A Semantic Hash Tree Based Verifiable Data Access Protocol on the Cloud

Fei Chen*, Tao Xiang†, Jianyong Chen‡, Wei Yu§, Xinwen Fu¶, and Shengyu Zhang‖

*Department of Computer Science and Engineering, Shenzhen University. Email:fchen@szu.edu.cn.
†College of Computer Science, Chongqing University. Email: txiang@cqu.edu.cn.
‡Department of Computer Science and Engineering, Shenzhen University. Email: jychen@szu.edu.cn.
§Department of Computer & Information Sciences, Towson University. Email: wyu@towson.edu.
¶Department of Computer Science, University of Massachusetts Lowell. Email: xinwenfu@cs.uml.edu.
‖Department of Computer Science and Engineering, The Chinese University of Hong Kong. Email:syzhang@cse.cuhk.edu.hk.

***Abstract*—With the popularity of outsourcing data to the cloud, security of cloud storage has drawn considerable attention. While many research efforts have been devoted to verifying the availability and integrity of the outsourced data, it remains a critical challenge how to efficiently verify the correctness of the cloud's response on data access request. In particular, when a user requests a data item from a mobile or Web interface, what if the cloud claims that the data item does not exist while the item does exist? We refer to this as *verifiable data access*. In this paper, we present a formal model for verifiable data access, and then propose a privacy-preserving and provably secure protocol to address the problem. In order to achieve the verifiability, we develop a novel mechanism called *semantic hash tree*. Through a sophisticated design of such a tree, our protocol supports verifiable data access in constant time with logarithm communication cost. Experimental evaluation further validates the practicality and efficiency of our protocol.**

I. Introduction

Cloud computing security has drawn considerable attention in recent years. To mitigate security issues for cloud storage, research efforts have been devoted to cloud storage auditing and encrypted search. Cloud storage auditing can help users verify whether their outsourced data remains available and intact in the cloud [1]–[6]. Encrypted search can enable users to search for their encrypted data (which is encrypted before outsourcing) efficiently in the cloud [7]–[15].

While cloud storage auditing and encrypted search address important security issues in cloud storage, other severe security problems remain. In this paper, We consider the following problem: On a data item access request from a user, an untrusted cloud may cheat when responding to the request. The cloud could either claim a data item not existing when the data item does exist or vice versa. From the perspective of the user, both cases are possible and then the user is fooled by an untrusted cloud. The cheating of the cloud could be due to various reasons, e.g. internal/external adversaries, hardware/software failures, government pressures, etc. Therefore, users who buy cloud storage services should have the ability to verify the cloud's answer. This requirement is referred to as *Verifiable Data Access* (VDA).

The corresponding author is Jianyong Chen (jychen@szu.edu.cn).

It is worth noting that encrypted search in the cloud cannot address verifiable data access efficiently. Most current research on encrypted search assumes that the cloud always follows the protocol [7]–[9], [12]–[14]. Only a few pieces of work such as [10], [11], [15] tackle malicious clouds. While these results can enable verifiability of encrypted search, they are not efficient. Thus, how to solve the verifiable data access problem efficiently still remains very challenging.

To address the aforementioned verifiable data access problem, we propose an efficient and provably secure protocol to empower the user to verify the received answer when accessing a data item in cloud storage. Our protocol achieves two desirable effects: one is to protect the privacy of the data item during a user access request and the other is to enable the user to verify whether the returned answer is correct. The first effect is obtained by employing a pseudorandom function, HMAC (Hash Message Authentication Code), in our protocol to mask the ID of the data item. The second effect is achieved by exploiting two ideas: one is to actively separate all possible data items (including both existing and non-existing data items) and the other is to sophisticatedly authenticate the data before outsourcing. The first idea is implemented by assigning a special ID for all non-existing data items and the second idea is realized by a novel enhancement of the hash authentication tree.

From a high-level view, our protocol runs as follows: When outsourcing data items, we employ a hash table to store all IDs of existing data items as well as non-existing data items in a sophisticated way. Along with the data items, the hash table also contains versatile semantics along with outsourced data. Then, the hash table is authenticated using a semantic hash tree. Later, the hash table together with the semantic hash tree is outsourced. When a user accesses a data item, the cloud returns both the data item and the corresponding semantics-embedded authentication path of the semantic hash tree as a correctness proof. We combine the hash table and the semantic hash tree gracefully to make our protocol highly efficient.

978-1-4673-8537-4/15 $31.00 © 2015 IEEE
DOI 10.1109/CBD.2015.43

To summarize, in this paper we make the following contributions:

- We propose an efficient, privacy-preserving, and provably secure protocol to enable users to verify the result of a data access on a remote cloud.
- We analyze the performance of our protocol and formally prove its security. Furthermore, we conduct extensive experiments to evaluate the effectiveness of our proposal.

The paper proceeds as follows. In Section II, we formulate the verifiable data access problem and present a solution framework. In Sections III and IV, we show the full details of our protocol design and a complete analysis of our protocol, respectively. The analysis includes correctness, security, privacy, and efficiency of our protocol. In Section V, we evaluate our protocol experimentally, followed by a discussion of related work in Section VI. Finally, we conclude the paper in Section VII.

II. Problem Formulation

In this section, we formulate the verifiable data access (VDA) problem. We also present a high-level design to a protocol solving the VDA problem by explicitly verifying the returned answers from the cloud. To better understand the security of a VDA protocol, we abstract a formal security definition by refining the security intuition of a practical VDA protocol. Finally, we introduce some technical mechanisms which are construction components in our verifiable data access protocol.

A. System Model, Assumptions & Design Goals

We model a system that supports VDA as in Fig. 1. The system contains three entities: a data owner, a data user and a cloud. In practice, the data owner could be a company and then the data user is an employee of the company. The data owner possesses valuable data, which is outsourced to the cloud. Before outsourcing the data, the data owner processes the data with a secret key such that the result returned from the cloud can be verified. Then, the data owner shares the secret key with the data user. When the data user wants to access a data item, the user sends the data item ID (e.g. filename) to the cloud to query for the file. If the data item does exist, the cloud returns the data item and a correctness proof showing that the returned data is indeed the requested data. If the data item does not exist, the cloud indicates that no such data with the requested ID exists using a proof. We aim to design a verifiable data access protocol in this paper to support the system in Fig. 1.

We model the cloud as malicious. The cloud can cheat in two ways: the cloud may claim that an existing data item does not exist or vice versa. We explicitly state several assumptions to make our problem clearer. We assume that the ID and the content of a data item is well-combined, authenticated, and encrypted using a Message Authentication Code (MAC) and a data encryption algorithm. Hence, a malicious cloud cannot modify the content of the data item and the privacy of the data content is well protected from the cloud. We are not focusing on protecting the access pattern of the user to trade-off for more efficiency, as most of previous works in encrypted search do, including [7]–[9], [12]–[14]. In theory, hiding the access pattern from the cloud can be enabled using oblivious RAM [16]. Nonetheless, oblivious RAM incurs a huge computational cost [16]. Thus, the VDS protocol in this paper aims at protecting the privacy of user by efficiently protecting the IDs of the requested data items.

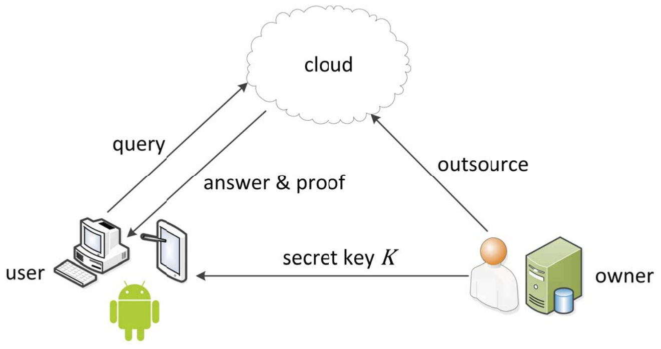

Fig. 1. System Model for VDA

It is worth stressing that the real challenging part of the verifiable data access problem is how to verify the returned result from the cloud on a data item access request. What is not challenging is how to prevent the cloud not modifying the content of a data item, which can be solved straightforwardly by authenticating the data content using a message authentication code (MAC). In our protocol design, such data content protection is also applied, as we have assumed explicitly in the above. Thus we mainly focus on the variability of the cloud's answer in this paper.

In summary, a verifiable data access protocol should protect the privacy of the user and enable the verifiability of the cloud's response. More specifically, a VDA protocol should have the following properties:

- **Correct**. The protocol should enable the user to get the requested data item if all parties are honest, i.e. following the protocol without cheating.
- **Verifiable**. If the cloud cheats, the data user is able to detect such behavior. For existing data items, the cloud cannot claim the non-existence of them; for non-existing data items, the cloud also cannot reply with an existence proof for the corresponding data item access request.
- **Privacy-preserving**. By running the protocol, the cloud should not get meaningful knowledge about the requested data item. The privacy of the data item ID and data content should be well protected.
- **Efficient**. The computation, storage, and communication of all parties should be as small as possible.

B. Solution Framework & Security Definition

We now present a framework to the VDA problem and formally define its security. By abstracting the system model as shown in Fig. 1, a verifiable data access protocol VDA =

(KeyGen, Outsource, Query, Search, Verify) contains five efficient algorithms as follows:

- $\mathsf{KeyGen}(1^\lambda) \to K$: On input a security parameter λ, the data owner runs this algorithm to output a secret key K.
- $\mathsf{Outsource}(F; K) \to F'$: On input a collection of data F, the data owner processes the data to embed some authentication information in the data using the secret key K and then outsources the processed data F' to the cloud.
- $\mathsf{Query}(f; K) \to q$: On input a data item ID f to be requested, the data user sends a request query q containing the information about f to the cloud.
- $\mathsf{Search}(q) \to (\chi, \Gamma)$: On input a query q for some data item, the cloud finds the data item in the outsourced data. Then, the cloud sends the queried data item χ back; the cloud also sends a proof Γ to show χ is indeed the data item the user requested.
- $\mathsf{Verify}(f, \chi, \Gamma; K) \to \delta$: On input the returned answer χ, correctness proof Γ, the original requested data item f, and the secrete key K, the data user checks whether the cloud cheats. If the verification succeeds, accept the answer and output $\delta = 1$; else, reject the answer and output $\delta = 0$.

We define the security of a VDA protocol based on the cloud's capability (i.e. thread model) and the security intuition. First, the cloud's capability is that it can observe many interactions between the users and the cloud when a VDA protocol is run. Second, intuitively, a VDA protocol is secure if no real-world polynomial time cloud can fool the user on any data item (either existing or non-existing) access request with a high probability. We capture this intuition using the concept of forgery: we say a tuple (f^*, χ^*, Γ^*) is a *forgery* if either f^* exists but the cloud (χ^*, Γ^*) claims that it does not exist, or f^* does not exist but the cloud (χ^*, Γ^*) claims that it does exist. The intuition of security is then that a malicious cloud cannot find such a forgery. Thus, we define security as follows:

Definition 1. *Let* VDA = (KeyGen, Outsource, Query, Search, Verify) *be a verifiable data access protocol, $\mathcal{A}$ be a malicious cloud, and* $\mathrm{Adv}(\lambda; \mathsf{VDA})$ *be the probability that $\mathcal{A}$ cheats successfully by finding a forgery (f^*, χ^*, Γ^*) after $\mathcal{A}$ observes many verification results.* $\mathrm{Adv}(\lambda; \mathsf{VDA})$ *is computed as*

$$\Pr\left[\begin{array}{c} K \leftarrow \mathsf{KeyGen}(1^\lambda) \\ F' \leftarrow \mathsf{Outsource}(F; K) \\ q_i \leftarrow \mathsf{Query}(f_i; K) \\ (\chi_i, \Gamma_i) \leftarrow \mathsf{Search}(q_i) \\ \delta_i \leftarrow \mathsf{Verify}(f_i, \chi_i, \Gamma_i; K) \\ i = 1, \cdots, \mathrm{poly}(\lambda) \end{array} : \begin{array}{c} \mathcal{A}(F', q_i, \delta_i) \\ \textit{outputs a} \\ \textit{forgery} \\ (f^*, \chi^*, \Gamma^*) \\ \textit{and the user} \\ \textit{accepts it} \end{array}\right] \tag{1}$$

where $\mathrm{poly}(\lambda)$ *denotes a polynomial function in λ. We say that a* VDA *protocol is secure if the cheating probability* $\mathrm{Adv}(\lambda; \mathsf{VDA})$ *is negligible (i.e.* $\mathrm{Adv}(\lambda; \mathsf{VDA}) < \frac{1}{\mathrm{poly}(\lambda)}$ *holds asymptotically for any polynomial in λ).*

C. Technical Mechanisms

CRYPTOGRAPHIC HASH FUNCTION. Our protocol employs hash functions extensively, which is introduced here. A hash function is a compressing function mapping any finite-length string over $\{0, 1\}$ to a fixed-length string. A *cryptographic* hash function is a hash function with more security constraints. Let $\mathcal{A}$ be a real-world polynomial time algorithm and

$$\Pr\left[\begin{array}{c} h_i = \mathsf{Hash}(m_i) \\ i = 1, \cdots, \mathrm{poly}(\lambda) \end{array} : \begin{array}{c} \mathcal{A}(m_i, h_i) \text{ outputs} \\ m \neq m^* \text{ such that} \\ \mathsf{Hash}(m) = \mathsf{Hash}(m^*) \end{array}\right] \tag{2}$$

be the probability of finding a successful collision, denoted as $\mathrm{Adv}(\mathsf{Hash})$. A cryptographic hash function is said to be collision-resistant if $\mathrm{Adv}(\mathsf{Hash})$ is negligible for any $\mathcal{A}$. In our protocol, hash functions are used in two different ways. One is to use a hash function to index an element in an array and the other is to use a cryptographic hash function to build a (semantic) hash authentication tree.

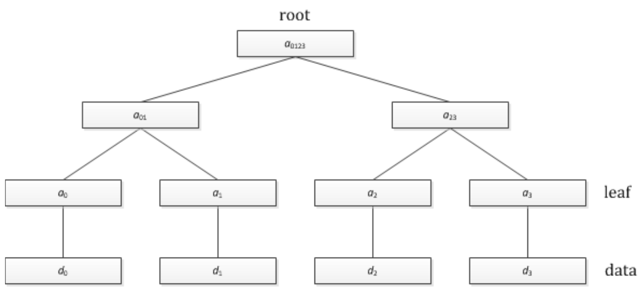

Fig. 2. Hash Tree Example

HASH AUTHENTICATION TREE. A hash authentication tree is a complete-binary-tree-like data structure which can authenticate the integrity of the data, which was first proposed by Merkle in late 1970's [17]. Figure 2 shows an example of a hash authentication tree with height 2. The leaf level of a hash authentication tree is associated with a collection of data items, which are to be authenticated as a whole. Each node of the tree has a value. For the i-th leaf node, its value is $\mathsf{Hash}(d_i)$ where d_i is the i-th data item and $\mathsf{Hash}(\cdot)$ is a cryptographic collision resistant hash function. The hash authentication tree is computed recursively from the bottom to the top. For each internal node of the hash tree, its value is computed as $\mathsf{Hash}(left \parallel right)$ where $left$ and $right$ are the values of its left child and right child respectively, and $\parallel$ denotes concatenation of two strings. The value of the root node is the authentication information of the data collection. To authenticate the i-th data item, the path from the i-th leaf node to the root node, which contains all necessary intermediate hash values and the i-th data item, is provided as a proof. For example, the tuple $(d_2, a_2, a_3, a_{01}, a_{0123})$ can authenticate d_2. To verify this proof, one first check whether a_{0123} is same with the root value, then checks whether the authentication path is correct, i.e. $a_2 = \mathsf{Hash}(d_2)$ and $a_{0123} = \mathsf{Hash}(\mathsf{Hash}(a_2, a_3), a_{01})$. If all verification passes, then the data item d_2 is authenticated.

PSEUDORANDOM FUNCTION. A pseudorandom function family $\{F_K(\cdot)\}$ indexed by K is a set of *deterministic* functions mapping a domain X to an image Y. Let the set $\mathbb{R}^{X \to Y}$ denote all functions mapping domain X to image Y. The

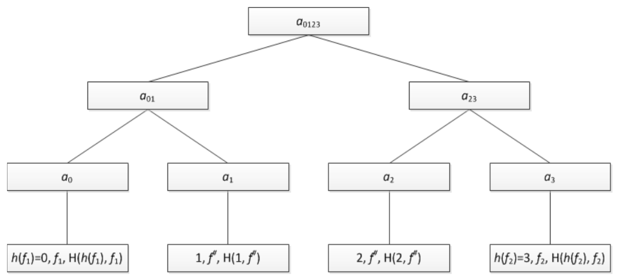

Fig. 3. An Example of Our Protocol

family $\{F_K(\cdot)\}$ is said to be pseudorandom if any polynomial-time algorithm cannot differentiate the set $\{F_K(\cdot)\}$ from $\mathbb{R}^{X \to Y}$ [18]. Intuitively, the output of a function in $\{F_K(\cdot)\}$ looks random.

III. Enabling Verifiable Data Access

In this section, we present all the details for our protocol design which enables verifiable data access on a cloud. We first explain the basic idea of our protocol design followed by an example. We then show how to address all technical challenges, followed by a detailed protocol design by discussing all detailed algorithms in VDA = (KeyGen, Outsource, Query, Search, Verify).

A. An Example

We use an example depicted in Fig. 3 to show our basic ideas. For convenience, we assume that there is no collision when hashing a data item into a slot of the hash table; we show how to handle collisions in a later section. Here, the data owner has two data items f_1' and f_2' to be outsourced. The data owner masks the data item IDs to obtain f_1 and f_2. The data owner use a unique ID $f^\#$ for all non-existing data items. Later, the data owner stores the data in a hash table of size 4. The hash function associated with the hash table is $h(\cdot)$ mapping the masked data item ID to an integer in $\{0, 1, 2, 3\}$. The data item f_1 is stored in the first slot of the hash table according to $h(f_1) = 0$ and f_2 in the last slot according to $h(f_2) = 3$. All remaining empty slots of the hash table are filled with the unique ID $f^\#$ for non-existing data items. Each slot of the hash table has the semantics $(index, ID, \mathsf{Hash}(index, ID))$, where $index$ is the position of data item ID in the hash table, which is equal to $h(ID)$ for existing data items, and $\mathsf{Hash}(\cdot)$ is a cryptographic hash function. We explain the reason for such semantics later. After the hash table is filled, the data owner constructs a semantic hash tree over the hash table. The leaf level of the hash tree is obtained by applying $\mathsf{Hash}(\cdot)$ on each slot of the hash table (e.g. $a_0 = \mathsf{Hash}(0, f_1, \mathsf{Hash}(0, f_1))$). The data owner then computes the semantic hash tree from bottom-up to get the root value a_{0123} as the authentication information for the outsourced data. The data owner shares the pseudorandom random function and the authentication information a_{0123} with the user.

When data item f_1' is requested, the user employs the pseudorandom function to obtain a masked ID f_1 and sends f_1 to the cloud. On receiving the request f_1, the cloud finds its slot in the hash table according to $h(\cdot)$, which is equal to 0. Then, the cloud returns the data item $(0, f_1, \mathsf{Hash}(0, f_1))$ and the authentication path $(a_0, a_1, a_{23}, a_{0123})$ to the user. On receiving the response from the cloud, the user verifies: (i) whether the returned data item is f_1 or $f^\#$ (in this case f_1 is matched to an existing data item), (ii) whether the returned slot is equal to $h(f_1)$, (iii) whether the data item is modified by the cloud by computing $\mathsf{Hash}(h(f_1), f_1)$ and comparing the result with the returned value, and (iv) whether the authentication path is correct using the stored authentication value a_{0123}.

When the data item with ID f_3' is requested, which does not exist, the user obtains a masked ID f_3 and sends f_3 to the cloud. Suppose $h(f_3) = 2$. The cloud checks the slot with index 2 in the hash table and finds that the unique ID $f^\#$ is contained in the corresponding slot. The unique ID $f^\#$ indicates that the requested data item does not exist; otherwise, the corresponding slot must contain an existing data item. Thus, the cloud returns $(2, f^\#, \mathsf{Hash}(2, f^\#))$ and the authentication path $(a_2, a_3, a_{01}, a_{0123})$ to the user. Then the user verifies: (i) whether the returned data item is f_3 or $f^\#$ (in this case, $f^\#$ is matched to a non-existing data item), (ii) whether the returned slot is $h(f_3)$, (iii) whether the data item is modified by the cloud by computing $\mathsf{Hash}(h(f_3), f^\#)$ and comparing the result with the returned value, and (iv) whether the authentication path is correct using the stored authentication value a_{0123}.

B. Addressing Technical Challenges

Our protocol design leaves several technical challenges which are addressed in this subsection:

Collision Handling. We employ the *double* hashing technique for handling the collisions of a hash table (i.e. using independent hash functions to find a different slot when a collision occurs) [19]. Specifically, suppose a data item ID is a string f' over $\{0, 1\}^*$ and its masked ID is also a string f over $\{0, 1\}^*$. To map this data item to a slot of the hash table, we can use a hash function $h_1 : \{0, 1\}^* \to \{0, 1, \cdots, L-1\}$, where L is the size of the hash table. The data item is to be put into the $h_1(f)$ slot. If slot $h_1(f)$ is occupied by some other data item, we use another hash function $h_2 : \{0, 1\}^* \to \{0, 1, \cdots, L-1\}$ to obtain another slot. If a collision happens again, a third hash function h_3 is used on the value $h_2(f)$. For later collisions, we fix the issue by using a third hash function $h_3(\cdot)$ iteratively on the value $h_3(\cdot)$.

Hash Table Size. We choose to balance the hash table size and the collision probability by using the *load factor* concept of the hash table (i.e. the ratio of the number of existing data items over the size of the hash table). Denote the load factor as α, the set of all existing data items as F, and $|F|$ as the hash table size. We can control the value of α to reduce the collision probability. An example for the load factor could be $\alpha = 0.1$. The size of the hash table should roughly be $\frac{|F|}{\alpha}$. Because a semantic hash tree is built on the hash table and the tree needs to be a complete binary tree, we set the hash table size to be $2^{\lceil \log_2 \frac{|F|}{\alpha} \rceil}$, where $\lceil x \rceil$ denotes the smallest integer that is greater than or equal to x. Note that the size of the hash table is linear only in the total number of outsourced data items.

Data Item Semantics. The data item in our protocol has the semantics $(index, ID, \mathsf{Hash}(index, ID))$. The "$index$" is determined by the masked data item ID and the hash functions of the hash table. To prevent the modification of a data item, $index$ and ID are then hashed using a cryptographic hash function. Note that including the plaintext ID in the data item gives explicit existence information about one data item. The term "$index$" is used to handle data item collisions when data items are placed in the slots of the hash table. A data item does not have a fixed index in the hash table if multiple collisions occur. If the term $\mathsf{Hash}(index, ID)$ is not contained in the data item semantics, the cloud has modified the returned answer in an attempt to cheat the user. We prove that this format is strictly secure in Section IV.

We note that the design of collision handling, hash tree size, and data item semantics in the hash table has a large impact on the correctness and security of our protocol as will be shown next.

C. Protocol Design

With the technical challenges addressed, we extend our basic idea to design a verifiable data access protocol VDA = (KeyGen, Outsource, Query, Search, Verify) in steps. In each step, we detail one algorithm in VDA = (KeyGen, Outsource, Query, Search, Verify).

VDA.KeyGen. The data owner runs this algorithm before outsourcing the data. Suppose a data owner wants to outsource a collection of data items F. The data owner first generates a secrete key K for a pseudorandom function $\mathrm{func}_K(\cdot)$. To protect the privacy of the data item with identity f', the data owner computes a masked ID $f = \mathrm{func}_K(f')$ for each data item.

VDA.Outsource. The data owner runs this algorithm when outsourcing the data. The data owner chooses a load factor α. According to the total number of the data items $|F|$ and α, the data owner instantiates an empty hash table with size $2^{\lceil \log_2 \frac{|F|}{\alpha} \rceil}$ and three hash functions h_1, h_2, h_3 associated with the hash table. The input of h_1, h_2, h_3 is any string in $\{0,1\}^*$ and the output is an integer in $\{0, 1, \cdots, 2^{\lceil \log_2 \frac{|F|}{\alpha} \rceil} - 1\}$. For each data item in F, the data owner uses the hash functions h_1, h_2, h_3 to find a slot in the hash table. Then, the data owner stores the item in the calculated slot of the hash table in the format $(index, ID, \mathsf{Hash}(index, ID))$, where $index$ is computed using h_1, h_2, h_3 and the data item ID, and $\mathsf{Hash}(\cdot)$ is a cryptographic hash function, e.g. SHA256. More specifically, if no collision occurs, $index = h_1(ID)$; if one collision occurs, $index = h_2(ID)$; if two collisions occur, $index = h_3(ID)$; if three collisions occur, $index = h_3(h_3(ID))$. Similar collision handling repeats until an empty slot of the hash table hash been found for each data item. All remaining empty slots of the hash table are filled with $(index, f^{\#}, \mathsf{Hash}(index, f^{\#}))$ where $index \in \{0, 1, \cdots, 2^{\lceil \log_2 \frac{|F|}{\alpha} \rceil} - 1\}$ is the natural position of the corresponding empty slot in the hash table and $f^{\#}$ is the unique identity for all non-existing data items. Once the data owner has filled the hash table, the has table is authenticated by using a semantic hash tree. For the i^{th} leaf node of the hash tree, the value is computed as the output of the cryptographic hash function $\mathsf{Hash}(\cdot)$ applied on the i^{th}-slot of the hash table, which is a tuple with semantics $(index, ID, \mathsf{Hash}(index, ID))$. Once the leaf level of the hash tree is determined, the hash tree is constructed from bottom to top. The data owner keeps the root value R of the semantic hash tree as an authentication of the outsourced data. Finally, the data owner sends the hash table together with the whole authentication tree to the cloud. The data owner also shares the secret key K for the pseudorandom function and the authentication value R with the user.

VDA.Query. The user runs this algorithm. To request a data item with ID f', the user computes the masked ID as $f = \mathrm{func}(f')$. Then, the user sends f as a data access request to the cloud.

VDA.Search. The cloud runs this algorithm on a data item access request as follows. The cloud first computes an index $h_1(f)$ for the requested data item f, and checks the content of the $h_1(f)$-th slot of the hash table, which is outsourced to the cloud by the data owner. The cloud then proceeds as follows:

- If the slot contains the unique ID $f^{\#}$, then the requested data item does not exist. The cloud thus returns this slot $(h_1(f), f^{\#}, \mathsf{Hash}(h_1(f), f^{\#}))$ and its authentication path in the semantic hash tree back as a proof. Denote the proof for slot $h_1(f)$ as $P = [(h_1(f), f^{\#}, \mathsf{Hash}(h_1(f), f^{\#}))$, $(left, right)_H$, $(left, right)_{H-1}$, $\cdots$, $(left, right)_1$, $R]$ where H is the height of the hash authentication tree, $(left, right)_i$ are the elements corresponding to the $h_1(f)$-th leaf node at height i for $i = H, \cdots, 1$, and R is the root value of the semantic hash tree. For this case, we say the proof of the cloud contains one entry.
- If the slot contains the requested ID f, then the requested data item does exist. The cloud thus returns this slot $(h_1(f), f, \mathsf{Hash}(h_1(f), f))$ and its authentication path P in the semantic hash tree back as a proof. For this case, we also say the proof of the cloud contains one entry.
- If the slot contains an ID that is different from both the unique ID $f^{\#}$ and requested ID f, a collision occurs. Then, the cloud computes another index $h_2(f)$ and checks whether the $h_2(f)^{th}$ slot of the hash table contains the unique ID $f^{\#}$ or the requested ID f. Three cases are possible: (i) if $f^{\#}$ is found, then the requested data item does not exist; (ii) if f is found, then the requested data item does exist; and (iii) if a different data item ID is found, another collision occurs. We then handle this collision again using $h_3(f)$ until no further collisions happen. Eventually, either f or $f^{\#}$ is found. Suppose in total that k collisions happen and P_i is the proof for the slot when the i^{th} collision occurs. The cloud then returns $P_1 P_2 \cdots P_k$ to the user.

VDA.Verify. The user runs this algorithm upon receiving a response from the cloud. We divide the cloud's proof into the following two cases:

1) The proof has only one entry. First, we check whether the returned index is correct using the requested data item ID f and the hash function h_1. Second, we check whether the cryptographic hash of the $h_1(f)$-slot of the hash table is correct. Third, we check whether the authentication path of the semantic hash tree is correct. Only if all the checks pass, then the cloud is honest. If the requested data item ID f is found, the requested data item does exist; otherwise $f^{\#}$ is found and the requested data item indeed does not exist.
2) The proof has k entries $P_1 P_2 \cdots P_k$. The user checks the correctness of each entry P_i using the different hash functions h_1, h_2, h_3 of the hash table sequentially as case 1. If all checks are correct, the cloud is honest. The information contained in P_k determines whether the requested data item exists. If P_k has the requested data item ID, the requested data item does exist; otherwise, $f^{\#}$ is in P_k and the requested data item indeed does not exist.

D. Summary of the Protocol

Finally, we summarize our protocol in a compact and formal manner. Our protocol VDA = (KeyGen, Outsource, Query, Search, Verify) contains five probabilistic polynomial-time algorithms as follows:

- KeyGen(1^λ): The data owner runs this algorithm to output a secret key K for a pseudorandom function $\mathrm{func}_K(\cdot)$. The data owner shares K with the data user.
- Outsource($F; K$): The data owner runs this algorithm on input a collection of data items F. The data owner first masks the original data item IDs using $\mathrm{func}(\cdot)$. Then, the data owner determines a load factor α and initiates an empty hash table with size $2^{\lceil \log_2 \frac{|F|}{\alpha} \rceil}$ and height $\left\lceil \log_2 \frac{|F|}{\alpha} \right\rceil$. The data owner also determines three hash functions h_1, h_2, h_3 to map a masked data item ID to a slot of the hash table. Then, the data owner fills in all slots of the hash table with semantics embedded and builds a semantic hash tree over the hash table to get the root authentication value R. The data owner then outsources the hash table and the semantic hash tree to the cloud. The data owner also shares R and h_1, h_2, h_3 with the data user.
- Query($f'; K$): The data user runs this algorithm to send a request $q = f$ to the cloud, where $f = \mathrm{func}_K(f')$.
- Search(q): For a request q for some data item, the cloud finds it in the semantic hash tree. After handling all collisions, the cloud returns the data item and a proof back $(\chi, \Gamma) = \{P_1 P_2 \cdots P_k\}$ where k is the number of total collisions, P_i is the authentication for the i-th collision with the format $[(index,\ f_i, \mathsf{Hash}(index, f_i)),\ (left, right)_H,\ (left, right)_{H-1},\ \cdots,\ (left, right)_1,\ R]$.
- Verify($f', \chi, \Gamma; K$): After the data user receives an answer $\{P_1 P_2 \cdots P_k\}$ from the cloud, the data user checks whether the index, the data item ID, and the authentication path of each P_i are all correct. If yes, accept the cloud's answer; else, reject the answer.

IV. Formal Analysis: Security and Performance

A. Correctness & Privacy

We first analyze the correctness of our protocol. Suppose all parties follow the protocol. There are two cases: an existing data item request and a non-existing data item request. First, when the data user requests an existing data item, an honest cloud is able to find it in the hash table since VDA.Outsource has placed all existing data items in the hash table and the semantic hash tree. The honest cloud also returns a correct authentication of the semantic hash tree following our protocol. Thus, the data user can successfully verify the returned result. Second, the data user accesses a non-existing data item. Although there are many non-existing data items, we only use one special data item ID $f^{\#}$ to denote all these data. When the honest cloud searches the hash table, the cloud first computes an index i using h_1 and requested data item ID and checks the corresponding i-th slot of the hash table. A simple case is that the i-th data item is not occupied by an existing data item and the data item ID in the hash table is $f^{\#}$. Thus, the cloud just returns the i-th slot and the corresponding authentication path in the semantic hash tree. By checking the returned result, the data user knows that indeed the requested data item doesn't exist. However, collisions may occur and thus the first computed slot may contain other data items. In this case, the cloud can figure out that the current slot is different from the requested data item by comparing their IDs; then, the cloud computes and checks another index using h_2, h_3 repeatedly until no collision occurs. Finally, a slot containing $f^{\#}$ must be found since the requested data item does not exist. Thus, by returning all the proof entries, the data user can confirm that the queried data item indeed doesn't exist.

For privacy, the data item ID is masked by a pseudorandom function. Then, the cloud cannot understand the real ID of the requested data item; otherwise the cloud can differentiate a pseudorandom function from a real random function. The data item content is encrypted using a standard secure encryption algorithm. Thus, the privacy of the data item content is well protected.

B. Provable Security

Theorem 2. *If* $\mathsf{Hash}(\cdot)$ *is collision resistant, then our protocol is secure with respect to Definition 1.*

Proof Sketch. We only show a proof sketch due to page limit. The key observation is as follows: if the cloud can fool a user, the cloud must be able to show a different authentication path which is different from the true one in the semantic hash tree. Then, a collision for the cryptographic hash function can be constructed from the two different authentication paths.

□

TABLE I
THEORETICAL ASYMPTOTIC PERFORMANCE OF OUR PROTOCOL

	Computation	Storage	Communication
Data Owner	$O(n)$	$O(1)$	$O(n)$
Data User	$O(\log n)$	$O(1)$	$O(1)$
Cloud	$O(1)$	$O(n)$	$O(\log n)$

C. Theoretical Performance

Table I summarizes the performance of our protocol. Detailed analysis follows. We first focus on the data owner. The data owner needs to compute the semantic hash tree to get the root hash value as the authentication information. The computation time is composed of two parts: computing the masked data item ID and computing the semantic hash tree. For the first part, it takes $O(n)$ time. For the second part, it depends on the total size of the hash tree. The size of the leaf level is $2^{\lceil \log_2 \frac{n}{\alpha} \rceil}$, which is $O(n)$. Then, the tree size is at most $O(n) + O(n-1) + O(n) = O(n)$ since we employ a complete binary tree as our semantic hash tree. Thus, the computation of the data owner is $O(n)$ in total. For storage, the data owner only stores the secret key for the pseudorandom function and the root value of the semantic hash tree; it is a constant $O(1)$. For communication, the data owner needs to send the semantic hash tree which contains the data to the cloud. The communication cost is then $O(n)$. All the cost for the data owner is a one-time cost; thus, it can be amortized in the subsequent data item access requests. The same analysis also applies to the data user and the cloud.

Data user. The data user needs to send a data access request and verify an answer from the cloud. The former takes time $O(1)$ and the latter depends on the total number of proof entries. As we have analyzed that the number of collisions is $O(1)$ and the semantic hash tree has height $O(\log n)$, the length of an authentication path is $O(\log n)$ and the total length of the proof is also $O(\log n)$. Thus, the computation cost of the data user is $O(\log n)$. For storage, the data user only needs to store the secret key to generate the data access request and the root hash value; it takes $O(1)$ storage. For communication, the data user only sends the masked data item ID to the cloud and the size of the data is just the output size of the pseudorandom function. Thus, it is a constant, $O(1)$.

Cloud. The cloud searches a data item in the hash table. Although some collisions may occur, the computation is still a constant $O(1)$ since the total number of collisions is $O(1)$. However, for storage, the cloud needs to store the semantic hash tree and then the storage cost depends on the size of the semantic hash tree. As in the case of the data owner, the size of the semantic hash tree is $O(n)$. Thus, the storage of the cloud is also $O(n)$. For communication, the cloud needs to send back a proof containing a series of authentication paths. Since a single authentication path is with size $O(\log n)$ and the number of collisions is $O(1)$, the communication cost is $O(\log n)$.

TABLE II
DATA SETS AND STORAGE COST IN BYTES

Benchmark	Toy	RFC50	RFC100	RFC200	RFC400
Total Data Items	4	50	100	200	400
Storage	46152	182632	258840	440448	946392

V. EXPERIMENTAL EVALUATION

A. Methodology

We implemented our protocol using Java on a PC with Intel i3 3.1G CPU and 4GB of memory. We simulate a data owner, a user, a cloud, and their interactions in our prototype. For performance metrics, we focus on storage, computation, and communication cost of the protocol. For each performance indicator, we run the experiments 40 times and report the average performance result. We use public data sets and open-source our prototype [20].

We use five test data sets that can be found in Table II. The first one has only 4 random data items that can confirm the correctness of our prototype. The second data set is archived RFC documents 1 through 50 [21]. Similarly, the third, fourth and fifth data sets contain the first 100, 200, and then 400 RFC documents, respectively.

In our prototype, we choose HMAC based on SHA256 as the pseudorandom function to mask the data item ID. We employ SHA256 as the cryptographic hash function to construct the semantic hash tree. The load factor of the protocol is set to $\alpha = 0.1$.

B. Results

Table II shows the storage cost of our protocol. We measure only the storage required for the semantic hash tree as this is the main additional storage cost in our protocol. From Table II, we can find that the storage cost roughly grows linearly with the total number of data items. This is consistent with the theoretical analysis in Section IV-C. The minimal storage is 46,152 bytes for the "Toy" data set and the maximal storage cost is 94,6392 bytes for the "RFC400" data set. We emphasize that our protocol trade-off greater storage requirements for more computation and especially for communication efficiency. Furthermore, the storage cost of our protocol is also small compared to the huge storage pool of today's cloud storage service providers.

Table III contains the computational cost of our protocol. The most time-consuming operation is VDA.Outsource. This is expected since the semantic hash tree is built in this process. The time cost becomes larger with the total number of outsourced data items. The maximal time cost is about 0.042s, which is very small in practice. For other operations, the time cost is also very small. The experimental results show that all operations of our protocol take little time.

VI. RELATED WORK

Because of limited space, we only briefly discuss the research efforts very close to our research; more detailed discussion is to be presented in a future enhanced version. Encrypted search, authenticated data structure, and authenticated

TABLE III
COMPUTATION TIME FOR DIFFERENT ALGORITHMS IN NANOSECONDS

Benchmark	Outsource	Query	Access Existing	Access Non-existing	Verify Existing	Verify Non-existing
Toy	2712092	20037	27188	4825	58830	45720
RFC50	11921354	23000	51704	5421	83850	44081
RFC100	14576172	15137	30565	7366	62405	46191
RFC200	24524374	15874	37932	5437	65079	47640
RFC400	42127152	22462	36185	7175	81359	56711

databases are closely related to our work in this paper. The works in [7]–[15] proposed protocols to verify the keyword search results returned from an untrusted cloud. The works in [22]–[24] proposed protocols to authenticate outsourced data structures and databases. When these protocols are transformed to solve the verifiable data access problem in this paper, they are either not verifiable or inefficient. The work in [25] can enable verifiability, but requires a large computational cost.

VII. CONCLUSION

In this paper, we first formalized the verifiable data access problem and then proposed a privacy-preserving protocol to address the problem. The protocol builds its provable security on collision-resistance with respect to a cryptographic hash function. The proposed protocol is also highly efficient: it takes only constant time to access a data item and the communication cost is only the logarithm of the total number of outsourced data items. We implemented the proposed protocol to evaluate its performance and the experimental data validates its efficiency. Our open-source prototype also provides a foundation for further real-world development.

ACKNOWLEDGMENTS

The work in this paper is supported by the National Natural Science Foundation of China (No. 61502314) and the Fundamental Research Funds for the Shenzhen University (No. 201533).

REFERENCES

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proc. of ACM CCS*. ACM, 2007, pp. 598–609.

[2] A. Juels and B. Kaliski Jr, "PORs: Proofs of retrievability for large files," in *Proc. of ACM CCS*. ACM, 2007, pp. 584–597.

[3] K. Yang and X. Jia, "An efficient and secure dynamic auditing protocol for data storage in cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 9, pp. 1717–1726, 2013.

[4] C. Wang, S. S. Chow, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for secure cloud storage," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 362–375, 2013.

[5] F. Chen, T. Xiang, Y. Yang, and S. S. M. Chow, "Secure cloud storage meets with secure network coding," in *Prof. of IEEE INFOCOM*, 2014.

[6] E. Shi, E. Stefanov, and C. Papamanthou, "Practical dynamic proofs of retrievability," in *Proc. of ACM CCS*. ACM, 2013, pp. 325–336.

[7] D. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. of IEEE S&P*. IEEE, 2000, pp. 44–55.

[8] C. Wang, N. Cao, K. Ren, and W. Lou, "Enabling secure and efficient ranked keyword search over outsourced cloud data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1467–1479, 2012.

[9] C. Wang, K. Ren, S. Yu, and K. Urs, "Achieving usable and privacy-assured similarity search over outsourced cloud data," in *Proc. of IEEE INFOCOM*. IEEE, 2012, pp. 451–459.

[10] Q. Chai and G. Gong, "Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers," in *Prof. of IEEE ICC*. IEEE, 2012, pp. 917–922.

[11] K. Kurosawa and Y. Ohtaki, "UC-secure searchable symmetric encryption," in *Financial Cryptography and Data Security*. Springer, 2012, pp. 285–298.

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

[13] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Financial Cryptography and Data Security*. Springer, 2013, pp. 258–274.

[14] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *Proc. of NDSS*, 2014.

[15] Q. Zheng, S. Xu, and G. Ateniese, "VABKS: Verifiable Attribute-based Keyword Search over Outsourced Encrypted Data," in *Proc. of IEEE INFOCOM*. IEEE, 2014, pp. 451–459.

[16] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[17] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Proc. of CRYPTO '87*, 1988, pp. 369–378.

[18] O. Goldreich, "Foundation of cryptography (in two volumes: Basic tools and basic applications)," 2001.

[19] R. Motwani and P. Raghavan, "Randomized algorithms," *Cambridge University*, 1995.

[20] F. Chen, "Prototype: Verifiable data access on the cloud," https://sites.google.com/site/chenfeiorange/semantic-hash-tree-based-verifiable-file-search, 2015.

[21] RFC, "Request for comments database," http://www.ietf.org/rfc.html, 2014.

[22] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Authenticated hash tables," in *Proc. of ACM CCS*. ACM, 2008, pp. 437–448.

[23] H. Pang, J. Zhang, and K. Mouratidis, "Scalable verification for outsourced dynamic databases," in *Proc. of VLDB*. VLDB Endowment, 2009, pp. 802–813.

[24] H. Pang and K. Mouratidis, "Authenticating the query results of text search engines," in *Proc. of VLDB*. VLDB Endowment, 2008, pp. 126–137.

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