Conjunctive Keyword Search on Encrypted Data with Completeness and Computational Privacy

Radu Sion[*] Bogdan Carbunar[†]

Abstract

We introduce mechanisms for secure keyword searches on a document server. We propose protocols with computational privacy, query correctness assurances and minimal or no leaks: the server either correctly executes client queries or (if it behaves maliciously) is immediately detected. The client is then provided with strong assurances proving the authenticity and *completeness* of server replies. This is different from existing research efforts, where a cooperating, non-malicious server behavior is assumed.

We also strengthen the privacy guarantees. The oblivious search protocol not only hides (from the server) the outsourced data but also does not leak client access patterns, the queries themselves, the association between previously searched keywords and returned documents or between newly added documents and their corresponding keywords (not even in encrypted form). This comes naturally at the expense of additional computation costs which we analyze in the context of today's off the shelf hardware. In a reasonable scenario, a single CPU off-the-shelf PC can easily handle hundreds of such oblivious searches per minute.

1 Introduction

In data outsourcing frameworks, data management needs of clients are off-loaded to specialized service providers. This is intuitively advantageous for parties with less experience, resources or trained man-power such as small companies and individuals. Moreover, the resulting expertise consolidation at the service provider is likely to result in increased availability, better quality and cheaper service. Examples include a wide spectrum of applications, from traditional services such as document and email hosting, where individual user mailboxes are stored on a dedicated remote mail server, to novel paradigms such as *database services*, in which dedicated database servers are providing query execution and an associated query interface to the outsourced data to multiple clients. Here we focus on a conjunctive keyword search application scenario in which outsourced documents are to be retrieved based on keyword searches to a file or email server.

In such scenarios it is often important to protect *both the outsourced information as well as the associated client access patterns*. Confidentiality can be achieved by storing the data encrypted. Maintaining the benefits of data outsourcing however, requires techniques for querying such encrypted data directly on the server with minimal additional overhead and load on the client side.

While the general problem of performing arbitrary queries over encrypted data seems to be quite challenging, several research results provide solutions for queries of specific levels of expressivity. In particular the core problem of performing simple [4, 5, 6, 11, 23] and conjunctive keyword searches [15] has been addressed. Existing research however, assume the existence of a cooperating, non-malicious—albeit illicitly curious—server. For example in the case of a keyword search on a document server, the server is assumed

[*]Department of Computer Sciences, Stony Brook University, Stony Brook, NY 11794, sion@cs.stonybrook.edu

[†]Department of Computer Sciences, Purdue University, West Lafayette, IN 47906, carbunar@cs.purdue.edu

to fully comply to the query protocol and reply with all the documents matching a particular (set of) keyword(s). Denial of service or malicious behavior on the side of the (possibly compromised) server has not been considered so far.

In this paper we propose to address the issues of *assurance* and privacy in outsourced data frameworks. We introduce a set of protocols which provide the client with strong cryptographic query assurances, proving the authenticity and completeness of server replies. Additionally, we strengthen the privacy guarantees as follows: we not only hide from the server the outsourced data but also do not leak client access patterns, the queries themselves (not even in encrypted form), the association between previously searched keywords and returned documents or between newly added documents and their corresponding keywords. This is especially challenging due to the *dynamic* nature of the interaction between the client and the server. The proposed solutions handle such dynamic behavior and enable the client to naturally add and remove documents on the server.

To do so, we rely on maintaining just enough state information at the client side, which serves as *checksum* to authenticate server messages. The client is assumed to be able to store a number of checksums that increases linearly with the number of search keywords. The proposed solutions are independent of the deployed encryption methods.

The paper is structured as follows. Section 2 introduces the operating model and discusses information leaks. Related work is explored in Section 3. Initial mechanisms for query assurance are introduced in Section 4. Section 5 proposes methods for querying data with computational privacy guarantees. Section 6 discusses alternative related protocols and Section 7 concludes.

2 Model

Informally, a server offers document (e.g., files, emails, relational data) hosting for storage or processing power constrained clients. The clients need to perform queries[1] on the stored documents while revealing a minimal amount of information to the server. Additionally, the client will require to be able to add new documents and remove or update arbitrary previously-stored ones.

The adversarial setting considered in existing research assumed a server that is curious but not malicious. In this work we propose to go one step further and consider malicious server behavior. The server is now not only curious, but also malicious (e.g., by actively withholding information from the client), given the chance to get away undetected.

Thus, not only do clients need to preserve the confidentiality of the documents themselves and the clear-text content of the keyword search queries but now also require assurances of *correctness* and *completeness* with respect to all the outsourced documents. Query correctness assures that only documents are returned that match the query; query completeness assures that all such documents are returned by the server. While correctness is trivial to check for a client, assuring completeness is challenging.

Notation. In the following let us consider the interaction perspective of a single client. Let each document be identified uniquely by a document identifier, which is assumed to be uncorrelated to the contents of the file. We assume in the following that documents are identified by random (uniformly distributed) numbers. Each document identifier is placed in a "header" of the corresponding document, before encryption. For simplicity, we will use "d_i" to denote these identifiers and $\mathbb{D} = (d_i)_{i=1..n}$ to denote the outsourced documents so far. Let $\mathbb{K} = (k_i)_{i>0}$ be the set of keywords seen so far and k be their number. Let χ be the average number of keywords per document.

Both the stored documents in $\mathbb{D}$ as well as the query keywords $q = \{k_{i_1}, k_{i_2}, \ldots, k_{i_q}\}$ are encrypted with a symmetric key under the client's control. For simplicity we will not use an explicit notation for this encryption but rather assume it

[1] In this paper we are discussing mainly conjunctive keyword search queries.

implicitly. The introduced mechanisms are independent of the used encryption method.

Unless specified otherwise let us assume that the client is able to store $O(k)$ data items. Let the actual client storage size be denoted by m.

Information Leaks. Given a method for searching for keywords on encrypted data, let us informally define certain classes of information leaks. As we are mostly concerned with the leaks that the query protocol itself introduces, we assume perfect (non-malleable) encryption of both the keywords and the outsourced documents.

Intuitively, the notion of a leak refers to the ability of the server to infer information about the plain-text of or the access patterns to the outsourced encrypted information. Naturally, in the absence of queries from the client, there are (by our assumption on the encryption system) no leaks. We are thus mostly concerned with the information leaked after a client submits a query and it is executed by the server.

Let us first define the concept of *keyword search tokens*. In order to perform a (conjunctive) keyword search on the server, the client will need to submit a query. This query will naturally be composed of a set of information items that relate uniquely to the searched keywords. In the simplest, un-encrypted case, these information items could be the keywords themselves for traditional databases. If the data is stored encrypted on the server, these items are secure constructs derived from the keywords (e.g. encrypted query keywords). We call any such information items for a given query *keyword search tokens*.

We say that a *type 1 leak* occurs, if, after receiving and executing a query $q = \{k_1, k_2, \ldots\}$, the server can systematically construct any association between the already seen keyword-tokens (including the ones for the current query) and the so far returned encrypted documents. For example, if, for a given query, the client simply provides the associated encrypted keywords to the server. Upon returning the query results, the server can easily infer that the returned documents are associated with the submitted encrypted query keywords.

An arguably more undesirable leak is a *type 2* leak. A type 2 leak is characterized by the server being able to construct a mapping between *each and every* considered keyword search tokens and all stored encrypted documents.

Type 2 leaks can be induced by data structures required by the query mechanism. For example, a type 2 leak occurs if the query protocol requires a data structure on the server which provides a mapping between all documents and encrypted keywords. Additionally, once all the considered keyword search tokens have been used individually (e.g., if all the keywords have been searched for), a type 1 leak converges naturally to a type 2 leak.

For completeness let us also define *type 3 leaks*. This type of leak occurs in the process of adding new keywords (not considered in existing research) if as a result the server knows that previously stored documents do not contain the added new keywords, or at least that the client is not aware of them.

The significance of each such leaks is application specific. Providing query assurances while also handling leaks becomes increasingly expensive. Thus, for example, if the deployment scenario allows for type 1 leaks, (e.g., a low-security email-server setup) then a lower-overhead assurance mechanism is desirable.

3 Related Work

Simple Keyword Searches. Song et al.[23] propose a scheme for search on encrypted data in a scenario where a mobile, bandwidth restricted user, wishes to store data (e-mail) on an untrusted server. The scheme requires the user to split the data into fix-sized words, encrypt each word separately using a symmetric key protocol and xor the result with a structure containing a pseudo-random bit string and a mapping of the string under a secret key, using a pseudo-random function. The secret key is made dependent on the encrypted word. The resulting data is stored on the server. The structure enables the detection of keyword matches, without revealing the server the keyword or the contents of the stored data. The drawbacks of the scheme are fix-sized words, the complexity of the encryption and search ($O(n)$ where n is the number of words) and the impossibility of verifying the completeness of the results returned by the server. The paper also discusses the use of an encrypted index, allowing the whole data to be encrypted as a block.

Additionally, this scheme features *type 1* leaks. Indeed, correlations between searched keywords and matched documents are naturally exposed by the scheme. Moreover, the server can further perform any combinations of conjunctive keyword searches with the keywords searched by the client. Also, when adding new documents, the server can search them with the tokens revealed by previous keyword searches of the client.

Eu-Jin Goh [11] proposes to associate indexes with the documents stored by the server. More precisely, the index of a document is a Bloom filter [3] containing a codeword for each each unique word in the document. The codeword of a word is derived by twice applying a pseudo-random function to the word. The size of document indexes, as documented in the paper, is proportional to the document size. Chang and Mitzenmacher [6] propose a similar approach, where the index associated with documents consists of a string of bits of length equal to the total number of words used (dictionary size). Two solutions are given, one where the dictionary of words can be stored at the client and one where it has to be stored encrypted at the server.

All of the above methods feature *type 1* leaks. As discussed above (Section 2), at the extreme, these leaks can degenerate into *type 2* leaks.

An interesting version of searching on encrypted data is proposed by Boneh et al.[4], where e-mails encrypted by senders with the public key of the intended receiver are stored on untrusted mail servers. The paper presents two protocols allowing receivers to securely search on the encrypted data. The first protocol, a non-interactive searchable encryption scheme, is based on a variant of the Diffie-Hellman problem and uses bilinear maps on elliptic curves. The second protocol, using only trapdoor permutations, needs a large number of public/private key pairs.

Conjunctive Keyword Searches. Golle et al.[15] extend the above problem to conjunctive keyword searches on encrypted data. They propose two solutions. The first solution requires the server to store capabilities for conjunctive queries, whose size is linear in the total number of documents. The paper claims that a majority of the capabilities can be transferred offline to the server, but this only assumes that the client knows beforehand its future conjunctive queries. The second solution requires much less communication between the client and the server, proportional with the number of keywords in the conjunctive search, but doubles the size of the data stored by the server. A limitation of both schemes is the requirement of specifying the exact positions where the matches have to occur.

None of the above protocols handle the issue of query completeness; there is no guarantee of the correct receipt of *all* query-matching documents by the client.

XML Documents. For XML documents, Brinkman et al.[5] build a special polynomial for each document, facilitating the retrieval of keyword matches. The polynomials are subsequently split between the server and client. Since the client's polynomial is randomly generated, the client can store only a random seed, used to generate its version of the polynomial. The polynomial expression stored by the server for each document is of size proportional to the number of words in the document.

Database Queries. Hacigumus et al.[16] propose a method for executing SQL queries over encrypted data outsourced to a server. The encrypted data is partitioned into secret partitions such that queries referencing original data items identifiers are rewritten in terms of such partition identifiers (which are kept publicly associated with the encrypted data partitions). The workload (e.g.

relational JOINS) of such a rewritten query over the encrypted partitioned data can now be performed partly on the server but the query results will contain a super-set of the actual desired results. The authors then propose to perform the remaining work (pruning of non-matching tuples) in a post-processing phase on the client side.

The information leaked to the server is claimed to be minimal. This is arguable, as it basically depends directly on the partitioning scheme. At the one extreme, if there are many partitions and each partition contains exactly one tuple in the original relation, the information leak becomes 100% unless the client stores enough information (order of the relation size) to randomize partitioning. But requiring the client to hold the same size of information as the server defeats the whole purpose of outsourcing data in the first place. At the other extreme, if there are only a very small number of partitions the outsourcing benefit is defeated again, as most of the data will be returned to the client for post-processing. Nevertheless, the proposed method elegantly illustrates the natural trade-off between information leaks and storage and computation overheads.

Private Information Retrieval. Private information retrieval (PIR) mechanisms were first proposed in [9] as a solution to accessing outsourced data, while preventing the data servers to learn anything about the client access patterns. It is important to note that the main purpose of PIR protocols is to hide access patterns but not the actual data content.

In initial results, Chor et al.[7] proved that, in an *information theoretic* setting (in which queries do not reveal any information at all about the accessed data items) any solution requires $\Omega(n)$ bits of communication. To avoid this overhead, Chor et al.[7] show that if multiple non-communicating databases hold replicated copies of the data, PIR schemes requiring sub-linear communication exist.

A more practical setting, namely *computational PIR* (cPIR) was explored in subsequent efforts. In [8, 21] the initial assumptions were relaxed and the server is assumed to be able to perform only polynomial time computations. The query access patterns are then only computationally hidden from the server. An instance of cPIR proposed by Kushilevitz and Ostrovsky [17], uses the hardness assumption of deciding quadratic residuosity [14] in order to provide an elegant result, namely a single-server PIR protocol with sub-linear communication.

Oblivious RAMs. In related seminal work [20], Ostrovsky and further in [13] Goldreich and Ostrovsky address the problem of hiding memory access patterns of software, specifically, the locations accessed, the access order and access count. In [20], a method for doing on-line simulations of arbitrary RAM accessing programs is proposed. The cost paid is just a slowdown poly-logarithmic in the initial program running time.

4 Query Assurances

In this and the following section we discuss a series of mechanisms that gradually increase privacy and completeness assurances. The final goal is a solution that not only provides query assurances but also oblivious keyword search, leaking nothing to the curious (and possibly malicious) server. In particular, the server (i) will not be able to behave maliciously (e.g., by incorrectly replying to queries) without being immediately detected and (ii) will not be able to determine what keywords were searched for (not even in encrypted form) or what documents correspond to which keywords.

In this section we handle mostly (i) and set the stage for oblivious keyword search mechanisms, discussed subsequently. In particular we propose a keyword search protocol that can cope with malicious behavior: the server either correctly executes client queries or (if it behaves maliciously) is immediately detected with high probability. The client is then provided with strong assurances proving the authenticity and *completeness* of server replies.

4.1 Dynamic Data

Let us first consider a simple query mechanism allowing a client to verify the correctness of server query responses.

Single Keyword Searches. In the case of single (non-conjunctive) keyword searches, one natural way for the client is to store a counter for each keyword, containing the number of outsourced matching documents. The client then only needs to compare the number of documents returned to a single keyword query with the locally stored counter. Authenticity is guaranteed by the fact that the documents are stored encrypted. Query completeness is guaranteed if the number of returned documents matches the expected value, as the server cannot—by the assumptions on the encryption scheme—create new encrypted documents that "look" authentic.

While this scheme is easy to implement, it has the drawback of not recording the identity information of matching documents. This becomes important when *document removals* are possible. In this case, the server can record document removal requests initiated by the client, but remove different documents. It can then return different sets of documents for future keyword queries. In effect this compromises the client's ability to authenticate the removal process. We call this a *replacement attack*. For example, if documents named d_i, d_j and d_l contain keyword k_1 and the client requests the removal of document d_j, upon a later request for the documents containing k_1, the server can return d_i and d_j. An arguably unscalable fix to this problem would be for the client to record all document deletion requests. Before exploring how this issue could be solved let us first consider the more interesting case of multiple keywords.

Conjunctive Keyword Searches. Counters alone cannot be applied for multiple keyword searches. A (not scalable) fix would be to store one count for all subsets of all keywords $\mathbb{K}$.

Another idea would be for the client to store, for each keyword, a simple set structure of identifiers of documents containing it. Let us call these *keyword document sets* (KDS). These will then be used to double-check server responses. For example the KDS sets corresponding to keywords k_1 and k_2 in Figure 1, are $KDS_1 = \{d_1, d_3\}$ and $KDS_2 = \{d_2, d_3\}$. If the client needs to query the server for all documents containing keywords k_1 *and* k_2, it will first look at KDS_1 and KDS_2, perform their intersection, and request from the server the corresponding documents (in this case $\{d_3\}$). In this solution, the client will need to store $O(nk)$ values.

4.1.1 Server-Side KD Sets.

To solve this inconvenience, the natural follow-up would be to store the KDS sets on the server. For each query, the client will requests them from the server. This immediately raises the problem of authenticating the KDS sets themselves.

Intuitively, to handle the above issue of authenticating the KDS sets retrieved from the server, the client will have to store some sort of checksum information. Then, when retrieving a KDS it can verify its authenticity by computing its checksum and comparing it with the stored (known) value.

Before discussing what such an actual checksum might look like, let us first briefly explore the server-side. Naturally, any outsourced data structure (including the KDS, as well as the documents themselves) will be kept encrypted on the server, allowing only the client direct access to its contents. By the assumption on the encryption system, the server is not able to gain information on the outsourced documents by looking at the encrypted versions; in addition, the server cannot create new valid encrypted documents.

Instead of using a single encryption key for all document identifiers in all such document sets we propose to use a different key per KDS for encrypting the items it contains (the reason for this choice will become clear in Section 6). Let *key* be a client-side secret and KDS_i by definition the set of all documents containing keyword k_i.

Note: In the following, let $H()$ denote a one-way cryptographic hash, such as SHA-512 [19]. Also, let "$||$" denote concatenation.

Then, let $key_i = H(key||k_i)$ be the key used to encrypt the document identifiers that are contained in KDS_i. These keys are easily generated by the client when a new document is to be outsourced on the server. Additionally, this is done without any additional storage requirements on the client (which needs to only remember a secret value *key*).

As we will see, in a static data case (only), the advantages of such a scheme are that a server will not be able to cross-correlate document identifiers between KDS sets (the same document will be represented differently in different KDS sets). This will thus significantly reduces the types of leaks incurred.

4.1.2 Onion Hashing.

Getting back to the issue of authenticating the KDS sets retrieved from the server, let us first note that the server will not be able to add new (valid-looking) entries to any KDS (as a non-malleable cipher is used). The server could however maliciously *remove* entries from the sets before returning them.

To alleviate this problem, we proposed above to compute (and store at the client) a checksum for each KDS that can verify its content. One natural choice would be a cryptographic hash or message digest of its content. One problem with such an approach is that for every new incoming document, simply updating these checksums will require the client to retrieve each and every one of the associated sets from the server first. This might not be acceptable. For new documents, the client would like the checksum values it keeps to be easy to update, preferably without significant communication with the server.

We propose the following mechanism. For illustration purposes, let us consider three documents all containing (at least) keyword k_1 coming in (or being generated) in the following chronological order: d_1, d_2, d_3 and d_4. The KDS for keyword k_1 on the server will now contain these new document identifiers, $KDS_1 = \{d_1, d_2, d_3, d_4\}$.

A structure that would allow the client to authenticate this set while also permitting further updates (without communicating with the server) works as follows. The client maintains one hash-value per set, initialized with zero (when no documents containing the corresponding keyword have been seen so far).

Next, for each new document received, the client will concatenate the previous hash value with the identifier of the document and compute a one-way cryptographic hash function of it. In the above case for example, for keyword k_1, the client would store the value $H_{check} = H(d_4||H(d_3||H(d_2||H(d_1||0))))$. This would effectively construct a one-way "onion hash" combining all the document identifiers in the set. This construct allows for a simple authentication process for the set of documents per keyword provided by the server at a later time: re-constructing the hash from the set received from the server and comparing it with the stored version. The scheme also has the advantages of being easy to build and of requiring a constant amount of storage space. Document additions are easy to perform, since the client only has to add one hash layer to the values stored for all the keywords contained in the new document. The problem of this scheme however consists in the difficulty of removing arbitrary documents. This is the case for example if d_2 is to be removed from the server. Next time the server replies with the set missing d_2, the checksum cannot be reconstructed.

This constitutes a problem. While the scheme can be augmented to allow the removal of *all* documents "older than a certain timestamp" (details are out of scope here), it will still not provide the ability to support removal of documents that were stored chronologically in the "middle". This is often undesirable. We now propose a novel checksum mechanism that solves this problem.

4.1.3 Multiplicative Checksums

Let the client choose two secrets: a large prime p and a random integer $1 < x < p$. Instead of a hash, for a given keyword, the client stores a product of terms, each term corresponding to one of the documents in the associated document (assumed numeric) identifier set. To authenticate a KDS set $\{d_1, \ldots, d_t\}$ the client stores the number

$$\prod_{i=1}^{t}(d_i + x) \bmod p;$$

for an example see Figure 1. We call this a *multiplicative checksum.* Such a checksum allows for both easy removal (multiplication by $(d_i + x)^{-1}$) and addition (multiplication with $(d_i + x)$) of arbitrary documents. In the following, unless specified we are going to name this construct "client checksum(s)" or, even simpler "checksums" if unambiguous.

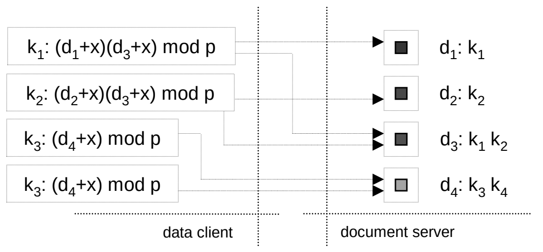

Figure 1: Checksums can be kept on the client side to authenticate server replies.

Suppose that the terms $(d_i + x)$ are uniformly distributed [2] over $\mathbb{Z}_p^*$. By appropriate coding of the document identifiers, this is easily achievable (e.g. through the full domain hash scheme of [2]). Then, the checksum for an arbitrary KDS set is also uniformly distributed over $\mathbb{Z}_p^*$.

But how about a malicious server? Let us remember the replacement attack discussed above. Apparently, multiplicative checksums are vulnerable to such an attack. When instructed to remove a certain document (d_i) the server will attempt to instead remove (a set of) other documents $\{d_{j_1}, .., d_{j_t}\}$ while preserving the value of the multiplication checksum. Upon closer inspection however, it becomes clear that this is not possible. A malicious server running such an attack only has an extremely low chance (of $1/p$) to succeed, as the checksum is stored under the client's control, and the server cannot produce new encrypted documents. This shows that client checksums can both serve as efficient and easy means to authenticate KDS sets.

Leaks. Since the server keeps for each keyword a set of identifiers of documents containing the keyword, even though the keywords and document identifiers were encrypted by the client, this solution exhibits a type 2 leak. Each time a document is added on the server, to keep the search structure consistent, the client needs to update the KDS sets corresponding to keywords contained in the document.

Overheads. The server is required to store the KDS sets, thus incurring a $O(n\chi)$ storage overhead. For each searched keyword, the client will have to retrieve the corresponding KDS set from the server. The average size of a KDS is $\frac{n \times \chi}{k}$, incuring $O(\frac{n \times \chi}{k})$ communication costs (a single message of this size). Additionally, the client will need to compute its checksum ($O(\frac{n \times \chi}{k}$ costs) multiplications).

Using a typical example of an email server storing (for a particular user) $n = 10000$ emails, $k = 1000$ keywords and $\chi = 50$ keywords per document, we have the following overheads: a message of roughly 2 KBytes and 500 multiplications at the client.

4.2 Static Data

The dynamic data solution presented so far features type 2 leaks and uses $O(k)$ client-side storage. In a static data scenario, no new documents are added by the client after outsourcing to the server. We would also like to extend the static scenario by allowing the client to remove some of the documents added to the server (we call this a "semi-static" case). This scenario is further built upon and explored in Section 6.1 where we discuss severely storage- constrained mobile clients.

In a static scenario, the client will compute/encrypt off-line all necessary data structures required by the server to perform searches. It will

[2] Although it can happen by construction that $(d_i + x) \equiv 0 \bmod p$, we ignore this event for the analysis, as it only occurs with negligible probability.

then outsource these structures to the server. In a mobile client scenario for example, this can happen e.g., on the "home PC", allowing for later read-only retrieval of documents by the mobile client, e.g., a laptop.

Semi-static data. In a semi-static scenario, if the client is allowed to dynamically remove documents, the above proposed storage mechanism used in conjunction with client-side storage of checksums ($O(k)$), features only type-1 leaks. This is so because document identifiers are encrypted with different keys in different KDS sets; the server is prevented from building associations between keywords and documents. Such associations are leaked only during actual keyword searches (type 1 leaks).

No Document Removals. Additionally, if the data model is completely static, (document removals are not allowed) it turns out that client storage requirements can be reduced. This is mainly so because now, instead of individually encrypting each document identifier in each server-stored KDS set, these sets can be encrypted as a whole with one single key (as no updates are expected). For example, KDS_i can be encrypted with the key $key_i = H(key||k_i)$ (see Section 4.1.2).

Then, if complete KDS encryption is deployed, the encryption mechanism also naturally provide for KDS authentication. There is no need any more for client-side checksums. The ability to properly decrypt the retrieved lists will be a guarantee of no tampering. Thus, while requiring only $O(1)$ client storage, this solution again features only type 1 leaks.

5 Towards Obliviousness

While the mechanisms proposed above have the advantage of being relatively cheap and not leaking more information than existing efforts (type 1 leaks), we propose to get closer to complete (computational) privacy. The following mechanisms prevent a curious server to determine (i) what keywords were previously searched for (not even in encrypted form), and (ii) what document identifiers correspond to which (possibly encrypted) keywords.

5.1 Less Information Leaks

The leak source of the above explored methods is the client signaling the server the keywords contained in newly added document. This happens in the process of document addition when the KDS sets will need to be updated on the server. If we would modify document addition such that for example the client updates *all* KDS sets stored at the server (instead of only the ones corresponding to keywords contained in the new document), the server will learn nothing of the new document.

To do so, let us represent all server-stored KDS sets as a $k \times n$ bit matrix C, where each row corresponds to a keyword and each column to a document. The bit at row i and column j, C_{ij}, is then set to 1 if and only if the document d_j contains the keyword k_i.

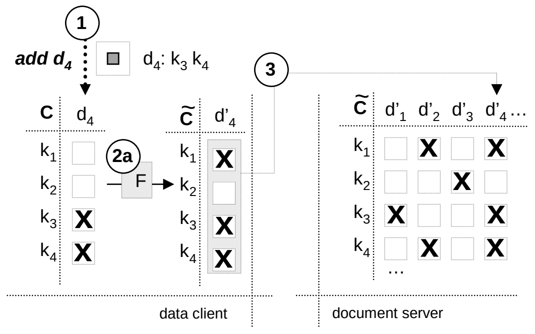

Figure 2: Adding a new document involves computing and then obfuscating the corresponding matrix column entry.

Storing C in clear at the server naturally leads to a type 2 leak. To avoid this, instead of revealing C to the server, the client provides an obfuscated version of the matrix, called $\widetilde{C}$, to the server. $\widetilde{C}$ is computed out of C in a pseudo-random but yet reversible manner. Let F be a bitwise pseudo-random function [12]. We compute the entries of the matrix $\widetilde{C}$ by

$$\widetilde{C}_{ij} = \mathit{last\_bit}(F(k_i, R_j, C_{ij})),$$

where R_j is a random number corresponding to document d_j. R_j can be easily computed (with only constant storage) by the client out of the document identifier using a pseudo-random number generator G and a constant random seed R.

The client can dynamically outsource a new document by constructing a matrix column having "1"s only in positions corresponding to keywords contained in the document and encoding this column using F as previously shown (see Figure 2). The resulting column, along with an encrypted version of the document are then sent to the server. If the client is storage-constrained, the computation of the initial $\widetilde{C}$ can be done in a similar, sequential manner. Removing a document is a much simpler operation, requiring only the elimination of the corresponding column of $\widetilde{C}$.

Keyword Searches. A conjunctive keyword search $\mathit{query} = \{k_1, k_2, \ldots, k_q\}$ is initiated by the client by revealing the corresponding row indexes, $p_1, p_2, \ldots, p_q$, to the server, which returns the actual rows of $\widetilde{C}$. To then recover the corresponding rows of C, the client initializes the pseudo-random generator G with the random seed R in order to sequentially compute the pseudo-random values $R_1, \ldots, R_n$ corresponding to all documents. Then, for all $1 \leq j \leq n$, the client sets $C_{p_ij} = 0$ if $\mathit{last\_bit}(F(k_i, R_j, 0)) = \widetilde{C}_{p_ij}$, otherwise $C_{p_ij} = 1$. Finally, the client uses the stored client checksums of keywords $k_1, \ldots, k_q$ in order to verify the correctness of the inverted $(r_1, \ldots, r_q)$ rows (and thus implicitly of the server response). If it checks, the client performs a bit-wise AND of the rows, $r = \wedge_{i=1}^{q} r_i$ and requests from the server the documents corresponding to positions in r that contain "1"s.

Query Completeness. The proposed solution provides query completeness, as the server is not able to alter the client checksum. Furthermore, the server can only cheat on the matrix $\widetilde{C}$, whose entries are transformed in a pseudo-random manner back into the matrix C by the client. Altering the matrix elements of $\widetilde{C}$ essentially results in a random selection of the KDS sets. However, due to the properties of the client checksum (see Section 4.1.3), the chance of the server of hitting a different KDS set that has the checksum expected by the client is negligible (bounded by $1/p$).

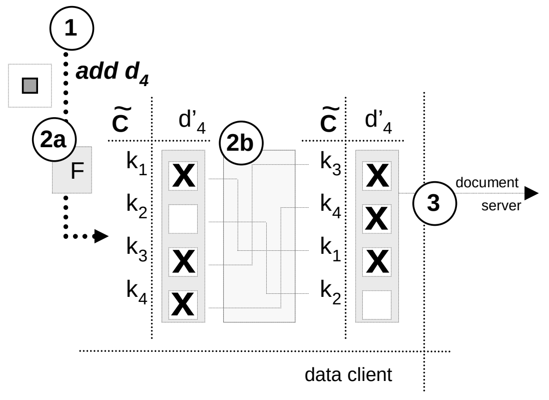

Figure 3: The ranks of keywords (rows in $\widetilde{C}$) are permuted to avoid leaks derived from the public nature of the keyword dictionary. Refer also to Figure 2.

Leaks. This solution exhibits a type 1 leak, since the client reveals relations between row indexes corresponding to the searched keywords and the encrypted documents containing them.

An additional leak is to be handled here. It occurs if the considered keywords become (or are) public, e.g., when an entire known dictionary is used. If keywords are kept alphabetically ordered in $\widetilde{C}$, the server can build a one-to-one correspondence between rows in the matrix and actual keywords. This then becomes a leak problem due to the fact that the client will reveal which keywords it is considering upon searching, as it first retrieves the corresponding rows from the matrix. While not explicitly pointing it out, we believe other authors also encountered this problem in a slightly different context [6].

We handle this (similarly to [6]) by randomly permuting the keywords (rows in the matrix, see Figure 3): the client generates a pseudo-random permutation [18] of the alphabetically ordered keywords. It then stores and keeps secret k records of the type (k_i, p_i, c_i), where k_i is the i-th alphabetically ranked keyword, $p_i \in [1, \ldots, k]$ its corresponding position in the permutation and c_i is k_i's checksum (see Section 4.1.3). Keyword k_i is then associated with the p_ith row of $\widetilde{C}$.

In the following we propose a method that prevents even type 1 leaks (by deploying a variation of computational PIR) at the expense of additional computation costs.

5.2 Oblivious Search

While type 1 leaks seem less revealing, in the worst case they still lead to type 2 leaks. Moreover, type 1 leaks allow the server to build statistics of keyword search popularity and eventually perform guesses. In this section we propose an *oblivious keyword search* method, allowing the client to hide any information about the queried keywords, albeit at increased computation cost for the client. The only information leaked to the server consists in the number of keywords contained in conjunctive queries, shared by sets of documents.

We use a modified version of the Computational PIR scheme of Kushilevitz and Ostrovsky [17] to achieve this goal. Initially, the client randomly chooses two prime numbers p and q of equal bit length, computes their product, $N = pq$ and sends N to the server. To search for documents containing keyword k_i, without revealing the row index p_i corresponding to k_i in matrix $\widetilde{C}$, the client generates k numbers, $s_1, s_2, \ldots, s_k$, such that the p_i-th number s_{p_i} corresponding to the row p_i of the matrix $\widetilde{C}$, is a quadratic non-residue (QNR) and the rest are quadratic residues (QR), all modulo N. The client sends $s_1, s_2, \ldots, s_k$ to the server.

For each column c in the bit matrix $\widetilde{C}$, the server computes exactly one value, v_c, as the product of k numbers, $v_c = \prod_{i=1}^{k} v_{ic}$, where $v_{ic} = 1$ iff. $\widetilde{C}_{ic} = 0$ and $v_{ic} = s_i$ otherwise. The server sends the computed values $v_1, v_2, \ldots, v_n$ to the client, that checks their quadratic residuosity. If value v_c is a quadratic residue, then $\widetilde{C}_{p_i c}$ is 0, otherwise it is 1. Since s_{p_i} is a quadratic non-residue, $v_{p_i c}$ is a quadratic residue if and only if $\widetilde{C}_{p_i c}$ is 0.

By the quadratic residuosity assumption, the server cannot check the quadratic residuosity of $s_1, s_2, .., s_n$ or of $v_1, v_2, .., v_n$ without being able to compute the primes p and q (known only to the client). The client however can easily check this residuosity and thus re-construct the complete p_i-th row of $\widetilde{C}$, without revealing p_i to the server. As the server is unable to distinguish quadratic residues from non-residues, the protocol leaks to the server nothing more than the number of keywords contained in a conjunctive query.

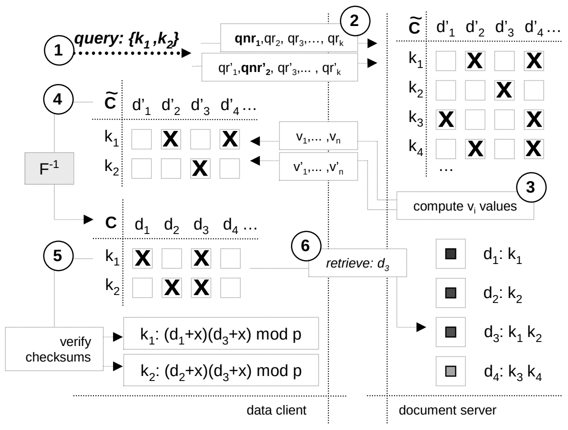

Figure 4: Oblivious Keyword Search.

The remaining steps of this solution follow exactly the protocol presented above (see Section 5.1). The client converts the p_i-th row of $\widetilde{C}$ into the p_ith row of C and uses the checksum information corresponding to k_i to verify the accuracy of the information sent by the server.

Overheads. Complete computational privacy comes at additional computation costs. First, the server is required to store $\widetilde{C}$, thus incurring a $O(nk)$ storage overhead.

For each searched keyword, the client will need to generate and send $k - 1$ QR values and one QNR ($O(k)$ costs). The server is required to perform on average $\frac{nk}{2} - n$ multiplications ($\widetilde{C}$ looks random, thus roughly half of its entries will trigger multiplications) and send back $O(n)$ data. The total computation overhead is clearly dominated by this PIR component at the server. The communication overhead is $O(k+n)$, induced by the transmission of two messages: one of size $O(k)$ followed by another one of size $O(n)$. The client can verify quadratic residuosity on the fly in not more than $O(1)$ extra storage.

Note: While more details are out of scope, it might be worth noting, that, because the same s_i values are repeatedly used in computations for each column of $\widetilde{C}$, deploying dynamic programming techniques could further reduce the number of actual multiplications performed.

This looks a bit worrisome. It is clear that obliviousness comes at an expense. But is this feasible with today's hardware? To find out, let us start by considering a set of benchmarks performed on an (obsolete) 1GHz Pentium III Processor (or its equivalent AMD Athlon) with 384Mb of RDRAM (Rambus) running at 800Mhz with ECC correction. These benchmarks report between 2500 and 3000 MIPS in Dhrystone [24] (available at [1]) and between 200 and 220 million 32-bit integer multiplications per second (reported by [22]).

Thus, for a 32-bit operation setup, things seems to be quite favorable, since a server can easily support 20-30 searches per second. The bit-size of N however has to be larger, as N's factoring would allow the server to understand each query and a type 1 leak would occur. For smaller values of N's bit-size, if the client uses a different N value for each search (or a finite set of searches) this can be further alleviated as it would require separate factoring for each of them. It is important however, that it is reasonable hard to factor N for each such *instance* and that the process is not sustainable over a large number of queries. We believe that currently this can be reasonably ensured with N size in the 256 to 512 bit range. For completeness purposes, in the following we also discuss the case of 1024 bits.

We thus set out to evaluate what can be achieved with today's cheap, off the shelf hardware, and available open source arbitrary precision libraries. For this purpose, we chose to benchmark 128, 256, 512 and 1024-bit operations on a 2.4GHz AMD Athlon64 CPU using the GMP [10] library. We naturally expected the figures above (for 32 bit operands) to scale down by a factor of 16 for 128-bit operands and 256 for 512-bit operands, in effect yielding roughly 14 million 128-bit and 0.85 million 512-bit multiplications per second in the above setup. And, given that our CPU is 2-3 times faster than this setup, we expected values roughly 2.5 higher.

Running in single-user mode on a Linux box, we obtained a speed of about 15.5 (128 bits), 4.1 (256 bits), 1.3 (512 bits) and 0.37 (1024 bits) millions of multiplications per second. Naturally, the fact that we did not achieve the expected 2.5 higher values is likely due to GMP optimization issues. We believe that in an industry-level optimized assembler implementation one could safely achieve higher speeds, e.g., at least 30 million 128-bit and 0.74 million 1024-bit operations per second on the same off the shelf hardware.

This is good news. What these results basically show is that in such a setup, even a simple PC-based server can easily support a throughput anywhere *between 4 and 360 complete oblivious keyword search queries per minute*. Let us see why. In the example above, for $n = 10000$ and $k = 1000$, the server will perform roughly $\frac{nk}{2} - n \approx 5$ million operations. Depending on the assumed size of N, this can result in roughly 360 (128 bits), 80 (256 bits), 15.5 (512 bits) and 4.5 (1024 bits) searches per minute.

We believe these numbers to be well within reasonable bounds. After all, it is unlikely that more than a small fraction of clients will submit queries at the exact same time. Additionally, it is expected that the rest of the activities of the server are mostly I/O related (e.g., serving documents over the network) which do not interfere significantly with modular arithmetic. Thus, deploying the oblivious keyword search mechanisms would easily allow one single server to handle tens to hundreds of users. Beyond that, the load can be naturally distributed on different servers by either having each serve different sets of users or simply splitting $\widetilde{C}$ between them. Moreover, a natural obliviousness-costs trade-off can be put in place, allowing the client to naturally choose a lower bit-size for N as time passes and the server "gains" more trust.

6 Discussion

6.1 Storage-Constrained Mobile Client

The oblivious keyword search method achieves a $1/k$ probability of the server guessing the searched keyword. This is the result of $O(k)$ client-side storage and $O(k+n)$ communication overhead. We now ask the following question: in the case of a severely storage-constrained (e.g., mobile) client, is it possible to reduce the storage and communication overhead (possibly at the expense of obliviousness) e.g., in scenarios involving static data? Moreover, we would also like to preserve the correct behavior of the server through the use of query assurance mechanisms. (see Section 4).

Let us adapt the solutions presented above to a scenario involving such a mobile client. In this scenario, the client transitions from an initial "home" stage, where it can process and store encrypted documents, along with resulting meta-data on the server, to a roaming behavior enabled by the usage of small mobile devices. During this stage, constrained by the scarce computational, storage and communication resources of mobile devices, the client only wants to retrieve documents matching desired keywords with minimal overhead, without adding new documents on the server. We now assume that the storage available at the client is m, such that $n < m < k$.

The main client-side storage requirement of the previous solutions is $O(k)$, one entry for each keyword. We propose a solution (see Figure 5) where instead of storing for each keyword k_i the corresponding row number p_i in $\widetilde{C}$ and the checksum c_i for verifying the accuracy of server answers, the client moves this information, kept in an ordered fashion, to the server.

Specifically, the client generates two keys, sk and key and a random value R and keeps them secret. During the initial stage, the client generates a record $E_{sk}(k_i, c_i, R)$ for each keyword k_i, and associates a value, $H(key, k_i)$, to the record. The client first orders the records according to $H(key, k_i)$, and then removes the auxiliary $H(key, k_i)$ values used for sorting and stores the ordered list of records, L, on the server. R is used by the client to randomize records. Moreover, the row index p_i of a keyword k_i is set to be the rank of $H(key, k_i)$ in the ordered list of records, $p_i = rank(H(key, k_i), L)$. That is, the row $\widetilde{C}_{p_i}$ of $\widetilde{C}$ encrypts information about documents containing keyword k_i. Due to the secret ordering of records, the server cannot build one-to-one mappings between keywords and records or keywords and rows in $\widetilde{C}$, even when the server has a dictionary of all the keywords. Finally, the client divides L into blocks of m keywords and $\widetilde{C}$ into corresponding blocks of m rows.

To obtain from the server the record of keyword k_i, without knowing and also without revealing its rank in L and in $\widetilde{C}$, the client generates $H(key, k_i)$ and performs a binary search using $H(key, k_i)$ in L, which is ordered by the $H(key, k_j)$ value of all keywords. For this, the client asks the server for the first and last entries of the middle block in L, corresponding to keywords k_f and k_l, and decrypts them. If $H(key, k_i)$ is ranked between the $H(key, k_f)$ and $H(key, k_l)$, the client requests the entire block of m records of L from the server. Subsequently, the client performs a binary search for $H(key, k_i)$ inside the block (requiring only $O(\log m)$ decryptions instead of $O(m)$). However, if $H(key, k_i)$ is ranked before $H(key, k_f)$, the client recursively searches the first half of L, otherwise it recursively searches the second half of L.

After retrieving the record of k_i, $E_{sk}(k_i, c_i, R)$, the client computes the rank p_i of $H(key, k_i)$ in L, according to the search pattern previously followed. Moreover, the client knows the block number b_i containing the record of k_i. The b_ith block of m rows of $\widetilde{C}$ contains the row corresponding to keyword k_i, at rank $r_i = p_i \ mod \ m$. Then, following the oblivious keyword search protocol (see Section 5.2), the client generates m numbers, $s_1, s_2, .., s_m$, where only s_{r_i} is a quadratic non-residue while the others are quadratic residues and sends them to the server, along with b_i. The server processes only the b_ith block of $\widetilde{C}$ using $s_1, s_2, .., s_m$ (see Section 5.2) and sends the client the n resulting numbers, one for each document. The client then reconstructs the desired row of $\widetilde{C}$ by checking quadratic residuosity.

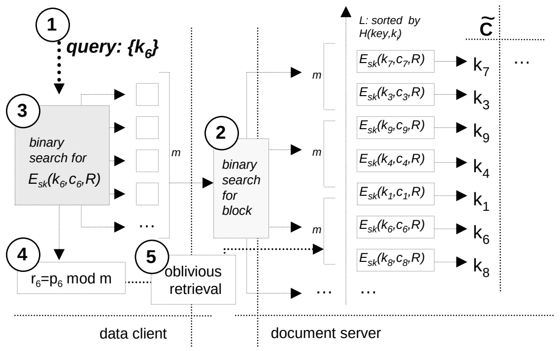

Figure 5: A storage-constrained client will place the encrypted checksums on the server and access it by performing two rounds of binary search. This comes at the expense of complete obliviousness.

Leaks. This solution prevents the server from learning the exact keyword searched, with a precision of m entries. That is, the client still leaks correlations between blocks containing m entries corresponding to arbitrary keywords and documents matching some keywords inside the blocks.

Overheads. The server storage overhead is $O(k)$. For each searched keyword, the client needs to generate and send $m - 1$ QR values and one QNR ($O(m)$). Subsequently, it will need to perform $O(\log k/m + \log m)$ decryptions and n quadratic residuosity verifications (cost $O(n)$) per searched keyword. Thus, the computation overhead is $O(n + \log k/m)$. The communication overhead can be broken down into: a message composed of m integers, followed by $\log k/m$ short messages, followed by a final message of size $O(n)$. This becomes $O(\log k/m + n)$.

7 Conclusions

In this paper we addressed query assurance and privacy in a *dynamic* outsourced data framework in the presence of (not only lazy but also) potentially malicious data servers. Our results focused on conjunctive keyword search scenarios, in which outsourced documents are retrieved based on keyword searches submitted by a client to a file or email server.

We proposed a set of protocols in which the client is provided with cryptographic assurances proving the authenticity and completeness of server replies. These protocols also strengthen the privacy guarantees: not only is the outsourced data hidden (from the server) but so are client access patterns, the queries themselves (even in encrypted form), and the association between previously searched keywords and returned documents or between newly added documents and their corresponding keywords.

References

[1] activewin.com. ActiveWin.com CPU Reviews. Online at `http://www.activewin.com/reviews/`, 2005.

[2] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS'93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM Press, 1993.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[4] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of Eurocrypt 2004*, pages 506–522. LNCS 3027, 2004.

[5] Richard Brinkman, Jeroen Doumen, and Willem Jonker. Using secret sharing for

searching in encrypted data. In *Secure Data Management*, 2004.

[6] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. Cryptology ePrint Archive, Report 2004/051, 2004. `http://eprint.iacr.org/`.

[7] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of FOCS*. IEEE Computer Society, 1995.

[8] Benny Chor and Niv Gilboa. Computationally private information retrieval (extended abstract). In *Proceedings of STOC*. ACM Press, 1997.

[9] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 41–50, 1995.

[10] GNU. GNU Multiple Precision Arithmetic Library. Online at `http://www.swox.com/gmp/`, 2005.

[11] Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. `http://eprint.iacr.org/2003/216/`.

[12] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4), October 1986.

[13] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[14] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, (28):270–299, 1984.

[15] P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In *Proceedings of ACNS*, pages 31–45. Springer-Verlag; Lecture Notes in Computer Science 3089, 2004.

[16] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 216–227. ACM Press, 2002.

[17] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings of FOCS*. IEEE Computer Society, 1997.

[18] M Luby and C Rackoff. Pseudo-random permutation generators and cryptographic composition. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 356–363, New York, NY, USA, 1986. ACM Press.

[19] National Institute of Standards and Technology (NIST). The secure hash standard. Online at `www.nist.gov/sha/`.

[20] R. Ostrovsky. Efficient computation on oblivious RAMs. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 514–523, New York, NY, USA, 1990. ACM Press.

[21] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *Proceedings of STOC*. ACM Press, 1997.

[22] PassMark Software. The PassMark CPU Review Benchmarks. Online at `http://www.passmark.com/cpureview/`, 2004.

[23] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P 2000)*. IEEE Computer Society, 2000.

[24] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.