7700K 4.2GHz CPU and 16 GB RAM is used. The query processing and the result verification programs are written in the Rust programming language. We choose SHA-3 as the cryptographic hash function in all algorithms and MNT4 298 as the elliptic curve for implementing the CVC. The cryptographic pairing operations used by the CVC is implemented using the ZEXE library [18].

We evaluate the performance of four ADS schemes, namely the baseline Merkle^{inv} index and the proposed Suppressed Merkle^{inv}, Chameleon^{inv}, and Chameleon^{inv*} indexes. For simplicity, we denote them as MI, SMI, CI, and CI*, respectively. The following performance metrics are measured and reported: (i) the smart contract's *gas* consumption for ADS maintenance; (ii) the SP's query processing time; (iii) the size of the VO (including both VO_{sp} and VO_{chain}); (iv) the client verification time.

B. Experimental Results

1) Gas Consumption for ADS Maintenance: Fig. 10 shows the average gas consumption[9] per object insertion with the increasing dataset size. Table III also shows the gas cost breakdown for the full Twitter dataset. Clearly, all of our proposed schemes (i.e., SMI, CI, and CI*) outperform the baseline MI. Moreover, their costs are only slightly increased when the dataset becomes larger. Specifically, compared to MI, SMI reduces the average gas consumption from US$10.39 down to US$2.50, achieving a saving of 76%. This is attributed to the idea of using the update proof to maintain the root hashes on the smart contract, which avoids the costly maintenance of the complete index on-chain. However, SMI still has a high transaction cost ($2.36 out of $2.50) owing to the logarithmic update proof. In contrast, the savings obtained by CI and CI* are more significant, as high as 98%, since both of them require only a constant maintenance cost. In detail, CI costs only an average of US$0.24 for each object insertion, which contains very small write cost and no read cost. As for CI*, it consumes slightly more gas to maintain the additional

[6] https://dblp.uni-trier.de/xml/

[7] https://www.kaggle.com/kazanova/sentiment140

[8] https://github.com/ethereum/go-ethereum

[9] The gas consumption is reported with the same cost settings of Table I.

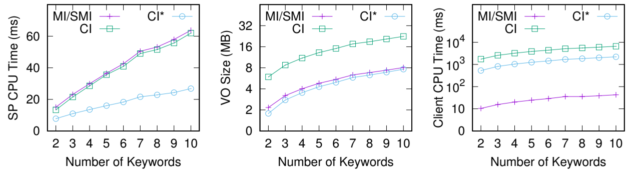

Fig. 11: Query Processing & Verification Costs (Twitter)

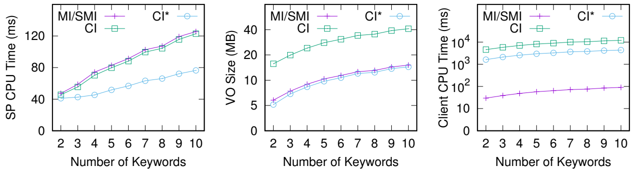

Fig. 12: Query Processing & Verification Costs (DBLP)

Bloom filters, with an average of US$0.50 for each object insertion. The observed performance differences conform to our theoretical cost analysis presented in Sections IV and V.

2) Query Performance: We now evaluate the query performance of different ADS schemes for the two datasets. Fig. 11 and Fig. 12 show the results for conjunctive query conditions, where the number of query keywords is varied from 2 to 10. Similar performance trends are observed when combined with disjunctive query conditions, so they are omitted in the interest of space. The query keywords are selected randomly from the most frequent 10,000 keywords. For the CI and CI* schemes, we use four threads during the result verification for better performance. For each experiment, 1,000 queries are generated and the average performance results are reported. Note that MI and SMI employ the same algorithms for query processing and, hence, their performances are exactly the same.

As can be seen from Fig. 11 and Fig. 12, the metrics of all schemes increase with the number of query keywords. This is because the more the query keywords, the more the index trees to be joined and, hence, a longer query time and a larger VO size are resulted. MI and CI have a similar SP processing time while CI* is more efficient owing to its use of Bloom filters. The Bloom filters help the SP efficiently determine the unmatched objects, which expedites the query processing. At the same time, they contribute to a smaller VO size. However, due to the costly cryptographic primitives used during the verification of the CVC, the verification times of CI and CI* are longer than that of MI, which relies only on the relatively fast hash operations. Nevertheless, the Bloom filters in the CI* scheme help reduce the verification time to some extent.

Finally, we evaluate the influence of the parameter b, which is the maximum number of objects allowed in a Bloom filter in a Chameleon* tree. We vary b from 20 to 50 and report the query performance for the Twitter dataset in Fig. 13. As we can see, the default setting $b = 30$ yields the best results in terms of the SP CPU time, VO size, and client verification time. This is because when b is set too small, the effectiveness of using Bloom filters to filter the unmatched objects is not obvious. However, if b is too large, since we fixed the Bloom filter's length, it leads to a high false positive rate, making its pruning capability less effective. For this reason, we chose $b = 30$ as the default parameter in the previous experiments.

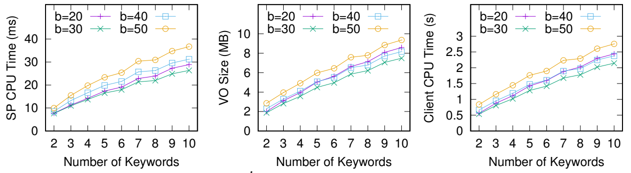

Fig. 13: Chameleon^{inv*} Performance vs. b (Twitter)

VIII. Related Work

To the best of our knowledge, no studies exist for authenticated keyword search in hybrid-storage blockchains. In the following, we briefly review the related works on blockchain data management and authenticated query processing.

A. Blockchain Data Management

Blockchain data management has recently received increasing attention from the database research community. To increase the system scalability, several studies proposed a variety of improvements over the blockchain transaction execution and commitment model [19]–[22]. Moreover, sharding techniques were investigated to scale the system horizontally [23], [24]. To reduce the storage overhead among peers, [25] and [26] explored distributed data storage schemes.

There are also some existing works that studied the keyword search upon the blockchain. Hu *et al.* [27] proposed a searchable encryption scheme on blockchain. Jiang *et al.* [28] proposed a verifiable keyword search framework for outsourced encrypted data based on blockchain. The multi-keyword search problem for encrypted data on blockchain was investigated in [29]. However, all of these prior works utilize smart contracts to process the queries directly. As such, they fail to consider the storage and computation costs of the blockchain. Moreover, they suffer from poor scalability and high query costs. In comparison, our proposed schemes not only achieve a low on-chain maintenance cost and but also support efficient authenticated keyword search.

B. Authenticated Query Processing

Authenticated query processing has been extensively studied in the outsourced database scenario to ensure result integrity [7]–[11]. They are two fundamental approaches. On the one hand, the computation task can be viewed to be run on a verifiable general Turing machine. By presenting the task as a Boolean or arithmetic circuit, one can achieve integrity assured-computations for arbitrary queries [30]. However, such an approach often yields high and sometimes impractical overhead. On the other hand, one can design an authenticated query processing scheme specifically based on the queries. This is often achieved by letting the DO sign a well-designed ADS. One of the most commonly used ADSs is the MHT [14] introduced in Section III. The MHT technique has been adapted to many indexes to cater for different applications, such as range queries [7], join queries [8], and text search engines [31]. It has also been applied to read query verification in the Byzantine learner problem [32].

Another closely related topic is on verifiable data streaming [16], [33], [34], in which a resource-limited client outsources a stream of data to an untrusted server. These data can later be retrieved with integrity ensured in a publicly verifiable fashion. In [16], a CVC scheme was proposed to achieve constant data appending cost and constant public key maintenance for the data owner. However, it is limited to basic data retrieval and not able to support authenticated keyword search.

In the prior work [35], we proposed an accumulator-based authentication structure for verifiable queries over the data stored on blockchain, where the off-chain storage is not considered. A gas-efficient structure called GEM^2-tree was designed for hybrid-storage blockchains in [13]. However, it supports only range queries. Moreover, as discussed in Section IV-B, because the on-chain structure is only partially suppressed in the GEM^2-tree, its index maintenance cost remains relatively high. In contrast, our proposed ADS schemes fully suppress the on-chain tree structures and achieve a lower maintenance cost.

IX. Conclusion

In this paper, we have studied the problem of authenticated keyword search over the data stored in a hybrid-storage blockchain system. To reduce the on-chain maintenance cost, we have proposed the Suppressed Merkle^{inv} index that incurs a logarithmic maintenance cost while supporting efficient query processing and result verification. Furthermore, with the inspiration of the CVC, we have designed the Chameleon^{inv} index and its optimized version Chameleon^{inv*} to further reduce the on-chain maintenance cost to a constant level. Authenticated query algorithms over the proposed indexes have also been developed. Theoretical analysis and empirical results have validated the robustness and performance of the proposed techniques.

Our proposed Suppressed Merkle^{inv} index can be easily extended to other indexes such as B-tree and R-tree to support various queries. For future work, we plan to extend the idea of the Chameleon tree to more general indexes where the keys of newly inserted objects are not monotonically increment. This poses new challenges since the objects can no longer be appended by the order of node positions. Besides, we are interested to explore learned indexes for authenticated queries in blockchain systems.

Acknowledgement This work is supported by Hong Kong RGC under Project Nos. 12201520 & 12200819 & 12201018 and Guangdong Science & Technology Special Fund SDZX2019042.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] J. M. Graglia and C. Mellon, "Blockchain and property in 2018: At the end of the beginning," *Innovations: Technology, Governance, Globalization*, 2018.

[3] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol," in *13th NSDI*, 2016, pp. 45–59.

[4] B. Liu, X. L. Yu, S. Chen, X. Xu, and L. Zhu, "Blockchain based data integrity service framework for IoT data," in *IEEE ICWS*, 2017.

[5] M. J. M. Chowdhury, A. Colman, M. A. Kabir, J. Han, and P. Sarda, "Blockchain as a notarization service for data sharing with personal data store," in *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2018.

[6] IBM, "Why new off-chain storage is required for blockchains," 2018. [Online]. Available: https://www.ibm.com/downloads/cas/RXOVXAPM

[7] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *ACM SIGMOD*, 2006.

[8] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis, "Authenticated join processing in outsourced databases," in *ACM SIGMOD*, 2009.

[9] Q. Chen, H. Hu, and J. Xu, "Authenticated online data integration services," in *ACM SIGMOD*, 2015.

[10] C. Xu, Q. Chen, H. Hu, J. Xu, and X. Hei, "Authenticating aggregate queries over set-valued data with confidentiality," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2017.

[11] C. Xu, J. Xu, H. Hu, and M. H. Au, "When query authentication meets fine-grained access control: A zero-knowledge approach," in *ACM SIGMOD*, 2018.

[12] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014.

[13] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi, "GEM^2-Tree: A gas-efficient structure for authenticated range queries in blockchain," in *IEEE ICDE*, 2019.

[14] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*, 1989.

[15] D. Catalano and D. Fiore, "Vector commitments and their applications," in *International Workshop on Public Key Cryptography*, 2013.

[16] J. Krupp, D. Schröder, M. Simkin, D. Fiore, G. Ateniese, and S. Nuernberger, "Nearly optimal verifiable data streaming," in *Public-Key Cryptography–PKC*, 2016.

[17] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys (CSUR)*, 2006.

[18] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," in *IEEE S&P*, 2020.

[19] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: the case of hyperledger fabric," in *ACM SIGMOD*, 2019.

[20] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran, "Blockchain meets database: Design and implementation of a blockchain relational database," *PVLDB*, 2019.

[21] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains," in *ACM SIGMOD*, 2020.

[22] V. Zakhary, D. Agrawal, and A. E. Abbadi, "Atomic commitment across blockchains," *PVLDB*, 2020.

[23] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *ACM SIGMOD*, 2019.

[24] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy, "BlockchainDB: A shared database on blockchains," *PVLDB*, 2019.

[25] Z. Xu, S. Han, and L. Chen, "Cub, a consensus unit-based storage scheme for blockchain system," in *IEEE ICDE*, 2018.

[26] X. Qi, Z. Zhang, C. Jin, and A. Zhou, "BFT-Store: Storage partition for permissioned blockchain via erasure coding," in *IEEE ICDE*, 2020.

[27] S. Hu, C. Cai, Q. Wang, C. Wang, X. Luo, and K. Ren, "Searching an encrypted cloud meets blockchain: A decentralized, reliable and fair realization," in *IEEE INFOCOM*, 2018.

[28] S. Jiang, J. Liu, L. Wang, and S.-M. Yoo, "Verifiable search meets blockchain: A privacy-preserving framework for outsourced encrypted data," in *IEEE International Conference on Communications*, 2019.

[29] S. Jiang, J. Cao, J. A. McCann, Y. Yang, Y. Liu, X. Wang, and Y. Deng, "Privacy-preserving and efficient multi-keyword search over encrypted data on blockchain," in *IEEE International Conference on Blockchain*, 2019.

[30] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE S&P*, 2013.

[31] H. Pang and K. Mouratidis, "Authenticating the query results of text search engines," *PVLDB*, 2008.

[32] J. Hellings and M. Sadoghi, "Coordination-free byzantine replication with minimal communication costs," in *23rd ICDT*, 2020.

[33] D. Schröder and H. Schröder, "Verifiable data streaming," in *ACM CCS*, 2012.

[34] D. Schöder and M. Simkin, "Veristream — a framework for verifiable data streaming," in *International Conference on Financial Cryptography and Data Security*, 2015.

[35] C. Xu, C. Zhang, and J. Xu, "vChain: Enabling verifiable boolean range queries over blockchain databases," in *ACM SIGMOD*, 2019.