Publicly Verifiable Conjunctive Keyword Search in Outsourced Databases

Monir Azraoui, Kaoutar Elkhiyaoui, Melek Önen, Refik Molva
EURECOM, Sophia Antipolis, France
{azraoui, elkhiyao, onen, molva}@eurecom.fr

***Abstract*—Recent technological developments in cloud computing and the ensuing commercial appeal have encouraged companies and individuals to outsource their storage and computations to powerful cloud servers. However, the challenge when outsourcing data and computation is to ensure that the cloud servers comply with their advertised policies. In this paper, we focus in particular on the scenario where a data owner wishes to (i) outsource its public database to a cloud server; (ii) enable anyone to submit multi-keyword search queries to the outsourced database; and (iii) ensure that anyone can verify the correctness of the server's responses. To meet these requirements, we propose a solution that builds upon the well-established techniques of Cuckoo hashing, polynomial-based accumulators and Merkle trees. The key idea is to (i) build an efficient index for the keywords in the database using Cuckoo hashing; (ii) authenticate the resulting index using polynomial-based accumulators and Merkle tree; (iii) and finally, use the root of the Merkle tree to verify the correctness of the server's responses. Thus, the proposed solution yields efficient search and verification and incurs a constant storage at the data owner. Furthermore, we show that it is sound under the strong bilinear Diffie-Hellman assumption and the security of Merkle trees.**

I. Introduction

Cloud computing offers an opportunity for individuals and companies to offload to powerful servers the burden of managing large amounts of data and performing computationally demanding operations. In principle, cloud servers promise to ensure data availability and computation integrity at the exchange of a reasonable fee, and so far they are assumed to always comply with their advertised policies. However, such an assumption may be deemed unfounded: For instance, by moving their computing tasks into the cloud, cloud customers inherently lend the control to this (potentially malicious) third party, which (if left unchecked) may return an incorrect result for an outsourced computation, so as to free-up some of its computational resources. This lack of control on the part of cloud customers in this particular scenario, has given rise to an important body of work on *verifiable* computation, which aims at providing cloud customers with cryptographic tools to verify the compliance of cloud servers (i.e. to check whether the cloud server returns the correct result for an outsourced computation). A major requirement of verifiable computation is the efficiency of the verification at the cloud customer. Namely, verification should need less computational resources than the outsourced function, in order not to cancel out the advantages of cloud computing.

Owing to its prevalence in cloud computing, data mining is at the heart of verifiable computation: Cloud servers are the best candidates to undertake big-data mining, in that they have means to store big-data and own the necessary computational resources to run various data processing primitives and analyze huge data sets. In this paper, we focus on one of the most frequently used primitives in data mining, that is *keyword search*, and design a solution that assures the correctness of the search result. More specifically, we consider a scenario wherein a data owner wishes to outsource a public database to a cloud server and wants to empower third-party users (i) to issue conjunctive keyword search queries to the database and (ii) to verify the correctness of the results returned by the cloud efficiently. In other words, the data owner wants to ensure the properties of **public delegatability** and **public verifiability** as defined by Parno et al. [1]. Roughly speaking, public delegatability enables any user to perform verifiable conjunctive keyword search without having access to the data owner's secret information; whereas public verifiability guarantees that any third-party verifier (not necessarily the user originating the search query) can check the server's responses.

The core idea of our solution is to use polynomial-based accumulators to represent keywords in the outsourced database. Thanks to their algebraic properties, polynomial-based accumulators give way to two cryptographic mechanisms, that are verifiable test of membership (cf. [2]) and verifiable set intersection (cf. [3]). These two mechanisms together can be tailored to allow any third-party user to search a public database for multiple keywords and any third-party verifier to check the integrity of the result. Nonetheless, a straightforward application of polynomial-based accumulators to keyword search is too computationally demanding for the server, especially in the case of large databases. To this effect, we suggest to build an efficient index of the keywords in the database by means of Cuckoo hashing, and to authenticate the resulting index by a combination of polynomial-based accumulators and Merkle trees. Thus, we (i) allow the verifier to assess the correctness of the server's response in a logarithmic time, and (ii) enable the server to search the outsourced database efficiently. Furthermore, since our solution relies on polynomial-based accumulators and Merkle trees to assure the integrity of the search results, we show that it is provably secure under the strong bilinear Diffie-Hellman assumption and the security of Merkle trees.

The rest of this paper is organized as follows: Section II defines the problem statement, whereas Section III formalizes publicly verifiable conjunctive keyword search and the corre-

sponding adversary model. Section IV and Section V describe the building blocks and the proposed solution. Section VII evaluates our solution in terms of computational and storage cost. Section VIII reviews existing work on verifiable keyword search, and finally, Section IX wraps up the paper.

II. Problem Statement

To further illustrate the importance of public delegatability and public verifiability in cloud computing, we take the case where a pharmaceutical company would like (i) to outsource a set $\mathcal{F}$ of sanitized records of its clinical trials (of already marketed products) to a cloud server, and (ii) to delegate the conjunctive search operations on these records to its employees, or to external third parties such as the European Medicines Agency. Following the framework of publicly verifiable computation [1], the pharmaceutical company in this scenario will run a one-time *computationally demanding* pre-processing operation to produce a public key $\mathsf{PK}_{\mathcal{F}}$ and a lookup key $\mathsf{LK}_{\mathcal{F}}$. Together, these keys will make possible the implementation of publicly verifiable conjunctive keyword search. Namely, given public key $\mathsf{PK}_{\mathcal{F}}$, any user (be it an employee or a representative of the European Medicines Agency) will be able to search the outsourced records and verify the returned results. The server on the other hand is provided with lookup key $\mathsf{LK}_{\mathcal{F}}$ and thus, will be able to generate correct proofs for any well-formed search query. Furthermore, from public key $\mathsf{PK}_{\mathcal{F}}$, any user desiring to search the outsourced records will be able to derive a public verification key VK_Q, that lets any other third party (for instance, a judge in the case of a legal dispute between the pharmaceutical company and a patient) fetch the search result and assess its correctness quickly.

It is clear from the above example that our approach to handle verifiable conjunctive keyword search falls into the *amortized model* as defined by Gennaro et al. [4]. That is, the data owner engages in a one-time expensive pre-processing operation which will be amortized over an *unlimited number* of fast verifications. This model has been exploited to devise solutions for publicly verifiable computation, be it a generic computation as in [1] or a specific computation cf. [3], [5]. Arguably, one might customize one of these already proposed schemes to come up with a solution for verifiable conjunctive keyword search. Nevertheless, a solution based on the scheme in [1] will incur a large bandwidth overhead, whereas a solution that leverages the verifiable functions in [5] will not support public delegatability. Therefore, we choose to draw upon some of the techniques used in [3] (namely *verifiable set intersections*) to design a dedicated protocol that meets the requirements of public delegatability and public verifiability without sacrificing efficiency.

III. Publicly Verifiable Conjunctive Keyword Search

As discussed previously, publicly verifiable conjunctive keyword search enables a data owner O to outsource a set of files $\mathcal{F}$ to a server $\mathcal{S}$, while ensuring:

- **Public delegatability**: Any user $\mathcal{U}$ (not necessarily data owner O) can issue *conjunctive search* queries to server $\mathcal{S}$ for outsourced files $\mathcal{F}$. Namely, if we denote CKS the function which on inputs of files $\mathcal{F}$ and a collection of words $\mathcal{W}$ returns the subset of files $\mathcal{F}_{\mathcal{W}} \subset \mathcal{F}$ containing all words in $\mathcal{W}$, then public delegatability allows user $\mathcal{U}$ to outsource the processing of this function to server $\mathcal{S}$.
- **Public verifiability**: Any verifier $\mathcal{V}$ (including data owner O and user $\mathcal{U}$) can assess the correctness of the results returned by server $\mathcal{S}$, that is, verify whether the search result output by $\mathcal{S}$ for a collection of words $\mathcal{W}$ corresponds to $\mathsf{CKS}(\mathcal{F}, \mathcal{W})$.

In more formal terms, we define publicly verifiable conjunctive keyword search by the following algorithms:

- $\mathsf{Setup}(1^\kappa, \mathcal{F}) \rightarrow (\mathsf{PK}_{\mathcal{F}}, \mathsf{LK}_{\mathcal{F}})$: Data owner O executes this randomized algorithm whenever it wishes to outsource a set of files $\mathcal{F} = \{f_1, f_2, ...\}$. On input of a security parameter 1^κ and files $\mathcal{F}$, algorithm Setup outputs the pair of public key $\mathsf{PK}_{\mathcal{F}}$ and lookup key (i.e. search key[1]) $\mathsf{LK}_{\mathcal{F}}$.
- $\mathsf{QueryGen}(\mathcal{W}, \mathsf{PK}_{\mathcal{F}}) \rightarrow (\mathcal{E}_Q, \mathsf{VK}_Q)$: Given a collection of words $\mathcal{W} = \{\omega_1, \omega_2, ...\}$ and public key $\mathsf{PK}_{\mathcal{F}}$, user $\mathcal{U}$ calls algorithm $\mathsf{QueryGen}$ which outputs an encoded conjunctive keyword search query $\mathcal{E}_Q$ and the corresponding public verification key VK_Q.
- $\mathsf{Search}(\mathsf{LK}_{\mathcal{F}}, \mathcal{E}_Q) \rightarrow \mathcal{E}_R$: Provided with search key $\mathsf{LK}_{\mathcal{F}}$ and the encoded search query $\mathcal{E}_Q$, server $\mathcal{S}$ executes this algorithm to generate an encoding $\mathcal{E}_R$ of the search result $\mathcal{F}_{\mathcal{W}} = \mathsf{CKS}(\mathcal{F}, \mathcal{W})$.
- $\mathsf{Verify}(\mathcal{E}_R, \mathsf{VK}_Q) \rightarrow \mathsf{out}$: Verifier $\mathcal{V}$ invokes this deterministic algorithm to check the integrity of the server's response $\mathcal{E}_R$. Notably, algorithm Verify first converts $\mathcal{E}_R$ into a search result $\mathcal{F}_{\mathcal{W}}$, then uses verification key VK_Q to decide whether $\mathcal{F}_{\mathcal{W}}$ is equal to $\mathsf{CKS}(\mathcal{F}, \mathcal{W})$. Accordingly, algorithm Verify outputs $\mathsf{out} = \mathcal{F}_{\mathcal{W}}$ if it believes that $\mathcal{F}_{\mathcal{W}} = \mathsf{CKS}(\mathcal{F}, \mathcal{W})$, and in this case we say that verifier $\mathcal{V}$ accepts the server's response. Otherwise, algorithm Verify outputs $\mathsf{out} = \perp$, and we say that verifier $\mathcal{V}$ rejects the server's result.

In addition to public delegatability and public verifiability, a conjunctive keyword search should also fulfill the basic security properties of **correctness** and **soundness**. Briefly, correctness means that a response generated by an *honest* server will be always accepted by the verifier; soundness implies that a verifier accepts a response of a (potentially malicious) server **if and only if** that response is the outcome of a *correct* execution of the Search algorithm.

Correctness. A verifiable conjunctive keyword search scheme is said to be correct, if whenever server $\mathcal{S}$ operates algorithm Search *correctly* on the input of some encoded search query $\mathcal{E}_Q$, it always obtains an encoding $\mathcal{E}_R$ that will be accepted by verifier $\mathcal{V}$.

Definition 1. *A verifiable conjunctive keyword search is correct,* **iff** *for any set of files $\mathcal{F}$ and collection of words $\mathcal{W}$:*

[1]In the remainder of this paper, we use the terms lookup key and search key interchangeably.

Algorithm 1 The soundness experiment of publicly verifiable conjunctive keyword search

for $i := 1$ **to** t **do**
  $\mathcal{A} \rightarrow \mathcal{F}_i$
  $(\mathsf{PK}_{\mathcal{F}_i}, \mathsf{LK}_{\mathcal{F}_i}) \leftarrow \mathcal{O}_{\mathsf{Setup}}(1^\kappa, \mathcal{F}_i)$
end for
$\mathcal{A} \rightarrow (\mathcal{W}^*, \mathsf{PK}^*_{\mathcal{F}})$
$\mathsf{QueryGen}(\mathcal{W}^*, \mathsf{PK}^*_{\mathcal{F}}) \rightarrow (\mathcal{E}^*_Q, \mathsf{VK}^*_Q)$
$\mathcal{A} \rightarrow \mathcal{E}^*_R$
$\mathsf{Verify}(\mathcal{E}^*_R, \mathsf{VK}^*_Q) \rightarrow \mathsf{out}^*$

If $\mathsf{Setup}(1^\kappa, \mathcal{F}) \rightarrow (\mathsf{PK}_{\mathcal{F}}, \mathsf{LK}_{\mathcal{F}})$, $\mathsf{QueryGen}(\mathcal{W}, \mathsf{PK}_{\mathcal{F}}) \rightarrow (\mathcal{E}_Q, \mathsf{VK}_Q)$ *and* $\mathsf{Search}(\mathsf{LK}_{\mathcal{F}}, \mathcal{E}_Q) \rightarrow \mathcal{E}_R$*, then:*

$$\Pr(\mathsf{Verify}(\mathcal{E}_R, \mathsf{VK}_Q) \rightarrow \mathsf{CKS}(\mathcal{F}, \mathcal{W})) = 1$$

Soundness. We say that a scheme for publicly verifiable conjunctive keyword search is sound, if for any set of files $\mathcal{F}$ and for any collection of words $\mathcal{W}$, server $\mathcal{S}$ cannot convince a verifier $\mathcal{V}$ to accept an incorrect search result. In other words, a scheme for verifiable conjunctive keyword search is sound if and only if, the only way server $\mathcal{S}$ can make algorithm Verify accept an encoding $\mathcal{E}_R$ as the response of a search query $\mathcal{E}_Q$ for a set of files $\mathcal{F}$, is by correctly executing the algorithm Search (i.e. $\mathcal{E}_R \leftarrow \mathsf{Search}(\mathsf{LK}_{\mathcal{F}}, \mathcal{E}_Q)$).

To formalize the soundness of verifiable conjunctive keyword search, we define an experiment in Algorithm 1 which depicts the capabilities of an adversary $\mathcal{A}$ (i.e. malicious server $\mathcal{S}$). On account of public delegatability and public verifiability, adversary $\mathcal{A}$ does not only run algorithm Search but is also allowed to run algorithms QueryGen and Verify. This leaves out algorithm Setup whose output is accessed by adversary $\mathcal{A}$ through calls to the oracle $\mathcal{O}_{\mathsf{Setup}}$.

More precisely, adversary $\mathcal{A}$ enters the soundness experiment by *adaptively* invoking oracle $\mathcal{O}_{\mathsf{Setup}}$ with t sets of files $\mathcal{F}_i$. This allows adversary $\mathcal{A}$ to obtain for each set of files $\mathcal{F}_i$ a pair of public key $\mathsf{PK}_{\mathcal{F}_i}$ and search key $\mathsf{LK}_{\mathcal{F}_i}$. Later, adversary $\mathcal{A}$ picks a collection of words $\mathcal{W}^*$ and a public key $\mathsf{PK}^*_{\mathcal{F}}$ from the set of public keys $\{\mathsf{PK}_{\mathcal{F}_i}\}_{1 \leq i \leq t}$ it received earlier. Adversary $\mathcal{A}$ is then challenged on the pair $(\mathcal{W}^*, \mathsf{PK}^*_{\mathcal{F}})$ as follows: (i) It first executes algorithm QueryGen with public key $\mathsf{PK}^*_{\mathcal{F}}$ and the collection $\mathcal{W}^*$ and accordingly gets an encoded search query $\mathcal{E}^*_Q$ and the matching verification key VK^*_Q; (ii) afterwards, it generates a response $\mathcal{E}^*_R$ for encoded search query $\mathcal{E}^*_Q$, and concludes the experiment by calling algorithm Verify with the pair $(\mathcal{E}^*_R, \mathsf{VK}^*_Q)$.

Let out^* denote the output of algorithm Verify on input $(\mathcal{E}^*_R, \mathsf{VK}^*_Q)$. Adversary $\mathcal{A}$ succeeds in the soundness experiment if: (i) $\mathsf{out}^* \neq \perp$ and (ii) $\mathsf{out}^* \neq \mathsf{CKS}(\mathcal{F}^*, \mathcal{W}^*)$, where $\mathcal{F}^*$ is the set of files associated with public key $\mathsf{PK}^*_{\mathcal{F}}$.

Definition 2. *Let $\mathcal{A}dv_{\mathcal{A}}$ denote the advantage of adversary $\mathcal{A}$ in succeeding in the soundness game, i.e., $\mathcal{A}dv_{\mathcal{A}} = \Pr(\mathsf{out}^* \neq \perp \wedge \mathsf{out}^* \neq \mathsf{CKS}(\mathcal{F}^*, \mathcal{W}^*))$.*

A publicly verifiable conjunctive keyword search is sound, **iff** *for any adversary $\mathcal{A}$, $\mathcal{A}dv_{\mathcal{A}} \leq \epsilon$ and ϵ is a negligible function in the security parameter κ.*

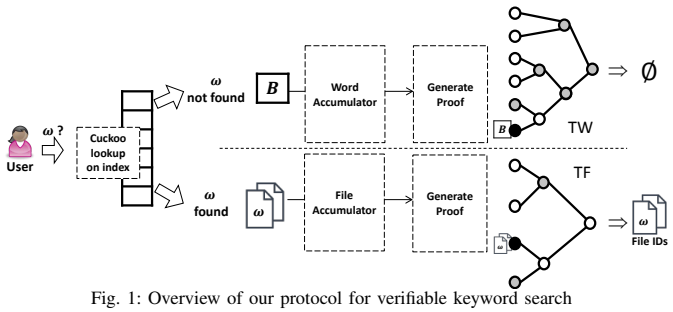

Fig. 1: Overview of our protocol for verifiable keyword search

IV. Building Blocks

Our solution relies on **polynomial-based accumulators** (i.e. bilinear pairing accumulators) defined in [6] and [2] to represent the keywords present in files $\mathcal{F} = \{f_1, f_2, ..., f_n\}$. By definition, a polynomial-based accumulator maps a set to a unique polynomial such that each root of the polynomial corresponds to an element in the set. Hence, polynomial-based accumulators allow efficient *verifiable test of membership* which can be tailored for verifiable keyword search.

A naive approach to accommodate polynomial-based accumulators to verifiable keyword search would be to represent the words in each file $f_j \in \mathcal{F}$ with a single accumulator. To check whether a word ω is in file f_j, user $\mathcal{U}$ first sends a search query to server $\mathcal{S}$, upon which the latter generates a proof of membership if word ω is present in f_j; and a proof of non-membership otherwise. This solution however is not efficient: (i) Given the mathematical properties of polynomial-based accumulators, the resulting complexity of keyword search in a file f_j is linear in the number of keywords in that file; (ii) additionally, to identify which files f_j contain a word, the user must search all files in $\mathcal{F}$ one by one.

To avoid these pitfalls, we combine polynomial-based accumulators with **Merkle trees** [7] to build an authenticated index of the keywords in files in $\mathcal{F}$ such that the keyword search at the server runs in *logarithmic time*. More specifically, data owner $\mathcal{O}$ first organizes the keywords in all files in $\mathcal{F}$ into an index $\mathcal{I}$ (i.e. hash table) where each entry corresponds to a bucket B storing at most d keywords. To construct an efficient index $\mathcal{I}$, data owner $\mathcal{O}$ employs the **Cuckoo hashing** algorithm introduced in [8] which guarantees a constant lookup time and minimal storage requirements. Later, data owner $\mathcal{O}$ authenticates index $\mathcal{I}$ as follows: (i) For each bucket B, it computes an accumulator of the keywords assigned to B; (ii) and it builds a binary Merkle tree TW that authenticates the resulting accumulators. Files in $\mathcal{F}$ are then outsourced together with Merkle tree TW to server $\mathcal{S}$. Hence, when server $\mathcal{S}$ receives a search query for a word ω, it finds the buckets corresponding to ω in index $\mathcal{I}$, retrieves the corresponding accumulator, generates a proof of membership (or non-membership), and authenticates the retrieved accumulator using the Merkle tree TW. Therefore, anyone holding the root of TW can verify the server's response.

The solution sketched above still does not identify which files exactly contain a word ω nor supports verifiable conjunc-

Fig. 2: Verifiable Test of Membership

- $(P_S(h), \Omega_{S,h}) \leftarrow \mathsf{GenerateWitness}(h, S)$

 # Computes the proof of (non-) membership of h with respect to set S.

1) Compute the value $P_S(h) = \prod_{h_i \in S}(h - h_i)$;
2) Determine polynomial $Q_{S,h}$ such that $P_S(X) = (X - h) \cdot Q_{S,h}(X) + P_S(h)$;
3) Compute the witness $\Omega_{S,h} = g^{Q_{S,h}(\alpha)}$;
4) **Return** $(P_S(h), \Omega_{S,h})$;

- $\{h \in S, h \notin S, \mathsf{Reject}\} \leftarrow \mathsf{VerifyMembership}(h, \mathcal{A}cc(S), P_S(h), \Omega_{S,h})$

 # Verifies the proof and outputs the result of the test of membership.

1) Verify $e(\Omega_{S,h}, g^{\alpha} \cdot g^{-h}) e(g^{P_S(h)}, g) \stackrel{?}{=} e(\mathcal{A}cc(S), g)$.
   If it fails **then return** Reject;
2) **If** $P_S(h) = 0$ **then return** $h \in S$ **else return** $h \notin S$;

tive keyword search. Thus, data owner $\mathcal{O}$ constructs another Merkle tree TF whereby each leaf is mapped to a single keyword and associated with the polynomial-based accumulator of the subset of files containing that keyword. Data owner $\mathcal{O}$ then uploads files $\mathcal{F}$ and Merkle trees TW and TF to server $\mathcal{S}$. Given the root of TF, user $\mathcal{U}$ will be able to identify which subset of files contain a word ω. In addition, since polynomial-based accumulators allow efficient *verifiable set intersection*, user $\mathcal{U}$ will also be able to perform verifiable conjunctive keyword search. Figure 1 depicts the steps of the protocol.

A. Symmetric Bilinear Pairings

Let $\mathbb{G}$ and $\mathbb{G}_T$ be two cyclic groups of prime order p. A bilinear pairing is a map $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ that satisfies the following properties: (*Bilinear*) $\forall\ \alpha, \beta \in \mathbb{F}_p$ and $\forall\ g \in \mathbb{G}$, $e(g^{\alpha}, g^{\beta}) = e(g, g)^{\alpha\beta}$; (*Non-degenerate*) If g generates $\mathbb{G}$ then $e(g, g) \neq 1$; (*Computable*) There is an efficient algorithm to compute $e(g, g)$, for any $g \in \mathbb{G}$.

B. Polynomial-based Accumulators

Let $S = \{h_1, ..., h_n\}$ be a set of elements in $\mathbb{F}_p$, encoded by its characteristic polynomial $P_S(X) = \prod_{h_i \in S}(X - h_i)$, and g a random generator of a bilinear group $\mathbb{G}$ of prime order p. Given the public tuple $(g, g^{\alpha}, g^{\alpha^2}, ..., g^{\alpha^D})$, where α is randomly chosen in $\mathbb{F}_p^*$ and $D \geq n$, Nguyen [6] defines the public accumulator of the elements in S:

$$\mathcal{A}cc(S) = g^{P_S(\alpha)} \in \mathbb{G}$$

1) Verifiable Test of Membership: Damgård et al. [2] observe that (i) h is in S iff $P_S(h) = 0$, and (ii) $\forall\ h \in \mathbb{F}_p$, there exists a unique polynomial $Q_{S,h}$ such that $P_S(X) = (X - h) \cdot Q_{S,h}(X) + P_S(h)$. In particular, $\forall\ h$, the accumulator can be written as $\mathcal{A}cc(S) = g^{P_S(\alpha)} = g^{(\alpha - h) \cdot Q_{S,h}(\alpha) + P_S(h)}$. The value $\Omega_{S,h} = g^{Q_{S,h}(\alpha)}$ defines the witness of h with respect to $\mathcal{A}cc(S)$. Following these observations, the authors in [2] define a verifiable test of membership depicted in Figure 2. This test is secure under the D-Strong Diffie-Hellman (D-SDH) assumption.

Definition 3 (D-Strong Diffie-Hellman Assumption)**.** *Let $\mathbb{G}$ be a cyclic group of prime order p generated by g. We say that the D-SDH holds in $\mathbb{G}$ if, given the tuple $(g, g^{\alpha}, g^{\alpha^2}, ..., g^{\alpha^D}) \in \mathbb{G}^{D+1}$, for some randomly chosen $\alpha \in \mathbb{F}_p^*$, no PPT algorithm $\mathcal{A}$ can find a pair $(x, g^{\frac{1}{\alpha + x}}) \in \mathbb{F}_p^* \times \mathbb{G}$ with a non-negligible advantage.*

Fig. 3: Verifiable Set Intersection

- $(I, \Pi_I) \leftarrow \mathsf{ProveIntersection}(S_1, ..., S_k)$

 # Generates the proof for the intersection of the k sets $S_1, ..., S_k$.

1) Compute $I = S_1 \cap ... \cap S_k$ and its characteristic polynomial P;
2) Compute the polynomials $U_i = \frac{P_i}{P}$ and the values $\Delta_i = g^{U_i(\alpha)}$;
3) Compute the polynomials V_i such that $\sum_i U_i V_i = 1$;
4) Compute the values $\Gamma_i = g^{V_i(\alpha)}$;
5) Define $\Pi_I = \{(\Delta_1, \Gamma_1), ..., (\Delta_k, \Gamma_k)\}$;
6) **Return** (I, Π_I).

- $\{\mathsf{Accept}, \mathsf{Reject}\} \leftarrow \mathsf{VerifyIntersection}(I, \Pi_I, \mathcal{A}cc(I), \{\mathcal{A}cc(S_i)\}_{1 \leq i \leq k})$

 # Verifies the proofs for I, the intersection of the sets $S_1, ..., S_k$.

1) Parse $\Pi_I = \{\{\Delta_i, \Gamma_i\}_{1 \leq i \leq k}\}$;
2) Verify the following equalities:
   - $e(\mathcal{A}cc(I), \Delta_i) \stackrel{?}{=} e(\mathcal{A}cc(S_i), g)$ *# Check $I \subseteq S_i$ for $1 \leq i \leq k$*
   - $\prod_i e(\Delta_i, \Gamma_i) \stackrel{?}{=} e(g, g)$ *# Check $\bigcap_i (S_i \setminus I) = \emptyset$*

   If any of the checks fails **then return** Reject **else return** Accept;

2) Verifiable Set Intersection: We consider k sets S_i and their respective characteristic polynomials P_i. If we denote $I = \bigcap_i S_i$ and P the characteristic polynomial of I then $P = \gcd(P_1, P_2, .., P_k)$. It follows that the k polynomials $U_i = \frac{P_i}{P}$ identify the sets $S_i \setminus I$. Since $\bigcap_i (S_i \setminus I) = \emptyset$, $\gcd(U_1, U_2, ..., U_k) = 1$. Therefore, according to Bézout's identity, there exist polynomials V_i such that $\sum_i U_i V_i = 1$. Based on these observations, Canetti et al. [3] define a protocol for verifiable set intersection described in Figure 3. The intersection verification is secure if the D-Strong Bilinear Diffie-Hellman (D-SBDH) assumption holds.

Definition 4 (Strong Bilinear Diffie-Hellman Assumption)**.** *Let $\mathbb{G}, \mathbb{G}_T$ be cyclic groups of prime order p, g a generator of $\mathbb{G}$, and e a bilinear pairing. We say that the D-SBDH holds if, given $(g, g^{\alpha}, g^{\alpha^2}, ..., g^{\alpha^D}) \in \mathbb{G}^{D+1}$, for some randomly chosen $\alpha \in \mathbb{F}_p^*$, no PPT algorithm $\mathcal{A}$ can find a pair $(x, e(g, g)^{\frac{1}{\alpha + x}}) \in \mathbb{F}_p^* \times \mathbb{G}_T$ with a non-negligible advantage.*

C. Cuckoo Hashing

Cuckoo hashing belongs to the multiple choice hashing techniques. In the seminal work [9], an object can be stored in one of the two possible buckets of an index. If both buckets are full, an object is "kicked out" from one of these two buckets, the current item is placed in the freed bucket and the removed item is moved to the other bucket of its two choices. This move may require another element to be kicked out from its location. This insertion procedure is repeated until all objects find a free spot, or the number of insertion attempts reaches a predefined threshold to declare an insertion failure. In this paper, we leverage a variant proposed by Dietzfelbinger and Weidling [8]: Their solution inserts N elements using two independent and fully random hash functions $\mathcal{H}_1, \mathcal{H}_2 : \{0,1\}^* \rightarrow [1, m]$ into an index $\mathcal{I}$ with m buckets B_i, such that: $m = \frac{1+\varepsilon}{d} N$, for $\varepsilon > 0$, and each bucket B_i stores at most d elements. As depicted in Figure 4, a lookup operation for a particular element x requires the evaluation of the two hash functions $\mathcal{H}_1(x)$ and $\mathcal{H}_2(x)$, whereas the insertion of a new element requires a random walk in the index.

D. Binary Merkle Trees

Merkle trees allow any third party to verify whether an element h is in set $S = \{h_1, ..., h_n\}$. In the following, we

Fig. 4: Cuckoo Hashing

• $\mathsf{CuckooInsert}(\mathcal{I}, \mathcal{H}_1, \mathcal{H}_2, x)$
 # Inserts x in index $\mathcal{I}$ using hash functions $\mathcal{H}_1, \mathcal{H}_2 : \{0,1\}^* \rightarrow [1, m]$.
1) Compute $i_1 = \mathcal{H}_1(x)$ and $i_2 = \mathcal{H}_2(x)$;
2) **If** bucket B_{i_1} is not full **then**
   Insert x in B_{i_1};
   Return;
   End
3) **If** bucket B_{i_2} is not full **then**
   Insert x in B_{i_2};
   Return;
   End
4) **If** buckets B_{i_1} and B_{i_2} both full **then**
   Randomly choose y from the $2d$ elements in $B_{i_1} \cup B_{i_2}$;
   Remove y;
   $\mathsf{CuckooInsert}(\mathcal{I}, \mathcal{H}_1, \mathcal{H}_2, x)$;
   $\mathsf{CuckooInsert}(\mathcal{I}, \mathcal{H}_1, \mathcal{H}_2, y)$;
   Return;
   End

• $\{\mathsf{true}, \mathsf{false}\} \leftarrow \mathsf{CuckooLookup}(\mathcal{I}, \mathcal{H}_1, \mathcal{H}_2, x)$
 # Searches for x in index $\mathcal{I}$.
1) Compute $i_1 = \mathcal{H}_1(x)$ and $i_2 = \mathcal{H}_2(x)$;
2) **Return** $(x \in B_{i_1}) \vee (x \in B_{i_2})$;

introduce the algorithms that build a binary Merkle tree for a set S and authenticate the elements in that set.

• $\mathsf{T} \leftarrow \mathsf{BuildMT}(S, H)$ builds a binary Merkle tree T as follows. Each leaf L_i of the tree maps an element h_i in set S and each internal node stores the hash of the concatenation of the children of that node. We denote σ the root of T.

• $\mathsf{path} \leftarrow \mathsf{GenerateMTProof}(\mathsf{T}, h)$ outputs the *authentication path*, denoted path, for leaf L corresponding to element h, that is, the set of the siblings of the nodes on the path from L to root σ.

• $\{\mathsf{Accept}, \mathsf{Reject}\} \leftarrow \mathsf{VerifyMTProof}(h, \mathsf{path}, \sigma)$ verifies that the root value computed from h and path equals the expected value σ.

V. Protocol Description

In our verifiable conjunctive keyword search protocol, data owner O outsources the storage of a set of files $\mathcal{F} = \{f_1, f_2, ..., f_n\}$ to a server $\mathcal{S}$. Once the data is uploaded, any third-party user $\mathcal{U}$ can search for some keywords in the set of files $\mathcal{F}$ and verify the correctness of the search results returned by $\mathcal{S}$. The proposed protocol comprises two phases: **Upload** and **Verifiable Conjunctive Keyword Search**.

A. Upload

In this phase, data owner O invokes algorithm Setup, which on input of security parameter κ and set of files $\mathcal{F}$, outputs a public key $\mathsf{PK}_\mathcal{F}$ and a search key $\mathsf{LK}_\mathcal{F}$. As shown in Figure 5, Setup operates in four steps.

1) It first generates the public parameters of the protocol.
2) It builds index $\mathcal{I}$ for the set $\mathbb{W} = \{\omega_1, \omega_2, ..., \omega_N\}$ using Cuckoo hashing. Without loss of generality, we assume that $\mathbb{W}$ is composed of the list of distinct words in $\mathcal{F}$ sorted in a lexicographic order.
3) Setup authenticates index $\mathcal{I}$ with Merkle tree TW where each leaf is mapped to a bucket in $\mathcal{I}$. We denote σ_W the root of TW.
4) Setup builds Merkle tree TF, with root σ_F, to identify which files exactly contain the keywords.

Fig. 5: Upload

• $(\mathsf{PK}_\mathcal{F}, \mathsf{LK}_\mathcal{F}) \leftarrow \mathsf{Setup}(1^\kappa, \mathcal{F})$
 # $\mathcal{F} = \{f_1, ..., f_n\}$: set of files
 # $\mathbb{W} = \{\omega_1, .., \omega_N\}$: list of distinct words in $\mathcal{F}$ sorted in lexicographic order.
1) **Parameter generation**
   Pick $D, g, \mathbb{G}, \mathbb{G}_T, e, H : \{0,1\}^* \rightarrow \mathbb{F}_p$ as function of security parameter κ;
   Pick random $\alpha \in \mathbb{F}_p^*$ and compute public values $\{g, g^\alpha, ..., g^{\alpha^D}\}$;
2) **Construction of the Index**
   # Creates an index $\mathcal{I}$ with m buckets of size d where $d < D$
   Identify $\mathbb{W}$ from $\mathcal{F}$;
   Pick random hash functions $\mathcal{H}_1, \mathcal{H}_2 : \{0,1\}^* \rightarrow [1, m]$;
   For $\omega_i \in \mathbb{W}$ **do**
   Compute $h_i = H(\omega_i)$;
   Run $\mathsf{CuckooInsert}(\mathcal{I}, \mathcal{H}_1, \mathcal{H}_2, h_i)$;
   End
3) **Authentication of Index**
   For $B_i \in \mathcal{I}$ **do**
   Compute $P_{B_i}(\alpha) = \prod_{h_j \in B_i}(\alpha - h_j)$;
   Compute $\mathsf{AW}_i = \mathcal{A}cc(B_i) = g^{P_{B_i}(\alpha)}$;
   Compute $\mathsf{HW}_i = H(\mathsf{AW}_i || i)$, where i is the position of B_i in $\mathcal{I}$;
   End
   $\mathsf{TW} = \mathsf{BuildMT}(\{\mathsf{HW}_i\}_{1 \leq i \leq m}, H)$;
4) **Encoding of files**
   # Identifies which files contain the keywords
   For $f_j \in \mathcal{F}$ **do**
   Generate fid_j;
   End
   For $\omega_i \in \mathbb{W}$ **do**
   Identify $\mathcal{F}_{\omega_i}$, the subset of files that contain ω_i;
   Compute $P_i(\alpha) = \prod_{\mathsf{fid}_j \in \mathcal{F}_{\omega_i}}(\alpha - \mathsf{fid}_j)$;
   Compute $\mathsf{AF}_i = \mathcal{A}cc(\mathcal{F}_{\omega_i}) = g^{P_i(\alpha)}$;
   Compute $\mathsf{HF}_i = H(\mathsf{AF}_i || \omega_i)$;
   End
   $\mathsf{TF} = \mathsf{BuildMT}(\{\mathsf{HF}_i\}_{1 \leq i \leq N}, H)$.
5) **Return** $\mathsf{PK}_\mathcal{F} = (g, \mathbb{G}, e, H, \{g^{\alpha^i}\}_{0 \leq i \leq D}, \mathcal{H}_1, \mathcal{H}_2, \sigma_W, \sigma_F)$;
   Return $\mathsf{LK}_\mathcal{F} = (\mathcal{I}, \mathsf{TW}, \mathsf{TF}, \mathcal{F}, \mathbb{W}, \{\mathcal{F}_{\omega_i}\}_{1 \leq i \leq N})$.

When server S receives $\mathsf{LK}_\mathcal{F}$, it creates a hash table HT where each entry is mapped to a keyword ω_i and stores the pair $(i, \mathsf{pointer})$ such that: i is the position of keyword ω_i in set $\mathbb{W}$ and in tree TF; whereas pointer points to a linked list storing the identifiers of files $\mathcal{F}_{\omega_i}$ that contain keyword ω_i. As such, hash table HT enables server $\mathcal{S}$ to find the position of ω_i in TF and to identify the files containing ω_i easily.

In the remainder of this paper, we assume that server $\mathcal{S}$ does not store $\mathsf{LK}_\mathcal{F}$ as $(\mathcal{I}, \mathsf{TW}, \mathsf{TF}, \mathcal{F}, \mathbb{W}, \{\mathcal{F}_{\omega_i}\}_{1 \leq i \leq N})$, but rather as $\mathsf{LK}_\mathcal{F} = (\mathcal{I}, \mathsf{TW}, \mathsf{TF}, \mathcal{F}, \mathsf{HT})$.

B. Verifiable Conjunctive Keyword Search

In this phase, we use the algorithms of verifiable test of membership and verifiable set intersection presented in Section IV to enable verifiable conjunctive keyword search. We assume in what follows that a user $\mathcal{U}$ wants to identify the set of files $\mathcal{F}_\mathcal{W} \subset \mathcal{F}$ that contain all words in $\mathcal{W} = \{\omega_1, \omega_2, ..., \omega_k\}$. To that effect, user $\mathcal{U}$ first runs algorithm QueryGen (cf. Figure 6) which returns the query $\mathcal{E}_Q = \mathcal{W}$ and the public verification key $\mathsf{VK}_Q = (\mathsf{PK}_\mathcal{F}, \mathcal{W})$. User $\mathcal{U}$ then sends query $\mathcal{E}_Q$ to server $\mathcal{S}$.

On receipt of query $\mathcal{E}_Q$ server $\mathcal{S}$ invokes algorithm Search (cf. Figure 6) which searches the index $\mathcal{I}$ for every individual keyword $\omega_i \in \mathcal{W}$. If all the keywords $\omega_i \in \mathcal{W}$ are found in the index, then Search identifies the subset of files $\mathcal{F}_{\omega_i}$ that contains ω_i and outputs the intersection of all these subsets $\mathcal{F}_\mathcal{W} = \mathcal{F}_{\omega_1} \cap ... \cap \mathcal{F}_{\omega_k}$. Moreover, to prove the correctness of the response (i.e. to prove that $\mathcal{F}_\mathcal{W}$ was computed correctly),

Fig. 6: Verifiable Conjunctive Keyword Search

- $\{\mathcal{E}_Q, \mathsf{VK}_Q\} \leftarrow \mathsf{QueryGen}(\mathcal{W}, \mathsf{PK}_\mathcal{F})$
  1) Assign $\mathcal{E}_Q = \mathcal{W}$ and $\mathsf{VK}_Q = (\mathsf{PK}_\mathcal{F}, \mathcal{W})$;
  2) **Return** $\{\mathcal{E}_Q, \mathsf{VK}_Q\}$;

- $\mathcal{E}_R \leftarrow \mathsf{Search}(\mathcal{E}_Q, \mathsf{LK}_\mathcal{F})$
  1) Parse $\mathcal{E}_Q = \mathcal{W}$ and $\mathsf{LK}_\mathcal{F} = (\mathcal{I}, \mathsf{TW}, \mathsf{TF}, \mathcal{F}, \mathsf{HT})$;
  2) **For** $\omega_i \in \mathcal{W}$ **do**
     Compute $h_i = H(\omega_i)$;
     If $\mathsf{CuckooLookup}(\mathcal{I}, \mathcal{H}_1, \mathcal{H}_2, h_i) = \mathsf{false}$ **then**
     # Keyword ω_i is not in $\mathcal{F}$
     Compute $i_1 = \mathcal{H}_1(h_i)$ and $i_2 = \mathcal{H}_2(h_i)$;
     Compute $\Pi_1 = \mathsf{GenerateWitness}(h_i, B_{i_1})$;
     Compute $\Pi_2 = \mathsf{GenerateWitness}(h_i, B_{i_2})$;
     Compute $\mathsf{AW}_{i_1} = \mathcal{A}cc(B_{i_1})$ and $\mathsf{HW}_{i_1} = H(\mathsf{AW}_{i_1}||i_1)$;
     Compute $\mathsf{AW}_{i_2} = \mathcal{A}cc(B_{i_2})$ and $\mathsf{HW}_{i_2} = H(\mathsf{AW}_{i_2}||i_2)$;
     Compute $\mathsf{path}_1 = \mathsf{GenerateMTProof}(\mathsf{TW}, \mathsf{HW}_{i_1})$;
     Compute $\mathsf{path}_2 = \mathsf{GenerateMTProof}(\mathsf{TW}, \mathsf{HW}_{i_2})$;
     Return $\mathcal{E}_R = (\emptyset, \omega, \mathsf{AW}_{i_1}, \mathsf{AW}_{i_2}, \Pi_1, \Pi_2, \mathsf{path}_1, \mathsf{path}_2)$;
     End
     End
  3) *# All the keywords have been found*
     For $\omega_i \in \mathcal{W}$ **do**
     Determine $\mathcal{F}_{\omega_i}$ using HT; *# the set of files that contain w_i*
     Compute $\mathsf{AF}_i = \mathcal{A}cc(\mathcal{F}_{\omega_i})$ and $\mathsf{HF}_i = H(\mathsf{AF}_i||\omega_i)$;
     Determine position l of w_i in TF using HT;
     # HF_i is in the l^{th} leaf of TF
     Compute $\mathsf{path}_i = \mathsf{GenerateMTProof}(\mathsf{TF}, \mathsf{HF}_i)$;
     End
     # $\mathcal{F}_\mathcal{W} = \mathcal{F}_{\omega_1} \cap ... \cap \mathcal{F}_{\omega_k}$ is the set of files that contain all the words in $\mathcal{W}$
     Compute $(\mathcal{F}_\mathcal{W}, \Pi_\mathcal{W}) = \mathsf{ProveIntersection}(\mathcal{F}_{\omega_1}, ..., \mathcal{F}_{\omega_k})$;
     Return $\mathcal{E}_R = (\mathcal{F}_\mathcal{W}, \Pi_\mathcal{W}, \{\mathsf{AF}_i\}_{1\le i\le k}, \{\mathsf{path}_i\}_{1\le i\le k})$;

- $\mathsf{out} \leftarrow \mathsf{Verify}(\mathcal{E}_R, \mathsf{VK}_Q)$
  1) Parse $\mathsf{VK}_Q = (\mathsf{PK}_\mathcal{F}, \mathcal{W})$;
  2) **If** $\mathcal{W}$ found in $\mathcal{F}$ **then**
     Parse $\mathcal{E}_R = (\mathcal{F}_\mathcal{W}, \Pi_\mathcal{W}, \{\mathsf{AF}_i\}_{1\le i\le k}, \{\mathsf{path}_i\}_{1\le i\le k})$;
     For $\omega_i \in \mathcal{W}$ **do**
     If $\mathsf{VerifyMTProof}(H(\mathsf{AF}_i||\omega_i), \mathsf{path}_i, \sigma_F) = \mathsf{Reject}$
     Then return $\mathsf{out} = \perp$;
     End
     Compute $\mathcal{A}cc(\mathcal{F}_\mathcal{W})$;
     If $\mathsf{VerifyIntersection}(\mathcal{F}_\mathcal{W}, \Pi_\mathcal{W}, \mathcal{A}cc(\mathcal{F}_\mathcal{W}), \{\mathsf{AF}_i\}_{1\le i\le k}) = \mathsf{Accept}$;
     Then return $\mathsf{out} = \mathcal{F}_\mathcal{W}$ **else return** $\mathsf{out} = \perp$;
     End
  3) **If** at least one keyword ω_i is not found in $\mathcal{F}$ **then**
     Parse $\mathcal{E}_R = (\emptyset, \omega_i, \mathsf{AW}_{i_1}, \mathsf{AW}_{i_2}, \Pi_1, \Pi_2, \mathsf{path}_1, \mathsf{path}_2)$;
     Compute $h_i = H(\omega_i)$, $i_1 = \mathcal{H}_1(h_i)$ and $i_2 = \mathcal{H}_2(h_i)$;
     If $\mathsf{VerifyMTProof}(H(\mathsf{AW}_{i_1}||i_1), \mathsf{path}_1, \sigma_W) = \mathsf{Reject}$
     Then return $\mathsf{out} = \perp$;
     If $\mathsf{VerifyMTProof}(H(\mathsf{AW}_{i_2}||i_2), \mathsf{path}_2, \sigma_W) = \mathsf{Reject}$
     Then return $\mathsf{out} = \perp$;
     If $\mathsf{VerifyMembership}(h_i, \mathsf{AW}_{i_1}, \Pi_1) = \mathsf{Reject}$
     Then return $\mathsf{out} = \perp$;
     If $\mathsf{VerifyMembership}(h_i, \mathsf{AW}_{i_2}, \Pi_2) = \mathsf{Reject}$
     Then return $\mathsf{out} = \perp$;
     Return $\mathsf{out} = \emptyset$;
     End

Search (i) authenticates the accumulators of each set $\mathcal{F}_{\omega_i}$ using Merkle tree TF; (ii) and generates a proof of intersection for $\mathcal{F}_\mathcal{W}$ using the verification algorithm described in Figure 3.

If at least one keyword ω_i is not found, then Search returns ω_i and an empty set, and proves the correctness of its response by (i) authenticating the accumulators of buckets B_{i_1} and B_{i_2} associated with ω_i in index $\mathcal{I}$ using Merkle tree TW; (ii) and generating a proof of non-membership of keyword ω_i for buckets B_{i_1} and B_{i_2} (cf. Figure 2).

On reception of the search result, verifier $\mathcal{V}$ checks the correctness of the server's response by calling algorithm Verify as shown in Figure 6. More precisely, if server $\mathcal{S}$ advertises that it has found all the keywords $\mathcal{W}$ in index $\mathcal{I}$, then algorithm Verify checks that the returned intersection $\mathcal{F}_\mathcal{W}$ is correct using the verification algorithm of Merkle tree and verifiable set intersection. Otherwise, $\mathcal{V}$ verifies that the returned keyword is actually not in $\mathcal{F}$ using again the verification algorithm of Merkle tree and verifiable test of membership.

VI. Security Analysis

Our protocol satisfies the two security properties of correctness and soundness.

Theorem 1 (Correctness)**.** *Our scheme is a correct verifiable conjunctive keyword search solution.*

Proof: Suppose that a user $\mathcal{U}$ sends to server $\mathcal{S}$ the query $\mathcal{E}_Q = \mathcal{W} = \{\omega_1, ..., \omega_k\}$. $\mathcal{S}$ correctly runs algorithm Search and returns the search response $\mathcal{E}_R$. According to Figure 6, the content of $\mathcal{E}_R$ varies depending on whether:

a) All words in $\mathcal{W}$ are found in $\mathcal{F}$:

Then $\mathcal{E}_R = (\mathcal{F}_\mathcal{W}, \Pi_\mathcal{W}, \{\mathsf{AF}_i\}_{1\le i\le k}, \{\mathsf{path}_i\}_{1\le i\le k})$ where:

- $\mathcal{F}_\mathcal{W} = \mathcal{F}_{\omega_1} \cap ... \cap \mathcal{F}_{\omega_k}$ such that $\mathcal{F}_{\omega_i}$ is the subset of files that contain keyword ω_i;
- $\Pi_\mathcal{W} = \{(\Delta_1, \Gamma_1), ..., (\Delta_k, \Gamma_k)\}$ is the proof of intersection;
- for all $1 \le i \le k$, $\mathsf{AF}_i = \mathcal{A}cc(\mathcal{F}_{\omega_i})$; if we denote P_i the characteristic polynomial of $\mathcal{F}_{\omega_i}$, then $\mathsf{AF}_i = g^{P_i(\alpha)}$;
- for all $1 \le i \le k$, path_i is the authentication path of $H(\mathsf{AF}_i||\omega_i)$ in TF.

Firstly, if we assume that the Merkle tree authentication is correct, then verifier $\mathcal{V}$ accepts the accumulators AF_i computed by server $\mathcal{S}$. Secondly, since $\mathcal{S}$ computes the proof $\Pi_\mathcal{W}$ using algorithm ProveIntersection (cf. Figure 3), for all $1 \le i \le k$, we have the following:

- $\Delta_i = g^{U_i(\alpha)}$, where $U_i = \frac{P_i}{P}$ and $P = \gcd(P_1, P_2, ..., P_k)$ is the characteristic polynomial of $\mathcal{F}_\mathcal{W}$;
- $\Gamma_i = g^{V_i(\alpha)}$, such that $\sum_i U_i V_i = 1$.

It follows that for all $1 \le i \le k$:

$$e(\mathcal{A}cc(\mathcal{F}_\mathcal{W}), \Delta_i) = e(g^{P(\alpha)}, g^{U_i(\alpha)}) = e(g,g)^{P(\alpha)\cdot U_i(\alpha)} = e(g,g)^{P_i(\alpha)} = e(\mathsf{AF}_i, g)$$

This means that the first equality in algorithm VerifyIntersection (cf. Figure 3) holds. Furthermore, the second equality is also verified, indeed:

$$\prod_{\omega_i \in \mathcal{W}} e(\Delta_i, \Gamma_i) = \prod_{\omega_i \in \mathcal{W}} e(g^{U_i(\alpha)}, g^{V_i(\alpha)}) = \prod_{\omega_i \in \mathcal{W}} e(g,g)^{U_i(\alpha)\cdot V_i(\alpha)} = e(g,g)^{\sum_{\omega_i \in \mathcal{W}} U_i(\alpha)\cdot V_i(\alpha)} = e(g,g)$$

These computations thus prove the correctness of our solution in the case where the targeted keywords are all found.

b) There exists $\omega_i \in \mathcal{W}$ not found in $\mathcal{F}$:

Here, $\mathcal{E}_R = (\emptyset, \omega_i, \mathsf{AW}_{i_1}, \mathsf{AW}_{i_2}, \Pi_1, \Pi_2, \mathsf{path}_1, \mathsf{path}_2)$ where:

- $\mathsf{AW}_{i_1} = \mathcal{A}cc(B_{i_1})$ and $\mathsf{AW}_{i_2} = \mathcal{A}cc(B_{i_2})$ are the accumulators of buckets B_{i_1} and B_{i_2} respectively, where i_1 and i_2 are the positions assigned to keyword ω_i in index $\mathcal{I}$;
- Π_1 and Π_2 are the proofs that ω_i is not a member of bucket B_{i_1} nor of bucket B_{i_2} respectively;
- path_1 and path_2 are the authentication paths of these two buckets in tree TW.

If we consider the Merkle tree to be correct, then verifier $\mathcal{V}$ accepts AW_{i_1} and AW_{i_2}. Moreover, if we denote $P_{B_{i_1}}$ the characteristic polynomial of B_{i_1}, then by definition $P_{B_{i_1}}(X) = \prod_{h_j \in B_{i_1}} (X - h_j)$ and $\mathsf{AW}_{i_1} = \mathcal{A}cc(B_{i_1}) = g^{P_{B_{i_1}}(\alpha)}$.

Recall now that the proof of non-membership Π_1 of keyword ω_i to bucket B_{i_1} is computed as: $\{P_{B_{i_1}}(h_i), \Omega_{B_{i_1},h_i}\}$, such that $h_i = H(\omega_i)$, $\Omega_{B_{i_1},h_i} = g^{Q_{B_{i_1},h_i}(\alpha)}$ and $Q_{B_{i_1},h_i}(X) = \frac{P_{B_{i_1}}(X) - P_{B_{i_1}}(h_i)}{X - h_i}$. It follows that:

$$\begin{aligned} & e(\Omega_{B_{i_1}}, g^{\alpha} \cdot g^{-h_i}) e(g^{P_{B_{i_1}}(h_i)}, g) \\ & \quad = e(g,g)^{Q_{B_{i_1},h_i}(\alpha)\cdot(\alpha - h_i)} e(g,g)^{P_{B_{i_1}}(h_i)} \\ & \quad = e(g,g)^{Q_{B_{i_1},h_i}(\alpha)\cdot(\alpha - h_i) + P_{B_{i_1}}(h_i)} \\ & \quad = e(g,g)^{P_{B_{i_1}}(\alpha)} \\ & \quad = e(\mathsf{AW}_{i_1}, g). \end{aligned}$$

This means that the first equality of algorithm GenerateWitness (cf. Figure 2) holds. Finally, since $\omega_i \notin B_{i_1}$, $P_{B_{i_1}}(h_i) \neq 0$. This implies that verifier $\mathcal{V}$ accepts the proof of non-membership for bucket B_{i_1} and concludes that $\omega_i \notin \mathcal{F}$.

Same computations can be performed for B_{i_2}, which proves the correctness of our solution when a keyword $\omega_i \notin \mathcal{F}$. ■

Theorem 2 (Soundness)**.** *Our solution for verifiable conjunctive keyword search is sound under the D-SDH and D-SBDH assumptions, provided that the hash function H used to build the Merkle trees is collision-resistant.*

Proof Sketch: Space limitations allow us to outline only a sketch of this proof. We leave out the details in the full paper[2].

We observe that an adversary can break the soundness of our scheme through two types of forgeries:

Type 1 forgery: On input of $\mathcal{W} = \{\omega_1, ..., \omega_k\}$ and search key $\mathsf{LK}_{\mathcal{F}}$, adversary $\mathcal{A}_1$ returns a search result that consists of a proof of non-membership of some keyword $\omega_i \in \mathcal{W}$ (meaning that ω_i is not in the set of files $\mathcal{F}$), although ω_i is in $\mathcal{F}$;

Type 2 forgery: On input of $\mathcal{W} = \{\omega_1, ..., \omega_k\}$ and search key $\mathsf{LK}_{\mathcal{F}}$, adversary $\mathcal{A}_2$ returns an incorrect $\widehat{\mathcal{F}}_{\mathcal{W}}$ and the corresponding proof[3]. This means that adversary $\mathcal{A}_2$ claims that all keywords in $\mathcal{W}$ have been found in $\mathcal{F}$ and that $\widehat{\mathcal{F}}_{\mathcal{W}}$ is the subset of files that contain them, although $\widehat{\mathcal{F}}_{\mathcal{W}} \neq \mathsf{CKS}(\mathcal{F}, \mathcal{W})$.

The proof consists in showing that if $\mathcal{A}_1$ and $\mathcal{A}_2$ run Type 1 and Type 2 forgeries respectively, then there exist an adversary $\mathcal{B}_1$ that breaks D-SDH and an adversary $\mathcal{B}_2$ that breaks D-SBDH.

Reduction of Type 1 forgery to D-SDH problem:

We define the oracle $\mathcal{O}_{\mathsf{D\text{-}SDH}}$ which, when invoked, returns the D-SDH tuple $T(\alpha) = (g, g^{\alpha}, g^{\alpha^2}, ..., g^{\alpha^D}) \in \mathbb{G}^{D+1}$, for some randomly selected $\alpha \in \mathbb{F}_p^*$. Adversary $\mathcal{B}_1$, who wants to break the D-SDH assumption, first calls $\mathcal{O}_{\mathsf{D\text{-}SDH}}$ which selects a random $\alpha \in \mathbb{F}_p^*$ and returns $T(\alpha)$; and then, $\mathcal{B}_1$ simulates the soundness game for adversary $\mathcal{A}_1$ (cf. Algorithm 1). Namely:

- $\mathcal{A}_1$ invokes $\mathcal{O}_{\mathsf{Setup}}$ with the sets of files $\mathcal{F}_i$ (for $1 \leq i \leq t$), and in turn, $\mathcal{B}_1$ simulates $\mathcal{O}_{\mathsf{Setup}}$ and generates $(\mathsf{PK}_{\mathcal{F}_i}, \mathsf{LK}_{\mathcal{F}_i})$. The key idea behind the proof is that $\mathsf{PK}_{\mathcal{F}_i}$ includes the tuple $T_i(\alpha) = (g, g^{\alpha_i}, g^{\alpha_i^2}, ..., g^{\alpha_i^D})$ where $\alpha_i = \alpha \cdot \delta_i + \beta_i$ for some random $\delta_i, \beta_i \in \mathbb{F}_p^*$. Note that this tuple can be easily computed by $\mathcal{B}_1$, without having access to α, thanks to $T(\alpha)$ and the Binomial Theorem: $\forall\ k \leq D,\ g^{\alpha_i{}^k} = g^{(\alpha\cdot\delta_i + \beta_i)^k} = \prod_{j=0}^{k} (g^{\alpha^j})^{\binom{k}{j}(\delta_i)^j \cdot \beta_i^{k-j}}$.
- Later, $\mathcal{A}_1$ selects a public key $\mathsf{PK}^*_{\mathcal{F}}$ from the keys he has received earlier, and a collection of keywords $\mathcal{W}^*$ to search for in the set of files $\mathcal{F}^*$ associated with $\mathsf{PK}^*_{\mathcal{F}}$.
- $\mathcal{A}_1$ runs $\mathsf{QueryGen}(\mathcal{W}^*, \mathsf{PK}^*_{\mathcal{F}})$ which yields $\mathcal{E}^*_Q$ and VK^*_Q.
- $\mathcal{A}_1$ then returns $\mathcal{E}^*_R = (\emptyset, \omega^*, \widehat{\mathsf{AF}}_1^*, \widehat{\mathsf{AF}}_2^*, \widehat{\Pi}_1^*, \widehat{\Pi}_2^*, \widehat{\mathsf{path}}_1^*, \widehat{\mathsf{path}}_2^*)$ and calls Verify on input of $\mathcal{E}^*_R$ and VK^*_Q.

Since we assume H is a collision-resistant hash function, the Merkle tree authentication proves that the accumulators returned in $\mathcal{E}^*_R$ are actually the ones computed by $\mathcal{B}_1$ when he was simulating Setup. The rest of the proof follows a similar reasoning proposed by Damgård and Triandopoulos [2]. Accordingly, if Verify accepts the proof of non-membership for keyword ω^*, then $\mathcal{B}_1$ finds a pair $(x, g^{\frac{1}{\alpha+x}})$ that breaks the D-SDH assumption with $x = \frac{\beta^* - h^*}{\delta^*}$ where $h^* = H(\omega^*)$ and α^*, β^*, δ^* are values associated with set of files $\mathcal{F}^*$, such that $\alpha^* = \alpha \cdot \delta^* + \beta^*$.

Reduction of Type 2 forgery to D-SBDH problem:

Let $\mathcal{O}_{\mathsf{D\text{-}SBDH}}$ be an oracle that returns for any random $\alpha \in \mathbb{F}_p^*$, the tuple $T(\alpha) = (g, g^{\alpha}, g^{\alpha^2}, ..., g^{\alpha^D}) \in \mathbb{G}^{D+1}$. Adversary $\mathcal{B}_2$, who wants to break the D-SBDH assumption, calls $\mathcal{O}_{\mathsf{D\text{-}SBDH}}$ which selects a random $\alpha \in \mathbb{F}_p^*$ and returns $T(\alpha)$. Afterwards, $\mathcal{B}_2$ simulates the soundness game for adversary $\mathcal{A}_2$ (cf. Algorithm 1). Specifically:

- $\mathcal{A}_2$ invokes $\mathcal{O}_{\mathsf{Setup}}$ with the sets of files $\mathcal{F}_i$ (for $1 \leq i \leq t$), and in turn, $\mathcal{B}_2$ simulates $\mathcal{O}_{\mathsf{Setup}}$ and generates $(\mathsf{PK}_{\mathcal{F}_i}, \mathsf{LK}_{\mathcal{F}_i})$. Similar to *Type 1 forgery*, $\mathsf{PK}_{\mathcal{F}_i}$ includes the tuple $T_i(\alpha) = (g, g^{\alpha_i}, g^{\alpha_i^2}, ..., g^{\alpha_i^D})$ where $\alpha_i = \alpha \cdot \delta_i + \beta_i$ for some random $\delta_i, \beta_i \in \mathbb{F}_p^*$.
- Then, $\mathcal{A}_2$ selects a public key $\mathsf{PK}^*_{\mathcal{F}}$ from the keys he has received earlier, and a collection of keywords $\mathcal{W}^*$ to search for in the set of files $\mathcal{F}^*$ associated with $\mathsf{PK}^*_{\mathcal{F}}$.
- $\mathcal{A}_2$ further invokes $\mathsf{QueryGen}(\mathcal{W}^*, \mathsf{PK}^*_{\mathcal{F}})$ which returns $\mathcal{E}^*_Q$ and VK^*_Q.
- $\mathcal{A}_2$ then outputs $\mathcal{E}^*_R = (\widehat{\mathcal{F}}_{\mathcal{W}^*}, \widehat{\Pi}_{\mathcal{W}^*}, \{\widehat{\mathsf{AF}}_i^*\}_{1\leq i\leq k}, \{\widehat{\mathsf{path}}_i^*\}_{1\leq i\leq k})$. It further runs Verify on input of $\mathcal{E}^*_R$ and VK^*_Q.

In this case since we also assume that H is a collision-resistant hash function, the Merkle tree authentication proves that the accumulators returned in $\mathcal{E}^*_R$ are actually the ones computed by $\mathcal{B}_2$ when he was simulating Setup. Given these accumulators, the remainder of the proof adopts an approach similar to the one proposed by Canetti et al. in [3]. Indeed, since $\widehat{\mathcal{F}}_{\mathcal{W}^*} \neq \mathsf{CKS}(\mathcal{F}^*, \mathcal{W}^*)$, either $\widehat{\mathcal{F}}_{\mathcal{W}^*}$ contains a file with identifier fid^* that is not in $\mathsf{CKS}(\mathcal{F}^*, \mathcal{W}^*)$, or there is a file with identifier fid^* that is in $\mathsf{CKS}(\mathcal{F}^*, \mathcal{W}^*)$ but not in $\widehat{\mathcal{F}}_{\mathcal{W}^*}$. If Verify accepts the proof of intersection, then $\mathcal{B}_2$ breaks D-SBDH by outputting a pair $(x, e(g,g)^{\frac{1}{\alpha+x}})$ where $x = \frac{\beta^* - \mathsf{fid}^*}{\delta^*}$ with α^*, β^* and δ^* are values associated with $\mathcal{F}^*$ such that $\alpha^* = \alpha \cdot \delta^* + \beta^*$. ■

[2]The full paper can be found here: http://www.eurecom.fr/publication/4540.

[3]The caret notation ($\widehat{\cdot}$) distinguishes the elements of the response returned by an adversary from the ones that would be returned by a honest server.

VII. Performance Evaluation

In light of the performances of the several building blocks (Cuckoo hashing, polynomial-based accumulators and Merkle trees), we analyze in the following the computational costs of our solution. A summary[4] of this analysis is provided in Table I, together with all notations.

1. Setup: As mentioned in Section II, the setup phase of our protocol is a one-time pre-processing operation that is amortized over an unlimited number of fast verifications. The computational cost of this phase is dominated by:

- The public parameter generation which amounts to D exponentiations in $\mathbb{G}$;
- N calls to CuckooInsert where, as shown in [8], each insertion is expected to terminate in $(1/\varepsilon)^{\mathcal{O}(\log d)}$ time ($\varepsilon > 0$);
- The computation of m accumulators AW which requires m exponentiations in $\mathbb{G}$ and md multiplications in $\mathbb{F}_p$;
- The computation of N accumulators AF which involves N exponentiations in $\mathbb{G}$ and Nn multiplications in $\mathbb{F}_p$;
- The generation of Merkle tree TW (resp. TF) which consists of $2m$ hashes (resp. $2N$).

2. QueryGen: This algorithm does not require any computation. It only constructs the query for the k keywords together with the corresponding VK_Q.

3. Search: Although this algorithm seems expensive, we highlight the fact that it is executed by the cloud server. Search runs k CuckooLookup which consist in $2k$ hashes and $2kd$ comparisons to search for all the k queried keywords (in the worst case). Following this operation, the complexity of this phase depends on whether all the keywords have been found:

- out $= \mathcal{F}_{\mathcal{W}}$: The complexity of Search is governed by:
  - The computation of k file accumulators AF. Without the knowledge of trapdoor α, and using FFT interpolation as specified in [3], this operation performs $kn \log n$ multiplications in $\mathbb{F}_p$ and k exponentiations in $\mathbb{G}$;
  - The generation of the authentication paths in tree TF for these accumulators, which amounts to $k \log N$ hashes;
  - The generation of the proof of intersection that takes $\mathcal{O}((kn)\log^2(kn)\log\log(kn))$ multiplications[5] in $\mathbb{F}_p$ to compute the gcd of the characteristic polynomials of the sets involved in the query result.
- out $= \emptyset$: The computational costs of this phase consist in:
  - The generation of the proof of membership for the missing keyword by calling twice GenerateWitness. This operation requires $2(d + d\log d)$ multiplications in $\mathbb{F}_p$ and $2d$ exponentiations in $\mathbb{G}$;
  - The computation of 2 bucket accumulators AW, which amounts to $2d \log d$ multiplications in $\mathbb{F}_p$ and $2d$ exponentiations in $\mathbb{G}$;
  - The generation of 2 authentication paths for these 2 buckets by running GenerateMTProof on tree TW, which performs $2\log m$ hashes.

[4]A more detailed table can be found in the full version of our paper here: http://www.eurecom.fr/publication/4540.

[5]More details on this complexity computation can be found in [3] and [10].

TABLE I: Computational complexity of our protocol, in the worst case where all N keywords are in all n files or where the not found keyword is the last in the query.

D: parameter of our system, $n \leq D$: number of files, N: number of keywords
m: the number of buckets in the index, d: size of a bucket
k: number of keywords in a query.
$\mathbf{E}_{\boldsymbol{G}}$: time to exponentiate elements in $\mathbb{G}$; $\mathbf{M}_{\boldsymbol{p}}$: time to multiply elements in $\mathbb{F}_p$;
$\mathbf{H}_*$: time to hash elements in $\{0,1\}^*$; **CI**: time to run CuckooInsert;
LC: light computation; **PI** : time to run ProveIntersection;
$\mathbf{BP}_{\boldsymbol{G}}$: time to compute bilinear pairings in $\mathbb{G}$; $\mathbf{M}_{\boldsymbol{T}}$: time to to multiply in $\mathbb{G}_T$

Algorithms	Approximate computational complexity
Setup	$(D+m+N)\ \mathbf{E}_{\boldsymbol{G}} + (md+Nn)\ \mathbf{M}_{\boldsymbol{p}} + 2(m+N)\ \mathbf{H}_* + N\ \mathbf{CI}$
QueryGen	k **LC**
Search	
out $= \mathcal{F}_{\mathcal{W}}$	$kn\ \mathbf{E}_{\boldsymbol{G}} + (kn\log n)\ \mathbf{M}_{\boldsymbol{p}} + (k\log N)\ \mathbf{H}_* + 1\ \mathbf{PI}$
out $= \emptyset$	$4d\ \mathbf{E}_{\boldsymbol{G}} + (2d + 4d\log d)\ \mathbf{M}_{\boldsymbol{p}} + (2\log m)\ \mathbf{H}_*$
Verify	
out $= \mathcal{F}_{\mathcal{W}}$	$3k\ \mathbf{BP}_{\boldsymbol{G}} + k\ \mathbf{M}_{\boldsymbol{T}} + (k\log N)\ \mathbf{H}_*$
out $= \emptyset$	$6\ \mathbf{BP}_{\boldsymbol{G}} + (2\log m)\ \mathbf{H}_*$

4. Verify: We also analyze the complexity of this algorithm according to whether all the keywords have been found:

- out $= \mathcal{F}_{\mathcal{W}}$: Verify runs k instances of VerifyMTProof on tree TF, which requires $k \log N$ hashes. Then, it executes VerifyIntersection which computes $3k$ pairings and k multiplications in $\mathbb{G}_T$.
- out $= \emptyset$: Verify runs twice VerifyMTProof on tree TW that computes $2\log m$ hashes and it invokes twice VerifyMembership that evaluates 2×3 pairings.

In summary, to verify the search results, a verifier $\mathcal{V}$ performs very light computations compared to the computations undertaken by the server when answering keyword search queries and generating the corresponding proofs. Besides, the verification cost depends on k only in the case where all the keywords have been found and is independent otherwise. Furthermore, we believe that for large values of k, the probability that the search returns a set of files containing all the k keywords is low. Hence, the verification cost will be constant and small (6 pairings and $2\log m$ hashes). On the other hand, for smaller values of k, the verification cost remains efficient.

Impact of D on the performance. This performance analysis assumes $n \leq D$, where n is the number of files. The value of D solely depends on security parameter 1^κ, and as such, defines an upper-bound to the size of sets for which we can compute a polynomial-based accumulator. It follows that in our protocol, the number of files that a data owner can outsource at once is bounded by D. However, it is still possible to accommodate files' sets that exceed the bound D. The idea is to divide the set of size n into $n' = \lceil \frac{n}{D} \rceil$ smaller sets of size D. By using the same public parameters, Setup accordingly creates for each set of D files an index and the corresponding Merkle trees. This increases the complexity of the Setup by a factor of n'. Namely, the data owner is required to build n' Cuckoo indexes and $2n'$ Merkle trees. whereas the server has to run n' ProveIntersection.

VIII. Related Work

Verifiable polynomial evaluation and keyword search. In [5], [11], the authors tackle the problem of verifiable delegation of polynomial evaluation. Their solutions allow a verifier to check whether a server evaluates the polynomial on the requested input correctly. As proposed in [11] and briefly

mentioned in [5], such a solution is suitable to the problem of verifiable keyword search where the file is encoded by its characteristic polynomial. Nevertheless, the application of [5] and [11] to verifiable keyword search would not be efficient. Besides to accommodate public delegatability and conjunctive queries, as achieved by our scheme, the proposals [5], [11] may require elaborate adjustments.

Verifiable keyword search on encrypted data. Recent work [12]–[15] adopt a different scenario from the one we follow here: While our setting focuses on verifiable keyword search on outsourced (sanitized) data and cares about public delegatability and public verifiability, the solutions proposed in [12]–[15] support verifiable keyword search on encrypted data and satisfy the data and query privacy properties. In particular, the work of Chai and Gong [12], extended in [13], exploits a searchable symmetric encryption scheme to develop a verifiable keyword search solution that preserves data confidentiality while enabling the verification of search results returned by the cloud. However, due to the use of a symmetric searchable encryption, these proposals do not offer public delegatability nor public verifiability. Besides, their adversary model considers a semi-honest-but-curious cloud whereas in this paper we consider malicious clouds. Cheng et al. [15] propose a protocol for verifiable conjunctive keyword search that leverages a combination of a searchable symmetric encryption scheme with an *indistinguishability obfuscation circuit* (*iO* circuit) realizing the search operation. While public verifiability is achieved via another public *iO* circuit representing the verification function, public delegatability is not addressed in this work. Nevertheless, it is worth considering generating an *iO* circuit to realize the public delegatability. Still, the generation and obfuscation of such circuits induce substantial costs that the authors in [15] barely mention. Furthermore, Zheng et al. [14] propose a solution called Verifiable Attribute-Based Keyword Search (VABKS) which allows a data owner to grant a user satisfying an access control policy the right to query a keyword over the owner's outsourced encrypted files and to verify the search result returned by the server. This solution does not support conjunctive keyword search. Besides, public delegatability and public verifiability are not in the scope of this work: only a fine-grained access control enables authorized users to issue search queries and verify search results. In summary, this review of existing work for verifiable keyword search on encrypted data [12]–[15] identifies the gap that should be addressed as a future work between verifiable private search and publicly delegatable and verifiable search. While our scheme does not support search on encrypted data (as this problem is orthogonal to our scenario), it offers public delegatability and verifiability, which most of the existing work on verifiable keyword search on encrypted data do not achieve. We can customize our protocol to allow search on encrypted data at the price of sacrificing public delegatability and verifiability. Nevertheless, methods such as attribute-based encryption can be used to delegate search capabilities to a third-party user.

IX. Conclusion

In this paper, we presented a protocol that enables a data owner to outsource its database to a cloud server, in such a way that any third-party user can perform search on the outsourced database and verify the correctness of the server's responses. The proposed solution is efficient: The storage overhead at the data owner and third-party users is kept to a minimum, whereas the verification complexity is logarithmic in the size of the database. Moreover, it is provably sound under well-understood assumptions, namely, the security of Merkle trees and the strong bilinear Diffie-Hellman assumption. As a future work, we will implement our protocol to demonstrate its feasibility with real data. We will also consider the problem of updates in the set of files and keywords.

X. Acknowledgments

This work was partially funded by the H2020 project CLARUS (grant No. 644024).

References

[1] B. Parno, M. Raykova, and V. Vaikuntanathan, "How to Delegate and Verify in Public: Verifiable Computation from Attribute-Based Encryption," in *Proceedings of the 9th Theory of Cryptography Conference, TCC 12*, 2012, pp. 422–439.

[2] I. Damgård and N. Triandopoulos, "Supporting Non-Membership Proofs with Bilinear-Map Accumulators," *IACR Cryptology ePrint Archive*, vol. 2008, p. 538, 2008.

[3] R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos, "Verifiable Set Operations over Outsourced Databases," in *Public-Key Cryptography–PKC 2014*. Springer, 2014, pp. 113–130.

[4] R. Gennaro, C. Gentry, and B. Parno, "Non-Interactive Verifiable Computation: Outsourcing Computation To Untrusted Workers," in *Advances in Cryptology–CRYPTO 2010*. Springer, 2010, pp. 465–482.

[5] D. Fiore and R. Gennaro, "Publicly Verifiable Delegation of Large Polynomials and Matrix Computations, with Applications," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 501–512.

[6] L. Nguyen, "Accumulators From Bilinear Pairings and Applications," in *Topics in Cryptology–CT-RSA 2005*. Springer, 2005, pp. 275–292.

[7] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology–CRYPTO'87*. Springer, 1988, pp. 369–378.

[8] M. Dietzfelbinger and C. Weidling, "Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins," *Theoretical Computer Science*, vol. 380, no. 1, pp. 47–68, 2007.

[9] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[10] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Optimal Verification of Operations on Dynamic Sets," in *Advances in Cryptology–CRYPTO 2011*. Springer, 2011, pp. 91–110.

[11] S. Benabbas, R. Gennaro, and Y. Vahlis, "Verifiable Delegation of Computation over Large Datasets," in *Advances in Cryptology – CRYPTO 2011*. Springer, 2011, pp. 111–131.

[12] Q. Chai and G. Gong, "Verifiable Symmetric Searchable Encryption for semi-Honest-but-Curious Cloud Servers," in *IEEE International Conference on Communications (ICC), 2012*. IEEE, 2012, pp. 917–922.

[13] Z. A. Kissel and J. Wang, "Verifiable Phrase Search over Encrypted Data Secure against a Semi-Honest-but-Curious Adversary," in *IEEE 33rd International Conference on Distributed Computing Systems Workshops (ICDCSW), 2013*. IEEE, 2013, pp. 126–131.

[14] Q. Zheng, S. Xu, and G. Ateniese, "VABKS: Verifiable Attribute-Based Keyword Search over Outsourced Encrypted Data," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 522–530.

[15] R. Cheng, J. Yan, C. Guan, F. Zhang, and K. Ren, "Verifiable Searchable Symmetric Encryption from Indistinguishability Obfuscation," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '15. New York, NY, USA: ACM, 2015, pp. 621–626.