RDAS: A Symmetric Key Scheme for Authenticated Query Processing in Outsourced Databases

Lil María Rodríguez-Henríquez, Debrup Chakraborty

Departamento de Computación, CINVESTAV-IPN
Av. Instituto Politécnico Nacional No. 2508, Col. San Pedro Zacatenco, México, D.F. 07300
email: lrodriguez@computacion.cs.cinvestav.mx, debrup@cs.cinvestav.mx

Abstract. In this paper we address the problem of authenticated query processing in outsourced databases. An authenticated query processing mechanism allows a client to verify the validity of the query responses that it gets from an un-trusted and remote server, who stores the client's database on its behalf. We introduce a general framework called RDAS for the problem of authenticated query processing, and define the security goals for this task in line with concrete provable security. We propose several schemes which enable a client to verify both the completeness and correctness of the query responses of a server. All the schemes follow the proposed framework and are provably secure in terms of the proposed security definition. The novelty of the proposed schemes is that they use bitmap indexes as a main component for providing authentication. Bitmap indexes have recently seen lot of applications for accelerated query processing and many commercial databases implement such indexes. Bitmaps have not been previously used for a security goal. We show that the proposed schemes can match in both functionality and efficiency compared to the existing schemes. We also implement the schemes on a real database and provide extensive experimental studies on the schemes.[1]

1 Introduction

Cloud computing holds the promise of revolutionizing the manner in which enterprises manage, distribute, and share information. The data owner (client) can out-source almost all its information processing tasks to a "cloud". The cloud can be seen as a collection of servers (we shall sometimes refer to it as the server) which caters the data storage, processing and maintenance needs of the client. Needless to say this new concept of computing has already brought significant savings in terms of costs for the data owner.

Among others, an important service provided by a cloud is *Database as a Service (DAS)*. In this service the client delegates the duty of storage and maintenance of his/her data to a third party (an un-trusted server). This model has gained lot of popularity in the recent times. The DAS model allows the client to perform operations like create, modify and retrieve from databases in a remote location [9]. These operations are performed by the server on behalf of the client. However, delegating the duty of storage and maintenance of data to a third party brings in some new security challenges.

The two main security goals of cryptography are privacy and authentication. These security issues are relevant to the outsourced data also. The client who keeps the data with an untrusted server has two main concerns. The first one being that the data may be sensitive and the client may not want to reveal the data to the server and the second one is the data whose storage and maintenance has been delegated to the server would be used by the client. The typical usage of the data would be that the client should be able to query the database and the answers to

[1] An abridged version of this work appears as:
Lil María Rodríguez-Henríquez, Debrup Chakraborty: RDAS: A Symmetric Key Scheme for Authenticated Query Processing in Outsourced Databases. Workshop on Security and Trust Management, STM 2013, *Lecture Notes in Computer Science* vol. 8203, pp. 115-130, 2013

the client's queries would be provided by the server. It is natural for the client to be concerned about a malicious server who does not provide correct answers to the client queries. In this work we are interested in this problem. We aim to devise a scheme in which the client would be able to verify whether the server is responding correctly to its queries.

The problem of interest is of data authentication, and there are well known cryptographic solutions to the basic data authentication problem. In the symmetric key setting this has been addressed by the use of message authentication codes and in the asymmetric key setting signature schemes provide this functionality.

We are interested in the problem of authenticated query processing in the context of relational databases. We consider the scenario where a client delegates a relational database to an untrusted server. When the client queries its outsourced data, it expects in return a set of records (query reply) satisfying the query's predicates. As the server is not trusted, so it must be capable of proving the correctness of its responses. In other words, a malicious server may attempt to insert fake records into the database, modify existing records or simply skip some of them from the query response. Hence there must exist a mechanism, which can protect the client from such malicious server behavior. We describe the intricacies of the problem with the help of an example. Consider the relational database of employees data shown in Table 1.

`EmpId`	`Name`	`Gender`	`Level`
TRW	Tom	M	L_2
MST	Mary	F	L_1
JOH	John	M	L_2
LCT	Lucy	F	L_1
ASY	Anne	F	L_1
RZT	Rosy	F	L_2

Table 1. Relation $R1$ (This relation would serve as a running example).

We consider that this relation has been delegated by a client to a server, and the client poses the following query

$$\texttt{SELECT * FROM } R1 \texttt{ WHERE } Gender = \texttt{'}M\texttt{' OR } Level = \texttt{'}L_2\texttt{'}.$$

The correct response to this query is the set Res consisting of three tuples

$$\mathsf{Res} = \{(\text{TRW, Tom, M}, L_2), (\text{JOH, John, M}, L_2), (\text{RZT, Rosy, F}, L_2)\}.$$

In answering the query the server can act maliciously in various ways. In the context of authentication, we are concerned with two properties of the response namely *correctness* and *completeness*. Correctness and completeness denote two different malicious activities of the server, we explain these notions with an example below:

1. **Incorrect result:** The server responds with three tuples, but changes the tuple $(\text{TRW, Tom, M}, L_2)$, with $(\text{TRW, Tom, F}, L_2)$. Moreover, it can be the case that the server responds with $\mathsf{Res} \cup \{(\text{BRW, Bob, M}, L_2)\}$, i.e., it responds with an extra tuple which is not a part of the original relation.
2. **Incomplete result:** The server may not respond with the complete result, i.e., it can delete some valid results from the response, i.e., instead of responding with Res it responds with $\mathsf{Res} - \{(\text{TRW, Tom, M}, L_2)\}$.

It is to be noted that incomplete results are also incorrect, but we differentiate the two scenarios (as it has been previously done in the literature) by the fact that in an incorrect result the server inserts something which is not in the database, and in case of an incomplete answer the answer is correct but is not complete, in the sense that the server drops some valid tuples from the correct response. A client must be able to verify both correctness and completeness of a response.

The problem of correctness can be easily handled in the symmetric setting by adding a message authentication code to each tuple. A secure message authentication code is difficult to forge, and thus this property would not allow the server to add fake entries in its response. The completeness problem is more difficult and its solution is achieved through more involved schemes.

The problem of query completeness has been largely addressed by some interesting use of *authenticated data structures.* The basic idea involved is to store the information already present in the relation in a different form using some special data structures. This redundancy along with some special structural properties of the used data structures help in verifying completeness.

A large part of the literature uses tree based authentication structures like the Merkeley hash tree [17] or its variants. Some notable works in this direction are reported in [17, 14, 5, 7, 18, 23, 25, 36]. These techniques involve using a special data structure along with some cryptographic authentication mechanism like hash functions and/or signatures schemes. The tree based structures yield reasonable communication and verification costs. But, in general they require huge storage at server side, moreover the query completeness problem is largely addressed with respect to range queries and such queries may not be relevant in certain scenarios, say in case of databases with discrete attributes which do not have any natural metric relationship among them.

Signature schemes have also been used in a novel manner for solving the problem. One line of research has focussed on aggregated signatures [20–22, 26, 29]. Signature aggregation helps in reducing the communication cost to some extent and in some cases can function with constant extra communication overhead. A related line of research uses chain signatures. If one uses chain signatures as in [21], the use of specialized data structures may no longer be required.

Though there has been considerable amount of work on authenticated query processing on relational databases, but it has been acknowledged (for example in [37]) that the problem of query authentication largely remains open. An unified cryptographic treatment of the problem is missing in the literature. In most existing schemes cryptographic objects have been used in an ad-hoc manner, and the security guarantees that the existing schemes provide are not very clear. In this work we initiate a formal cryptographic study of the problem of query authentication in a distinct direction. We propose a new scheme which does not use any specialized data structure to address the completeness problem. Our solution involves usage of bitmap indices for this purpose. Bitmap indices have gained lot of popularity in the current days for their use in accelerated query processing [2, 32], and many commercially available databases like Oracle, IBM DB2, Sybase IQ now implement some form of bitmap index scheme in addition to the more traditional B-tree based schemes, thus it may be easy to incorporate a bitmap based scheme in a modern database without significant extra cost. To our knowledge, bitmaps have not been used till date for a security goal.

In addition to bitmap indices we use a secure message authentication code (MAC) as the only cryptographic object. We show that by the use of these simple objects one can design a query authentication scheme which allows verification of both correctness and completeness of query results. As the basic cryptographic object is a symmetric key primitive, thus our scheme does not provide public verifiability. Moreover, in this work, we restrict ourself to static databases only. We see private verifiability of our scheme more as a design goal than a limitation, as we believe that there exist scenarios where public verifiability may not be required, and in such scenarios it is better not to use the heavy machinery of public key signatures which uses computationally intensive number theoretic operations, whereas computational overheads of symmetric key message authentication schemes are minimal.

Extension of our scheme to dynamic scenarios may be possible, but in this current work we do not further deal with such possibilities, we plan to discuss the extension of our scheme to dynamic scenarios in a separate work. There exist many static transactional databases, say databases related to data warehousing applications, where efficient authenticated query processing may be required.

Next, we summarize the concrete contributions in this paper :

1. We define a generic scheme which we call as relational database authentication scheme (RDAS) which would provide the functionality of authenticated query processing. We carefully define the security goals of RDAS in line with the tradition of concrete provable security. The security definition encompasses both correctness and completeness of a query response. Such a definition is new to the literature, and we hope that this definition would help to evaluate security of existing schemes.
2. We propose an RDAS called $\mathsf{RDAS1}$. $\mathsf{RDAS1}$ is designed using message authentication codes and bitmap indices in a novel manner. $\mathsf{RDAS1}$ is capable of authenticated query processing of simple select queries and select queries involving disjunctions of equality conditions. The extra overhead for using $\mathsf{RDAS1}$ both in terms of extra bandwidth and computation cost is not significant. We formally prove that $\mathsf{RDAS1}$ provides authentication in accordance to our security definition.
3. Though $\mathsf{RDAS1}$ is efficient and secure, a serious limitation of it is that it can only authenticate a restricted class of queries. We propose a provably secure modification called $\mathsf{RDAS2}$ which is capable of authenticating a large class of queries but it has more storage and communication overhead than $\mathsf{RDAS1}$. Unlike in $\mathsf{RDAS1}$, in $\mathsf{RDAS2}$ one needs to explicitly store bitmap indexes and also send some bitmap indexes along with the query responses. This leads to increased storage and communication costs. We discuss methods to compress bitmaps, which can lead to significant savings in case of RDAS2.
4. We provide extensions of both $\mathsf{RDAS1}$ and $\mathsf{RDAS2}$ using the functionality of aggregate MACs. We call these schemes as $\mathsf{RDAS1\text{-}agg}$ and $\mathsf{RDAS2\text{-}agg}$. These schemes are more efficient in terms of communication costs compared to the basic schemes.
5. We provide extensive experimental results on all the schemes on a real database, and provide hard experimental data on various performance measures. Our experiments suggests that the RDAS framework can be a viable option for practical authenticated query processing.

2 Preliminaries and Notations

Relations. In what follows, by $R(A)$ we would denote a relation over a set of attributes A. If $A = \{a_1, a_2, \cdots a_n\}$, we shall sometimes write $R(a_1, a_2, \cdots, a_n)$ instead of $R(A)$. We will assume that each attribute has a set of permitted values, i.e., the domain of the attribute. Given an attribute a, $\mathsf{Dom}(a)$ would represent its domain. We are mainly concerned with attributes whose domains are finite, note that for a static database each attribute always has a finite domain. By cardinality of an attribute we shall mean the cardinality of the domain of the attribute. We will denote the cardinality of an attribute a by $\mathsf{Card}(a) = |\mathsf{Dom}(a)|$.

A tuple t in a relation over a set of attributes is a function that associates with each attribute a value in its specific domain. Specifically if $A = \{a_1, a_2, \cdots a_n\}$ and $R(A)$ be a relation then the

j^{th} tuple of relation $R(A)$ would be denoted by t_j^R and for $a_i \in A$ by $t_j^R[a_i]$ we shall denote the value of attribute a_i in the j^{th} tuple in R. For $B \subseteq A$, $t_j^R[B]$ will denote the set of values of the attributes in B in the j^{th} tuple. We shall sometimes omit the subscripts and superscripts from t_j^R and denote the tuple by t if the concerned relation is clear from the context and the tuple number is irrelevant.

Binary strings: The set of all binary strings would be denoted by $\{0,1\}^*$, and the set of n bit strings by $\{0,1\}^n$. For $X_1, X_2 \in \{0,1\}^*$, by $X_1||X_2$ we shall mean the concatenation of X_1 and X_2; and $|X_1|$ will denote the length of X_1 in bits. By $\mathsf{pad}_\ell(X)$ we will denote the operation of padding ℓ zeros to the end of X, i.e., $\mathsf{pad}_\ell(L) = L||0^\ell$. Also, if $|X| \leq \ell$, by $\mathsf{cpad}_\ell(X)$ we shall mean $\mathsf{pad}_{\ell-|X|}(X)$. By $\mathsf{bit}_i(X)$ we will denote the i^{th} bit of X. Given $x \in \{0,1\}^*$ the procedure $\mathsf{parse}_n(x)$ will break x into constituent n bit strings, i.e, $\mathsf{parse}_n(x) = (x_1, x_2, \ldots, x_m)$, where $x = x_1||x_2||\ldots||x_m$ and $|x_i| = n$ for $1 \leq i \leq m-1$ and $|x_m| \leq n$, thus making $m = \lceil \frac{|x|}{n} \rceil$.

We shall always consider that the domains of all attributes in the relations are subsets of $\{0,1\}^*$, this convention would allow us to apply transformations and functions on the values of the tuples in a relation without describing explicit encoding schemes.

By $\mathbb{F}_q$ we shall mean a finite field with q elements. For our purpose we shall be interested in the field $\mathbb{F}_{2^n}$ for some n. n bit strings can be represented by a polynomial which coefficients are in $\mathbb{F}_2$. For example, if $A \in \{0,1\}^n$ such that $A = a_0, a_1, \cdots, a_{n-1}$ where each $a_i \in \{0,1\}$ then A can be represented by the polynomial $A(x) = \sum_{i=0}^{n-1} a_i x^i$. Thus the set of all n bit strings can be treated as the field $\mathbb{F}_{2^n}$ where the addition is defined as the *xor* of the strings and multiplication is defined as the multiplication of the two polynomials corresponding to the strings modulo a fixed n degree irreducible polynomial $\tau(x)$.

Bitmaps: Consider a relation $R(a_1, \ldots, a_m)$ with nT many rows. Consider that for each attribute a_i, $\mathsf{Dom}(a_i) = \{v_1^i, v_2^i, \ldots v_{\lambda_i}^i\}$, thus $\mathsf{Card}(a_i) = \lambda_i$ for $1 \leq i \leq m$. We define the bitmap of an attribute a_i corresponding to its value v_j^i in the relation R as $\mathsf{BitMap}_R(a_i, v_j^i) = X$, where X is a binary string, such that $|X| = \mathsf{nT}$ and for $1 \leq k \leq \mathsf{nT}$,

$$\mathsf{bit}_k(X) = \begin{cases} 1 & \text{if } t_k^R[a_i] = v_j^i \\ 0 & \text{otherwise.} \end{cases}$$

This would be more clear with an example. Consider the specific relation $R1$ on the attributes $\{\mathtt{EmpID}, \mathtt{Name}, \mathtt{Gender}, \mathtt{Level}\}$ as shown in Table 1. $\mathsf{Dom}(\mathtt{Gender}) = \{\text{M, F}\}$ and $\mathsf{Dom}(\mathtt{Level}) = \{L_1, L_2\}$.

From this relation we can compute the following bitmaps

$$\begin{aligned} \mathsf{BitMap}_{R1}(\mathtt{Gender}, \text{F}) &= 010111 \\ \mathsf{BitMap}_{R1}(\mathtt{Gender}, \text{M}) &= 101000 \\ \mathsf{BitMap}_{R1}(\mathtt{Level}, L_1) &= 010110 \\ \mathsf{BitMap}_{R1}(\mathtt{Level}, L_2) &= 101001. \end{aligned}$$

Message authentication codes: Message authentication codes provide authentication in the symmetric key setting. It is assumed that the sender and the receiver share a common secret key K. Given a message x, the sender uses K to generate a footprint of the message. This footprint (commonly called a tag) is the message authentication code (MAC) for the message x. The sender transmits the pair $(x; \mathsf{tag})$ to the receiver. The receiver uses K to verify that (x, tag) is a properly generated message-tag pair. In most cases, verification is performed by regenerating the tag on the message x and comparing the generated tag with the one received. In what follows we shall call the algorithm for generating the tag as a MAC, thus assuming that the size of the tag is τ bits, we see the tag generation scheme as a function $\mathsf{MAC} : \mathcal{K} \times \mathcal{M} \to \{0,1\}^\tau$, where $\mathcal{K}$ and $\mathcal{M}$ are the key and message spaces respectively. In most cases we shall write $\mathsf{MAC}_K(x)$ instead of $\mathsf{MAC}(K, x)$.

An attack on a MAC scheme signifies forging a message-tag pair. The types of attacks which are important for MAC schemes can be formally described as an interaction of an adversary $\mathcal{A}$ and the procedure MAC. $\mathcal{A}$ is given an oracle access to the MAC generation procedure $\mathsf{MAC}_K(.)$, instantiated with a randomly generated key K, which is unknown to $\mathcal{A}$. $\mathcal{A}$ can query $\mathsf{MAC}_K(.)$ with messages of its choice, and for each query x it gets $\mathsf{MAC}_K(x)$ as a response. Let us assume that $\mathcal{A}$ queries with the messages $x_1, x_2, \ldots, x_q$ and gets $y_1, \ldots, y_q$ as the responses. In the end, $\mathcal{A}$ produces a pair $(\tilde{x}, \tilde{y})$, such that $\tilde{x} \notin \{x_1, x_2, \ldots, x_q\}$. It is said that $\mathcal{A}$ had committed a successful *forgery* if $\mathsf{MAC}_K(\tilde{x}) = \tilde{y}$. We define the advantage of the adversary $\mathcal{A}$ in forging the message authentication code MAC as follows:

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathsf{MAC}}(\mathcal{A}) = \Pr[\mathcal{A} \text{ forges }]. \tag{1}$$

The probability is taken over the random choice of the key K and the randomness of the adversary.

3 Relational Database Authentication Scheme (RDAS): Definitions and Basic Notions

A relational database authentication scheme (RDAS) consists of a tuple of algorithms $(\mathcal{K}, \mathcal{F}, \Phi, \Psi, \mathcal{V})$, which are described in details in the following paragraphs.

$\mathcal{K}$ is the **key generation algorithm** and it selects one (or more) keys from a pre-specified key space and outputs them.

$\mathcal{F}$ is called the **authentication transform**, which takes in a set of relations $\mathcal{R}$ and a set of keys and outputs another set of relations $\mathcal{R}'$ along with some additional data (M_s, M_c). If the set of keys is K, we shall denote this operation as $(\mathcal{R}', M_c, M_s) \leftarrow \mathcal{F}_K(\mathcal{R})$. A client who wants to store the set of relations $\mathcal{R}$ in an un-trusted server, transforms $\mathcal{R}$ to $\mathcal{R}'$ using the authentication transform $\mathcal{F}$ and a set of keys. The transform $\mathcal{F}$ produces some additional data other than the set of relations $\mathcal{R}'$, the additional data consists of two distinct parts M_s and M_c. The set of relations $\mathcal{R}'$ along with M_s are stored in the server and the keys and the data M_c are retained in the client. The key generation algorithm and the authentication transform are executed in the client side.

We call Φ as the **query translator**, it is a transformation which takes in a query for the relations in $\mathcal{R}$ and converts it into a query for relations in $\mathcal{R}'$. For ease of discussion we shall refer a query for $\mathcal{R}$ to be a $\mathcal{R}$-query and a query for $\mathcal{R}'$ to be a $\mathcal{R}'$-query. Thus, given a $\mathcal{R}$-query q, $\Phi(q)$ would be a $\mathcal{R}'$-query. Thus by use of the transform Φ, the client would be able to translate queries meant for $\mathcal{R}$ to queries which can be executed on the transformed relations in $\mathcal{R}'$.

Ψ is the **response procedure**. To execute a query q on $\mathcal{R}$, the client converts the query to $\Phi(q)$ and sends it to the server. The server executes the function Ψ, which takes in the query $\Phi(q)$ and uses $\mathcal{R}'$ and M_s. The output of Ψ is ρ, which we call as the response of the server. The server returns its response S to the client.

The **verification procedure** is a keyed transform $\mathcal{V}_K$ which runs in the client. It takes as input the query q, a response S of the server and M_c and outputs either an answer ans for the query q or outputs a special symbol $\perp$ which signifies reject.

3.1 Correctness and security

If we fix the set of relations $\mathcal{R}$, then an $\mathcal{R}$-query q when executed in $\mathcal{R}$ would have a fixed answer say $\mathsf{ans}(\mathcal{R}, \mathsf{q})$. Our goal is to transform $\mathcal{R}$ to $\mathcal{R}'$ using an RDAS in such a way that if the query $\Phi(q)$ is sent to the server, then the answer ans should be recoverable from the server response ρ through the procedure $\mathcal{V}$, if the server follows the protocol correctly. On the other hand, if the server is malicious, i.e., it deviates from the protocol and sends a response ρ' distinct from the correct response ρ then the procedure $\mathcal{V}$ should reject the response by outputting $\perp$. In other words, if the answer to a $\mathcal{R}$-query is ans, then after running the protocol, $\mathcal{V}$ will either produce ans or $\perp$, it would not produce an answer ans' distinct from ans.

In the security model, we allow the adversary to choose the primary set of relations $\mathcal{R}$. Given this choice of $\mathcal{R}$, we compute $(\mathcal{R}', M_c, M_s) \leftarrow \mathcal{F}_K(\mathcal{R})$, for a randomly selected set of keys K which is unknown to the adversary. We give $\mathcal{R}'$ and M_s to the adversary. The adversary chooses an $\mathcal{R}$-query q and the challenger provides the adversary with $\Phi(q)$, finally the adversary outputs a response ρ, and we say that the adversary is *successful* if $\mathcal{V}_K(\rho, q, M_c) \notin \{\perp, \mathsf{ans}(\mathcal{R}, \mathsf{q})\}$.

Definition 1. *Let* $\mathsf{Succ}_{\mathcal{A}}$ *be the event that a specific adversary* $\mathcal{A}$ *is successful in the sense as described above. We say that an RDAS is* (ϵ, t)*-secure if for any adversary* $\mathcal{A}$ *which runs for time at most* t $\Pr[\mathsf{Succ}_{\mathcal{A}}] \leq \epsilon$.

Some immediate observations regarding this security definition are as follows:

1. **Encompasses both correctness and completeness**: An important thing to note is that the security definition covers both completeness and correctness, as RDAS is considered secure if the verification algorithm does not accept (except with some small probability) a wrong response.
2. **Concrete security and adversarial resources**: The definition follows the paradigm of concrete security, where we specify the running time and the probability of success of an adversary. An (ϵ, t)-secure RDAS is really secure where t is "reasonable" and ϵ is "small". As is common in the paradigm of "concrete security", we do not precisely define "reasonable" and "small". These are interpreted in the context.

4 RDAS1: A generic scheme for select queries involving arbitrary disjunctions

We discuss a basic scheme for a secure RDAS which works only if the queries made are single attribute select queries or select queries involving disjunctions of an arbitrary number of equality conditions. We call this scheme as RDAS1. RDAS1 can be modified to handle certain other class of queries, but for the sake of simplicity we just concentrate on a scheme which works on disjunction queries. In the later sections we would discuss several extensions of RDAS1 which can handle other types of queries and provide additional functionalities.

We describe the scheme assuming that the set of initial relations $\mathcal{R}$ is a singleton set consisting of a single relation $R(B)$, where $B = \{b_1, b_2, \ldots, b_{|B|}\}$ is the set of attributes, and consider $A = \{a_1, \ldots, a_m\} \subseteq B$ to be a set of attributes on which queries are allowed, we shall call A the set of allowed attributes. Note, it is possible that $B = A$. The procedure $\mathcal{F}$ converts R into two relations R_α and R_β, i.e, $\mathcal{R}' = \{R_\alpha, R_\beta\}$ and M_s is empty and $M_c = \mathsf{nT}$, where nT is the number of tuples in R. The only cryptographic object used by RDAS1 is a message authentication code $\mathsf{MAC} : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^\tau$, where $\mathcal{K}$ is the key space. Next, we discuss the details of each of the procedures involved in RDAS1. In what follows, we shall describe the procedures considering a generic relation $R(B)$, where the set of allowed attributes is $A \subseteq B$. Also for ease of exposition we shall throughout consider the relation $R1$ as depicted in Table 1 as a concrete example, and for simplicity, for $R1$ we shall consider the set of allowed attributes to be $\{\texttt{Gender}, \texttt{level}\}$.

$\mathsf{RDAS1}.\mathcal{K}$: The key space for $\mathsf{RDAS1}$ is the same as the key space of the associated message authentication code MAC. The key generation algorithm selects a key K uniformly at random from $\mathcal{K}$.

$\mathsf{RDAS1}.\mathcal{F}$: $\mathcal{F}$ produces two relations R_α and R_β by the action of the key. The relation R_α is defined on the set of attributes $B \cup \{\texttt{Nonce}, \texttt{Tag}\}$, i.e., R_α has two more attributes than in R. If R contains nT many tuples then R_α also contain the same number of tuples. The procedure for populating the tuples of R_α is depicted in Figure 1. Basically, what this procedure does is compute a MAC for each row.

```
Creating $R_\alpha$
1. for $j = 1$ to $\mathsf{nT}$
2.     for $i = 1$ to $|B|$
3.         $t_j^{R_\alpha}[b_i] \leftarrow t_j^R[b_i]$;
4.     end for
5.     $t_j^{R_\alpha}[\texttt{Nonce}] \leftarrow j$;
6.     $H \leftarrow t_j^R[b_1]|| \ldots ||t_j^R[b_m]||j$;
7.     $t_j^{R_\alpha}[\texttt{Tag}] \leftarrow \mathsf{MAC}_K(H)$;
8. end for
```

```
Creating $R_\beta$
1. for $j = 1$ to $N$
2.     $t_j^{R_\beta}[\texttt{Name}] \leftarrow \mathcal{L}_j^1$;
3.     $t_j^{R_\beta}[\texttt{SearchKey}] \leftarrow \mathcal{L}_j^2$;
4.     $t_j^{R_\beta}[\texttt{RowNo}] \leftarrow \mathsf{nT} + j$;
5.     $L \leftarrow \mathcal{L}_j^1||\mathcal{L}_j^2||\mathsf{BitMap}_R(\mathcal{L}_j^1, \mathcal{L}_j^2)||(\mathsf{nT} + j)$;
6.     $t_j^{R_\beta}[\texttt{Tag1}] \leftarrow \mathsf{MAC}_K(L)$;
7. end for
```

Fig. 1. Creating R_α and R_β

The relation R_β contains the attributes $\{\texttt{Name}, \texttt{SearchKey}, \texttt{RowNo}, \texttt{Tag1}\}$, irrespective of the attributes in relation R. Where $\mathsf{Dom}(\texttt{Name}) = \{a_1, \ldots, a_m\}$, i.e., the allowed attributes in R. And, $\mathsf{Dom}(\texttt{SearchKey}) = \mathsf{Dom}(a_1) \cup \mathsf{Dom}(a_1) \cup \cdots \cup \mathsf{Dom}(a_m)$. Let $\Omega = \cup_{i=1}^m (\{a_i\} \times \mathsf{Dom}(a_i))$, note that the elements of Ω are ordered pairs of the form (x, y) where $x \in \mathsf{Dom}(\texttt{Name})$ and $y \in \mathsf{Dom}(\texttt{SearchKey})$, and $|\Omega| = \sum_{i=1}^m \mathsf{Card}(a_i) = N$. Let $\mathcal{L}$ be a list of the elements in Ω in an arbitrary order. If (x, y) be the i-th element in $\mathcal{L}$, then we shall denote x and y by $\mathcal{L}_i^1$ and $\mathcal{L}_i^2$ respectively, where $1 \leq i \leq N$. The way the relation R_β is populated is also shown in Figure 1. This procedure allows the client to store all possible pairs $\mathcal{L}_i^1, \mathcal{L}_i^2$ along with the MAC calculated over this pair concatenated with the respective bitmap and $\texttt{RowNo}$. Note that the bitmap is not explicitly stored in the relation R_β. The transform $\mathcal{F}$ is executed in the client side, and the resulting relations R_α and R_β are stored in the server.

For a concrete example, if $\mathsf{RDAS1}.\mathcal{F}$ has as input the relation $R1$ (see Table 1) and the set of allowed attributes is $\{\texttt{Gender}, \texttt{level}\}$, then it would produce as output the relations $R1_\alpha$ and $R1_\beta$ as shown in Table 2. The relation $R1_\alpha$ is almost the same as that of $R1$, except that it has two additional attributes, $\texttt{Nonce}$ and $\texttt{Tag}$. The attribute $\texttt{Nonce}$ just contains the row numbers and is thus unique for each row. The attribute $\texttt{Tag}$ is the message authentication code computed for a message which is produced by concatenating all the values of the attributes in that tuple.

Relation $R1_\alpha$

EmpId	Name	Gender	Level	Nonce	Tag
TRW	Tom	M	L_2	1	Y_1
MST	Mary	F	L_1	2	Y_2
JOH	John	M	L_2	3	Y_3
LCT	Lucy	F	L_1	4	Y_4
ASY	Anne	F	L_1	5	Y_5
RZT	Rosy	F	L_2	6	Y_6

Relation $R1_\beta$

Name	SearchKey	RowNo	Tag1
Gender	F	7	Y_7'
Gender	M	8	Y_8'
Level	L_1	9	Y_9'
Level	L_2	10	Y_{10}'

Table 2. Relations $R1_\alpha$ and $R1_\beta$

The relation $R1_\beta$ contains the attributes $\{\texttt{Name}, \texttt{SearchKey}, \texttt{RowNo}, \texttt{Tag1}\}$, where in this case, $\mathsf{Dom}(\texttt{Name}) = \{\texttt{Gender}, \texttt{Level}\}$, $\mathsf{Dom}(\mathsf{SearchKey}) = \{M, F\} \cup \{L_1, L_2\}$. The tuples in $R1_\beta$ are populated according to the procedure as shown in Figure 1(b), and the specific relation $R1_\beta$ is shown in Table 2.

RDAS1.Φ: The transform Φ, transforms a query meant for the original relation R to a set of queries which are meant to be executed on the relations R_α and R_β which are stored in the server side. RDAS1 can authenticate only certain types of queries, the allowed queries for RDAS1 are of the following form:

Q: `SELECT * FROM` R `WHERE` $a_1 = v_1$ `OR` $a_2 = v_2$ `OR ...... OR` $a_l = v_l$

The allowed set of queries are thus select queries on arbitrary numbers of disjunctions on different or repeated attributes[2], which includes select queries on a single attribute of the form `SELECT * FROM` R `WHERE` $a_i = v$. Given as input a valid query q, $\Phi(q)$ outputs two queries one for the relation R_α (which we call q_α) and the other for R_β (which we call q_β). For the specific query Q, $\Phi(Q)$ will output the following queries:

Q_α: `SELECT * FROM` R_α `WHERE` $a_1 = v_1$ `OR` $a_2 = v_2$ `OR ...... OR` $a_l = v_l$
Q_β: `SELECT * FROM` R_β `WHERE (Name` $=$ a_1 `AND SearchKey` $=$ v_2`) OR ...... OR ( Name` $=$ a_l `AND SearchKey` $= v_l$`)`

Going back to the concrete example, consider the following query $Q1$ on the relation $R1$

$Q1$: `SELECT * FROM` $R1$ `WHERE Gender = 'M' OR Level= '`L_2`'`

After applying the transformation $\Phi(Q1)$, the output queries $Q1_\alpha$ and $Q1_\beta$ would be the following:

$Q1_\alpha$: `SELECT * FROM` $R1_\alpha$ `WHERE` $Gender =$ `'M' OR` $Level =$ `'`L_2`'`
$Q1_\beta$: `SELECT * FROM` $R1_\beta$ `WHERE (`$Name =$ `'Gender' AND` $Searchkey =$`'M') OR (`$Name =$ `'Level' AND` $Searchkey =$`'`L_2`')`

The reason for the specific structure of the q_β queries would be clear from the description of the verification process and the associated example.

RDAS1.Ψ: As discussed, Ψ is the transform executed in the server to generate the response for a set of queries produced by Φ. In RDAS1 the response of the server is constructed just by running the queries specified by Φ on R_α and R_β. We denote the response by $S = (S_\alpha, S_\beta)$ where S_α

[2] By a query of disjunction on repeated attributes we mean a query like : `SELECT * FROM` R `WHERE` $a_1 = v_1$ `OR` $a_1 = v_2$ `OR` $a_2 = v_3$. Here the attribute a_1 is repeated twice.

and S_β corresponds to responses of q_α and q_β respectively. Thus, for the example, the server executes the queries Q1_α and Q1_β on R1_α and R1_β respectively and thus returns the response $S1 = (S1_\alpha, S1_\beta)$ which is shown in Table 3.

Relation $S1_\alpha$

EmpId	Name	Gender	Level	Nonce	Tag
TRW	Tom	M	L_2	1	Y_1
JOH	John	M	L_2	3	Y_3
RZT	Rosy	F	L_2	6	Y_6

Relation $S1_\beta$

Name	SearchKey	RowNo	Tag1
Gender	M	8	Y_8'
Level	L_2	10	${Y_{10}}'$

Table 3. Left side: Answer $S1_\alpha$, Right side: Answer $S1_\beta$

RDAS1.$\mathcal{V}$: The verification procedure receives as input the response $S = (S_\alpha, S_\beta)$ from the server, the original query and the keys. The response of the server consists of two parts. We denote these two parts as two sets S_α and S_β which are responses to the queries q_α and q_β respectively. Thus, S_α and S_β contains tuples from the relations R_α and R_β respectively.

The transformed queries q_α and q_β are also disjunctions of conditions, for a q_α query the conditions are of the form $a_i = v_i$, where a_i is an attribute and v_i its value, and for a q_β query the conditions are of the form Name $= v$ AND SearchKey $=$ w. Thus, for the description below, we consider that C_1^α OR C_2^α OR $\ldots$ C_l^α is a α query where each C_i^α is an equality condition and C_1^β OR C_2^β OR $\ldots$ C_l^β is a β query where each C_i^β is a conjunction of two equality conditions. Note that the number of conditions in q_α and q_β would always be the same. Let SaT be a predicate which takes as input a tuple t and a condition C (which can also be a query q) and outputs a 1 if the tuple t satisfies the condition C, otherwise outputs a zero. With these notations defined, we are ready to describe the verification algorithm. The verification algorithm consists of three procedures. We name the procedures as α-Verify, makeBitMap and β-Verify. The procedures are shown in Figure 2, and they are applied sequentially in the same order as stated above.

```
α-Verify
1.  for all tuples $t \in S_\alpha$
2.      if $\mathsf{SaT}(t, q_\alpha) = 0$, return $\perp$
3.      $ta \leftarrow \mathsf{MAC}_K(t[b_1]||\ldots||t[b_{|B|}]||t[\mathtt{Nonce}])$;
4.      if $ta \neq t[\mathtt{Tag}]$ , return $\perp$;
5.  end for

makeBitMap
6.  for $i \leftarrow 1$ to $l$
7.      $X_i \leftarrow 0^{\mathsf{nT}}$;
8.  end for
9.  for all tuples $t \in S_\alpha$
10.     for $i \leftarrow 1$ to $l$
11.         if $\mathsf{SaT}(t, C_i)$
12.             $j \leftarrow t[\mathtt{Nonce}]$;
13.             $\mathsf{bit}_j(X_i) \leftarrow 1$;
14.         end if
15.     end for
16. end for

β-Verify
17. for $i \leftarrow 1$ to $l$
18.     $T[i] \leftarrow 0$;
19. end for
20. for $i \leftarrow 1$ to $l$
21.     for all tuples $t \in S_\beta$
22.         if $\mathsf{SaT}(C_i^\beta, t) = 1$
23.             $T[i] \leftarrow T[i] + 1$;
24.             $LL \leftarrow t[\mathtt{Name}]||t[\mathtt{SearchKey}]||X_i||t[\mathtt{RowNo}]$;
25.             if $\mathsf{MAC}_K(LL) \neq t[\mathtt{Tag1}]$ return $\perp$;
26.         endif
27.     end for
28. end for
29. for $i \leftarrow 1$ to $l$
30.     if $T[i] \neq 1$ return $\perp$;
31. end for
32. return $\prod_{(b_1,b_2,\cdots,b_{|B|})} S_\alpha$;
```

Fig. 2. The procedures involved in the verification process: we assume that the verification procedure has as input the queries $q_\alpha = C_1^\alpha$ OR C_2^α OR $\ldots$ C_l^α, $q_\beta = C_1^\beta$ OR C_2^β OR $\ldots$ C_l^β, and the server responses S_α and S_β.

The verification procedure checks for both the correctness and the completeness of the server response against the original query q. Note that the server response consists of two distinct parts S_α and S_β, the S_α part corresponds to the real result of the original query q and the S_β part assists the verification process to verify the completeness of the result in S_α. In the part α-Verify, the verification procedure checks for the correctness of the tuples returned by the server. As in the transformed relation R_α a message authentication code is associated with each tuple of the original relation, hence the α-Verify part of the verification procedure checks whether the contents of the tuples in S_α are not modified. If any of the the tuples in S_α are modified then the computed message authentication code on the tuple will not match the attribute `Tag`. If the computed value of tag does not match with the attribute `Tag` for any tuple then the verification process rejects by returning $\perp$. Moreover in line 2 it checks whether each tuple in S_α does satisfy the specified query. If the verification process does not terminate in the α-Verify phase then it means that the tuples in S_α are all valid tuples of the relation R_α and they all satisfy the specified query q_α. The other two parts of the verification process checks the completeness of the response.

Corresponding to each condition $\texttt{Name} = v$ AND $\texttt{SearchKey} = \texttt{w}$ in the query q_β the procedure makeBitMap constructs the corresponding bitmap $\mathsf{BitMap}_{R_\alpha}(v, w)$ using the server response S_α. Note that if the server response S_α is correct then makeBitMap would be able to construct the bitmaps corresponding to each condition in q_β correctly. This is possible due to the specific type of the allowed queries. Recall that an allowed query is formed only by the disjunctions of equality conditions. In the procedure corresponding to the l conditions in q_β, l bitmaps are constructed which are named $X_1, \ldots, X_l$ (See the example later for more explanation).

In the procedure β-Verify the response S_β is verified using the bitmaps $X_1, \ldots, X_l$ constructed before. The procedure β-Verify first verifies whether S_β contains tuples corresponding to each condition in q_β, this is done using the counter $T[i]$, where i runs over the conditions in q_β. Notice, that for every condition C_i^β the server must return only one tuple in S_β. The other parts of the procedure involves in verifying the tags of the tuples against the tag's of the computed bitmaps.

To make the exposition clearer let us consider the same example we have so far considered, i.e., the relation $R1$ the queries $Q1_\alpha$, $Q1_\beta$ and the corresponding server responses of $S1_\alpha$ and $S1_\beta$ (which are shown in Table 3). Given these responses the procedure α-Verify will not terminate, as all the tuples in $S1_\alpha$ do satisfy the conditions in $Q1_\alpha$ and as they are correct responses in the sense that they are just copies of the tuples present in the relation R_α, hence the corresponding message authentication codes will match. Given the responses in $S1_\alpha$, one can compute the bitmaps $\mathsf{BitMap}_{R_\alpha}(\texttt{Gender}, M)$ and $\mathsf{BitMap}_{R_\alpha}(\texttt{Level}, L_2)$. To see this, see the response $S1_\alpha$ in Table 3, where it says that the tuples satisfying the condition $\texttt{Gender}{=}M$ `OR` $\texttt{Level}{=}L_2$ are the tuples with the nonce values 1, 3 and 6. Now, as the verification procedure has as input the whole of response $S1_\alpha$, hence it can predict correctly that the rows with the nonce value 1 and 3 satisfies the condition $\texttt{Gender}{=}M$ and all the tuples in $S1_\alpha$ (i.e., with nonce values 1, 3, 6) satisfies the condition $\texttt{Level}{=}L_2$. Thus, knowing that the total number of tuples in R_α to be 6, and assuming that server response is complete then the bitmap can be computed as $\mathsf{BitMap}_{R_\alpha}(\texttt{Gender}, M) = 101000$. Note that the 1st and 3rd bits of this bitmap are only one, as it corresponds to the response in $S1_\alpha$. Similarly one can compute $\mathsf{BitMap}_{R_\alpha}(\texttt{Level}, L_2) = 101001$. This is precisely what the procedure makeBitMaps would do for the example that we consider. The computation of the individual bitmaps $\mathsf{BitMap}_{R_\alpha}(\texttt{Gender}, M)$ and $\mathsf{BitMap}_{R_\alpha}(\texttt{Level}, L_2)$ are possible from $S1_\alpha$ as the $Q1_\alpha$ query is a disjunction of equality conditions, if in the contrary the query was a conjunction of conditions then there would be no way to compute the individual bitmaps in a straightforward way, this explains the reason for the query restriction that we impose.

Once these bitmaps are computed by using the procedure β-Verify one can verify the correctness of the response $S1_\beta$. As one can concatenate corresponding the bitmaps computed by the procedure makeBitMaps with the other attributes of the tuples in S_β and compute the tag using the message authentication code and thus verify if the computed tag matches the attribute Tag1.

The procedure β-Verify basically verifies the correctness of the response S_β, this verification is done by using the bitmaps constructed using the response S_α. The correctness of the response S_β implies the completeness of the response S_α. We discuss about this more in the following section.

4.1 Security of RDAS1

In this section we show that the security of RDAS1 can be reduced to the security of MAC. In other words, if there exists an adversary capable of breaking RDAS1 then the MAC is not secure.

We can distinguish two possibilities for breaking RDAS: infringe the correctness or violate the completeness of the response for a fixed query. To break the correctness the opponent must make changes in one or more tuples of S_α and still pass the verification process. This implies that the adversary must forge the respective MACs. On the other hand, to violate the completeness, the adversary must change the respective bitmaps in S_β which also implies forging the respective MACs. Now, we introduce this notion in a formal way.

Theorem 1. *Consider an adversary $\mathcal{A}$ attacking* RDAS1 *in the sense of definition 1. Let $\mathcal{A}$ choose a relation R with* nT *tuples and the relation be such that the transformed relation R_β contains n' tuples. Then there exists an adversary $\mathcal{B}$ attacking the message authentication code* MAC *such that*

$$\Pr[\mathsf{Succ}_{\mathcal{A}}] \leq \mathbf{Adv}^{\mathrm{auth}}_{\mathsf{MAC}}(\mathcal{B}).$$

Also, $\mathcal{B}$ asks at most $\mathsf{nT}+n'$ queries to its oracle and runs for time $t_{\mathcal{A}}+(\mathsf{nT}+n')(c+t_{\mathsf{MAC}})$, where $t_{\mathcal{A}}$ is the running time of $\mathcal{A}$, t_{MAC} is the time for one MAC computation and c is a constant.

Proof. The idea of the proof is to construct an adversary $\mathcal{B}$ whose task is to forge the message authentication code MAC. $\mathcal{B}$ will use the adversary $\mathcal{A}$ by acting as its challenger. The interaction between $\mathcal{B}$ and $\mathcal{A}$ is represented in the procedure below:

```
Adversary $\mathcal{B}^{\mathsf{MAC}_K(.)}$
1. Receive the relation $R$ from $\mathcal{A}$;
2. Send $(R_\alpha, R_\beta) \leftarrow \mathsf{RDAS1}.\mathcal{F}(R)$ to $\mathcal{A}$;
   (Use the oracle $\mathsf{MAC}_K(.)$ to construct $(R_\alpha, R_\beta)$;
   Let $Q$ be the set of queries asked to $\mathsf{MAC}_K(.)$)
3. Receive a query $q$ from $\mathcal{A}$;
4. Send $(q_\alpha, q_\beta) \leftarrow \mathsf{RDAS1}.\Phi(q)$ to $\mathcal{A}$
5. Receive a response $\rho'$ from $\mathcal{A}$
6. if $\mathsf{RDAS1}.\mathcal{V}(\rho') \neq \perp$ AND $\mathsf{RDAS1}.\mathcal{V}(\rho') \neq \mathsf{ans}(\mathcal{R}, \mathsf{q})$;
7.     $(x, \mathsf{tag}) \leftarrow$ GenerateForgery();
8. else $x \xleftarrow{\$} \{0,1\}^* - Q$, $\mathsf{tag} \xleftarrow{\$} \{0,1\}^\tau$;
9. return $x, \mathsf{tag}$
```

In line 7 we mention a routine **GenerateForgery**() which will be defined later. The heart of the reduction is the following claim:

Claim. If the condition of line 6 is satisfied, then $\mathcal{B}$ forges the MAC with probability 1.

If the above claim is valid then we have

$$\begin{aligned}
\Pr[\mathcal{B} \text{ forges}] &= \Pr[\mathcal{B} \text{ forges}|\mathsf{Succ}_{\mathcal{A}}] \Pr[\mathsf{Succ}_{\mathcal{A}}] + \Pr[\mathcal{B} \text{ forges}|\overline{\mathsf{Succ}_{\mathcal{A}}}] \Pr[\overline{\mathsf{Succ}_{\mathcal{A}}}] && (2)\\
&= \Pr[\mathsf{Succ}_{\mathcal{A}}] + \frac{1}{2^{\tau}} \Pr[\overline{\mathsf{Succ}_{\mathcal{A}}}] && (3)\\
&\geq \Pr[\mathsf{Succ}_{\mathcal{A}}],
\end{aligned}$$

as desired. Eq. (3) follows from (2) as according to our claim $\Pr[\mathcal{B} \text{ forges}|\mathsf{Succ}_{\mathcal{A}}] = 1$ and $\Pr[\mathcal{B} \text{ forges}|\overline{\mathsf{Succ}_{\mathcal{A}}}] = \frac{1}{2^{\tau}}$ as the probability that the τ bit tag of a message matches with a random τ bit string is $\frac{1}{2^{\tau}}$.

Proof of Claim: Here we will basically describe the procedure **GenerateForgery**(). Note that the response ρ' of $\mathcal{A}$ consists of $(S'_{\alpha}, S'_{\beta})$ and as the verification procedure does not output $\perp$ hence the following must be true:

1. The tuples in S'_{α} and S'_{β} satisfies the conditions in the queries q_{α} and q_{β} respectively, moreover there is only one tuple returned in S_{β} corresponding to each condition in q_{β} (see procedures α-Verify and β-Verify in Figure 2).
2. All the tuples in S'_{α} and S'_{β} are associated with their valid tags.

Moreover the condition in line 6 says that response produced by $\mathcal{V}$ is not correct. This can happen in the following two scenarios.

Case 1: There is a tuple $\mathtt{tup} \in S'_{\alpha}$ such that $\mathtt{tup}$ is not in R_{α}. Moreover, if $\{b_1, \ldots, b_{|B|}\}$ be the original set of attributes, and $X = \mathtt{tup}[b_1]|| \ldots ||\mathtt{tup}[b_{|B|}]||\mathtt{tup}[\mathtt{Nonce}]$, then $(X, \mathtt{tup}[\mathtt{Tag}])$ is a valid message tag pair. Note, that this case signifies that the server response is incorrect.

Case 2: There is no $\mathtt{tup} \in S'_{\alpha}$ which is not present in R_{α}. This case can only occur if the response is incomplete. This signifies that $\mathcal{A}$ has been able to forge a tag in R_{β}. To see this, note that the procedure $\mathcal{V}$ constructs the bitmaps corresponding to the conditions in the q_{β} query based on the information in S_{α}. If the result returned in S_{α} is incomplete then the bitmap corresponding to some condition would be wrongly computed by $\mathcal{V}$, but this wrong bitmap corresponds to the tag returned in S'_{β}.

In both the above cases $\mathcal{B}$ can construct a forgery for the MAC in the following way:

GenerateForgery() for Case I: $(X, \mathtt{tup}[\mathtt{Tag}])$ is a valid forgery, as according to the condition in line 6, $\mathtt{tup}[\mathtt{Tag}])$ is a valid tag for X, moreover as $\mathtt{tup}$ is not in R_{α} hence $\mathcal{B}$ has never asked its oracle a query of X.

GenerateForgery() for Case II: Consider an arbitrary attribute a, and its value v, which is related to the query in question, such that the bitmap computed by $\mathcal{V}$ for the condition $a = v$ is Y' and $Y' \neq Y$, where $Y = \mathsf{BitMap}_v(a)$ is the real bitmap. In this case S'_{β} would contain a tuple (a, v, r, tag) where r is the row number and it must be the case that $\mathsf{MAC}_K(a||v||Y'||r) = tag$. Thus $(a||v||Y'||r, tag)$ is a valid forgery, as $\mathcal{B}$ has never asked $a||v||Y'||r$ to its MAC oracle.

$\square$

4.2 Costs and Overheads

Storage cost: Given a relation $R(B)$ with nT tuples, let $\mathsf{size}(t_i[b])$ denote the size of the attribute b in the tuple t. Then the total size of R (which we also denote by $\mathsf{size}(R)$) would be given by

$$\mathsf{size}(R) = \sum_{i=1}^{\mathsf{nT}} \sum_{b \in B} \mathsf{size}(t_i[b]).$$

If this relation R is converted into (R_α, R_β) with the help of the authentication transform $\mathsf{RDAS1}.F$, then we would have,

$$\mathsf{size}(R_\alpha) = \mathsf{size}(R) + \sum_{i=1}^{\mathsf{nT}} (\mathsf{size}(t_i[\mathtt{Nonce}]) + \mathsf{size}(t_i[\mathtt{Tag}])),$$

if we assume a tag of constant length of τ bits then we would have

$$\mathsf{size}(R_\alpha) \leq \mathsf{size}(R) + \mathsf{nT}(\lg \mathsf{nT} + \tau).$$

Again considering the set of allowed attributes of R as $A = \{a_1, a_2, \ldots, a_m\}$, and $N = \sum_{i=1}^{m} \mathsf{Card}(a_i)$, we will have

$$\mathsf{size}(R_\beta) = \sum_{i=1}^{N} (\mathsf{size}(t_i[\mathtt{Name}]) + t_i[\mathtt{SearchKey}] + t_i[\mathtt{RowNo}] + \mathsf{size}(t_i[\mathtt{Tag1}])).$$

If we consider s_{Name} and s_{sk} the maximum size of the values of the attributes `Name` and `SearchKey`, then we would have

$$\mathsf{size}(R_\beta) \leq N(s_{\mathsf{Name}} + s_{\mathsf{sk}} + \lg(\mathsf{nT} + N) + \tau).$$

The total cost of storage at the server side would be $\mathsf{size}(R_\alpha) + \mathsf{size}(R_\beta)$, and at the client side would be $\lg(\mathsf{nT})$ as in the client we need to store the number of tuples in the original relation.

Communication Cost: Consider the query `SELECT * FROM` R_α `WHERE` $a_1 = v_1$ `OR` $a_2 = v_2$ `OR ...... OR` $a_l = v_l$, let the number of tuples satisfying the query be num. Let siz be the size of the response in a normal scenario without authentication. Then the maximum size of the server response in case of $\mathsf{RDAS1}$ would be

$$\mathsf{siz}_{\mathsf{RD1}} = \mathsf{siz} + \mathsf{num} \times (\lg \mathsf{nT} + \tau) + l \times (s_{\mathsf{Name}} + s_{\mathsf{sk}} + \lg(\mathsf{nT} + N) + \tau), \qquad (4)$$

where the first two terms corresponds to the S_α response and the remaining term counts for the S_β response.

5 RDAS2: Selects Involving Arbitrary Boolean Connectives

RDAS1 can be modified to support SELECT queries involving all kinds of Boolean connectives at the cost of the size of the query responses. Recall that the query restriction for RDAS1 arises from the problem of constructing the bitmaps of all the attributes involved in the query. We propose an extension of RDAS1 which can support queries of the form

Q: `SELECT * FROM` R `WHERE` $(a_1 = v_1)$ Δ_1 $(a_2 = v_2)$ Δ_2 ...... Δ_{l-1} $(a_l = v_l)$,
where Δ_is are arbitrary Boolean connectives. An easy solution to this case would be to change RDAS1 to a new protocol RDAS2 along the following lines:

1. The relation R_β produced by $\mathsf{RDAS2}.\mathcal{F}$ would contain explicit bitmaps corresponding to the attributes and the values. Specifically, the attributes present in R_β should be $\{\texttt{Name}, \texttt{SearchKey}, \texttt{RowNo}, \texttt{bitmap}, \texttt{tag1}\}$. Thus, for creating the relation R_β we need to add a line $t_j^{R_\beta}[\texttt{bitmap}] \leftarrow \mathsf{BitMap}_R(\mathcal{L}_j^1, \mathcal{L}_j^2)$ after line 5 in the procedure **Creating** R_β in Fig. 1.
2. The query translation procedure and the response procedure for RDAS2 remains same as that of RDAS1.
3. The response procedure also remains the same, i.e., the server just answers the q_α and q_β queries, but as the R_β relation now explicitly contains the bitmaps, hence the bitmaps would also be a part of the query.
4. For the verification procedure in RDAS2 it is not required to create the bitmaps any more, the client verifies the S_α response by the procedure α-Verify in Fig. 2, then it verifies the tags of the individual bitmaps returned in S_β and finally computes the result bitmap using the returned bitmap and checks if the result bitmap matches with the result returned.

We now state the storage and communication costs for RDAS2 following the notations in Section 4.2. The size of R_α in case of RDAS2 would be the same as in RDAS1, the size of R_β would be

$$\mathsf{size}(R_\beta) \leq N(s_{\mathsf{Name}} + s_{\mathsf{sk}} + \lg(\mathsf{nT} + N) + \tau + \mathsf{nT}).$$

The size of a server response in case of RDAS2 would be

$$\mathsf{siz}_{\mathsf{RD2}} = \mathsf{siz}_{\mathsf{RD1}} + l \times \mathsf{nT} \tag{5}$$

where $\mathsf{siz}_{\mathsf{RD1}}$ is the size of the response of RDAS1, as given in Eq. (4).

In case of RDAS2, though we state that the bitmaps are to be explicitly stored in the relation R_β, but as most commercial databases uses bitmaps indices for accelerating query processing, hence this may not amount to extra storage in some systems. Moreover bitmaps can be compressed, there has been substantial work on suitable encoding of bitmaps such that their sizes can be reduced and the Boolean operations be applied on the compressed bitmaps [3, 4, 31, 34, 13]. Applying proper encoding of the bitmaps can drastically reduce both storage and communication costs, we elaborate on this in Section 7.

5.1 Security of RDAS2

In this section we show that as in RDAS1 the security of RDAS2 depends on the strenght of the MAC. Notice that the pairs m, tag in S_α, S_β are the same that in RDAS1. The only difference is that in RDAS2, the complete message $t[Name]||t[SearchKey]||t[Bitmap]||t[RowNo]$ is stored in S_β. Thus, the security theorem for RDAS1 apply to RDAS2 without any change.

Theorem 2. *Consider an adversary $\mathcal{A}$ attacking* RDAS2 *in the sense of definition 1. Let $\mathcal{A}$ choose a relation R with* nT *tuples and the relation be such that the transformed relation R_β contains n' tuples. Then there exists an adversary $\mathcal{B}$ attacking the message authentication code* MAC *such that*

$$\Pr[\mathsf{Succ}_{\mathcal{A}}] \leq \Pr[\mathcal{B} \textit{ forges }].$$

Also, $\mathcal{B}$ asks at most $\mathsf{nT}+n'$ queries to its oracle and runs for time $t_{\mathcal{A}}+(\mathsf{nT}+n')(c+t_{\mathsf{MAC}})$, where $t_{\mathcal{A}}$ is the running time of $\mathcal{A}$, t_{MAC} is the time for one MAC computation and c is a constant.

The proof follows in the same way as the proof or Theorem 1.

6 Using Aggregated MACs

Deterministic message authentication codes can be suitably aggregated [10]. The main motivation behind aggregated MACs is to provide authenticated communications in bandwidth constrained scenarios, say in case of a sensor network. In case of a sensor network various sensor nodes have their own secret key which they only share with the base station. Let us see an example of a sensor network consisting of k nodes labeled $i_1, \ldots, i_k$. We assume a communication protocol, where first the node i_1 sends it authenticated data (m_{i_1}, t_{i_1}) (consisting of the message m_{i_1} and the authentication tag t_{i_1}, which is computed using m_{i_1} and the key k_{i_1} of the node) to the next node i_2. Node i_2 aggregates its own authenticated message (m_{i_2}, t_{i_2}) to the received message and sends $[(m_{i_1}, t_{i_1}), (m_{i_2}, t_{i_2})]$ to i_3. Thus, ultimately the base station receives $[(m_{i_1}, t_{i_1}), \ldots, (m_{i_k}, t_{i_k})]$ from node i_k. As the base station owns the key of all the nodes, hence it can verify the authenticity of all the messages received. If we assume that the authentication tags are n bit long irrespective of the message lengths, then this communication protocol has an extra bandwidth overhead of $O(k^2 n)$ bits. In case of sensor networks the data send by them are generally very short, say a temperature reading etc, thus the size of the data may be much smaller than the size of the authentication tag (if one uses a secure MAC to generate the tag, then n would be around 128 bits). Thus a $O(k^2 n)$ bit extra bandwidth overhead may not be tolerated in a standard scenario. Aggregated MAC comes as a solution to this problem, if the MAC used for generating the authenticated tag is "aggregate-able", then node i_1 sends (m_{i_1}, t_{i_1}) to i_2, i_2 sends $[(m_{i_1}, m_{i_2}), t_{i_1} \oplus t_{i_2})]$ etc. And finally, node i_k sends $[(m_{i_1}, \ldots, m_{i_k}); t_{i_1} \oplus \ldots \oplus t_{i_k})]$ to the base station. This protocol only requires an extra bandwidth overhead of $O(kn)$.

In the example above, we used $\oplus$ as the aggregation operator, but one can define it generically as was done in [10]. Moreover, it was proved in [10] that if one have a secure deterministic message authentication code then by using $\oplus$ as the aggregation operator one can securely aggregate MACs, in the multiuser setting. The security model describes a successful adversary as one who can produce a set of messages along with the user identifiers for each message and an aggregated tag which verifies. The adversary is allowed to see the tags corresponding to messages of his/her choice, and additionally (s)he is allowed to corrupt some users and know their keys. The final message set produced by the adversary should contain at least one message user id pair (m, id) such that (s)he has not seen the tag corresponding to m and has not corrupted the user id.

Aggregated MACs can reduce communication costs both in case of RDAS1 and RDAS2. Functionally, using aggregated MACs would reduce the response sizes, as then the tags corresponding to all the tuples in S_α and S_β will not be required to be sent, but an aggregation of these would be required. But it is to be noted that our case is fundamentally different from the above example in that we do not have multiple users, the authentication tags are all generated using the same key. This has an advantage, that the security reduction that can be obtained in this scenario is tighter than that obtained in the multiuser scenario. We discuss this a bit more in Section 6.1.

We propose two variants of RDAS namely RDAS1-agg and RDAS2-agg which extends RDAS1 and RDAS2 by using the functionality of aggregated MACs. We describe these variants next.

To convert RDAS1 to RDAS1-agg we just need to apply little modifications in the server response procedure and the verification procedure. In the response procedure, the server sends S_α and S_β as in RDAS1 but without the attributes Tag and Tag1, and it sends two additional strings str_α and str_β which contains the xor of the tags in S_α ad S_β respectively. For example, consider the responses $S1_\alpha$ and $S1_\beta$ in Table 3, the response for RDAS1-agg would be projections of $S1_\alpha$ and $S1_\beta$ without the attributes Tag and Tag1 respectively. And additionally the response procedure would return $\mathsf{str}_\alpha = Y_1 \oplus Y_3 \oplus Y_6$ and $\mathsf{str}_\beta = Y'_8 \oplus Y'_{10}$. The changes that are to be applied to RDAS2 to obtain RDAS2-agg are exactly the same.

This aggregation of tags leads to a savings in the communication cost in both RDAS1 and RDAS2. Following the notation of Section 4.2, the size of server response for RDAS1-agg and RDAS2-agg would be

$$\mathsf{siz}_{\mathsf{RD1\text{-}agg}} = \mathsf{siz}_{\mathsf{RD1}} - \tau(\mathsf{num} + l) \tag{6}$$

$$\mathsf{siz}_{\mathsf{RD2\text{-}agg}} = \mathsf{siz}_{\mathsf{RD2}} - \tau(\mathsf{num} + l). \tag{7}$$

6.1 Security of RDAS1-agg and RDAS2-agg

For discussion of security, first we try to formalize the security definition of the aggregated MACs. Firstly, given a MAC $\mathsf{MAC} : \mathcal{K} \times \mathcal{M} \to \{0,1\}^\tau$ we define the corresponding aggregated MAC MAC^* in the single user setting as follows. The MAC generation algorithm given as input $m \in \mathcal{M}$ and a key $K \in \mathcal{K}$ outputs $\mathsf{MAC}_K(m)$. The aggregation algorithm when given two sets of messages $M_1, M_2 \subset \mathcal{M}$ and two corresponding tags $\mathsf{tag}_1, \mathsf{tag}_2 \in \{0,1\}^\tau$ outputs $\mathsf{tag}_1 \oplus \mathsf{tag}_2$. The verification algorithm when given a key $K \in \mathcal{K}$, a set of messages $M \subset \mathcal{M}$ and a tag $\mathsf{tag} \in \{0,1\}^\tau$ computes

$$t = \bigoplus_{m \in M} \mathsf{MAC}_K(m),$$

and accepts if $\mathsf{tag} = t$ and rejects otherwise.

An adversary $\mathcal{A}$ attacking MAC^* is given oracle access to $\mathsf{MAC}_K()$, i.e., it can know the tags corresponding to the messages of it's choice. Let us consider that $\mathcal{A}$ asks q queries $Q = \{x_1, \ldots, x_q\}$ to its oracle and gets back the corresponding tags $t_1, \ldots, t_q$. Finally $\mathcal{A}$ produces a set of messages $M_f \subset \mathcal{M}$ and a tag tag_f. $\mathcal{A}$ is said to be successful in forging if there is at least one element $m \in M_f$ such that $m \notin Q$ and the verification algorithm of MAC^* on input (K, M_f, tag_f) accepts. We define the forging advantage of $\mathcal{A}$ as

$$\mathbf{Adv}^{\text{auth-ag}}_{\mathsf{MAC}^*}(\mathcal{A}) = \Pr[\mathcal{A} \text{ forges}].$$

If $\mathsf{MAC}_K()$ is a secure MAC then so is $\mathsf{MAC}^*_K()$. The following theorem states this more formally.

Theorem 3. *Let $\mathcal{A}$ be an arbitrary adversary attacking the aggregated MAC MAC^*, and $\mathcal{A}$ runs for time T and makes q queries to its oracle. Then there exists an adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}^{\text{auth}-\text{ag}}_{\mathsf{MAC}^*}(\mathcal{A}) = \mathbf{Adv}^{\text{auth}}_{\mathsf{MAC}}(\mathcal{B}).$$

$\mathcal{B}$ in turn makes $\mathcal{O}(q)$ queries and runs for time $\mathcal{O}(T)$.

Proof. Note $\mathcal{B}$ is attacking MAC_K, hence it has MAC_K as its oracle. $\mathcal{B}$ runs $\mathcal{A}$ as follows: whenever $\mathcal{A}$ asks a query m, $\mathcal{B}$ returns to $\mathcal{A}$, $\mathsf{MAC}_K(m)$ through its oracle. Let Q be the set of queries made by $\mathcal{A}$. Finally $\mathcal{A}$ outputs a forgery (M_f, tag_f). If $\mathcal{A}$ is successful in forging then $\Gamma = M_f \setminus Q \neq \emptyset$. Fix $r \in \Gamma$ and compute

$$t_{all} = \begin{cases} \bigoplus_{x \in \Gamma \setminus \{r\}} \mathsf{MAC}_K(x), & \text{if } \Gamma \setminus \{r\} \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\mathcal{B}$ can compute t_{all} using its oracle. Finally, $\mathcal{B}$ outputs $(r, \mathsf{tag}_f \oplus t_{all})$ as its forgery. It is easy to see that if $\mathcal{A}$ successfully forges then so does $\mathcal{B}$, and the runing time of $\mathcal{B}$ and the number of queries asked by $\mathcal{B}$ are as desired. □

With this discussion we are ready to state the security of RDAS1-agg and RDAS2-agg.

Theorem 4. *Consider an adversary $\mathcal{A}$ attacking $\Upsilon \in \{\mathsf{RDAS1\text{-}agg}, \mathsf{RDAS2\text{-}agg}\}$ in the sense of definition 1. Let $\mathcal{A}$ choose a relation R with nT tuples and let R_β contain n' tuples. Then there exists an adversary $\mathcal{B}$ such that*

$$\Pr[\mathsf{Succ}_{\mathcal{A}}] \leq \mathbf{Adv}^{\mathrm{auth}}_{\mathsf{MAC}}(\mathcal{B}).$$

Also, B asks at most $\mathsf{nT}+n'$ queries to its oracle and runs for time $t_{\mathcal{A}}+(\mathsf{nT}+n')(c+t_{\mathsf{MAC}})$, where $t_{\mathcal{A}}$ is the running time of $\mathcal{A}$, t_{MAC} is the time for one MAC computation and c is a constant.

Proof. The proof is similar to proof of Theorem 1, with some small differences. We present the reduction in two steps. First we construct an adversary $\mathcal{C}$ which attacks the aggregated MAC MAC^*, and it runs the adversary $\mathcal{A}$, such that

$$\Pr[\mathsf{Succ}_{\mathcal{A}}] \leq \mathbf{Adv}^{\mathrm{auth-ag}}_{\mathsf{MAC}^*}(\mathcal{C}). \tag{8}$$

Then using Theorem 3, we have that there exists an adversary $\mathcal{B}$ such that

$$\mathbf{Adv}^{\mathrm{auth-ag}}_{\mathsf{MAC}^*}(\mathcal{B}) = \mathbf{Adv}^{\mathrm{auth}}_{\mathsf{MAC}}(\mathcal{C}). \tag{9}$$

Thus, from equations (8) and (9) the Theorem follows.

To prove equation (9) we construct an adversary $\mathcal{C}$ which attacks the aggregate MAC MAC^*. $\mathcal{C}$ runs $\mathcal{A}$ ($\mathcal{A}$ attacks Υ) as follows.

```
Adversary $\mathcal{C}^{\mathsf{MAC}_K(.)}$
1. Receive the relation $R$ from $\mathcal{A}$;
2. Send $(R_\alpha, R_\beta) \leftarrow \Upsilon.\mathcal{F}(R)$ to $\mathcal{A}$;
   (Use the oracle $\mathsf{MAC}_K(.)$ to construct $(R_\alpha, R_\beta)$;
   Let $Q$ be the set of queries asked to $\mathsf{MAC}_K(.)$)
3. Receive a query $q$ from $\mathcal{A}$;
4. Send $(q_\alpha, q_\beta) \leftarrow \Upsilon.\Phi(q)$ to $\mathcal{A}$
5. Receive a response $\rho'$ from $\mathcal{A}$
6. if $\Upsilon.\mathcal{V}(\rho') \neq \perp$ AND $\Upsilon.\mathcal{V}(\rho') \neq \mathsf{ans}(\mathcal{R}, \mathsf{q})$;
7.     $(x, \mathsf{tag}) \leftarrow$ AggregateForgery();
8. else $x \xleftarrow{\$} \{0,1\}^* - Q$, $\mathsf{tag} \xleftarrow{\$} \{0,1\}^\tau$;
9. return $\{x\}, \mathsf{tag}$
```

Description of $\mathcal{C}$ is almost the same as the description of adversary $\mathcal{B}$ as given in the proof of Theorem 1. As in the proof of Theorem 1, we claim that if the condition in line 6 is satisfied then $\mathcal{C}$ generates a forgery for MAC^* with probability 1. We can see this by following the same line of arguments as in the proof of Theorem 1. We explain the procedure of **AggregateForgery**() briefly for Υ =RADS2-agg, the case of Υ =RADS1-agg is similar.

Let $B = \{b_1, b_2, \ldots, b_{|B|}\}$ be the original set of attributes and let $\rho' = (S'_\alpha, S'_\beta), \mathsf{str}_\alpha, \mathsf{str}_\beta$. Note that here S'_α, S'_β does not contain the attribute `Tag` and `Tag1` respectively. If the condition in line 6 is satisfied then either of the two cases must be satisfied:

- *Case I*: There exists at least one tuple X in S_α which does not correspond to any tuple in R_α. Construct a set $\mathbb{S}_\alpha$ as

$$\mathbb{S}_\alpha = \{t[b_1]|| \cdots ||t[b_{|B|}]||t[\texttt{Nonce}] : t \in S'_\alpha\}.$$

  Then $\mathbb{S}_\alpha, \mathsf{str}_\alpha$ constitutes a valid forgery for MAC^*.

– *Case II*: There exists at least one tuple X in S_β which does not correspond to any tuple in R_β. Construct a set $\mathbb{S}_\beta$ as

$$\mathbb{S}_\beta = \{t[\texttt{Name}]||t[\texttt{SearchKey}]||t[\texttt{bitmap}]||t[\texttt{RowNumber}] : t \in S'_\beta\}.$$

Then $\mathbb{S}_\beta, \mathsf{str}_\beta$ constitutes a valid forgery for MAC^*. □

7 Bitmap Compression

There had been a lot of work that has shown that bitmap based indexing works effectively in database applications. To further improve their effectiveness, compression schemes have been developed, these schemes are capable of reducing the index size without increasing query processing time. The most frequent operations over bitmaps are bitwise logical operations [11], the specific compression schemes developed for bitmaps also allows logical operations to be performed on the compressed bitmaps.

RDAS2 and its variants require explicit storage of bitmaps and also the bitmaps need to be transmitted as the part of the query response, hence bitmap compression in this context can be very helpful. Our experiments that we present later also validates that using compressed bitmaps not only reduces storage and communication costs but also results in considerable savings in computation time.

Most of the compression schemes applied to bitmaps are based on run length encoding (RLE) scheme. Basic RLE works on the basis of the following simple idea. Consecutive occurrences of identical bits are detected in a bit string, such occurrences are known as a *fill*. Each fill in a bit string can be recorded with a counter representing its length and one bit indicating the actual value. This compression scheme is lossless and very efficient.

Several variants of RLE has been developed for application to bitmap compression. In our implementations we use a specific scheme called Enhanced Word Aligned Hybrid (EWAH). EWAH was studied independently by Wu et al. [11] and Lemire et al. [13]. We describe the basic scheme with an example.

EWAH divides the whole bit string X into 32 bit blocks and classify each block as either a *clean word* or a *dirty word*. A clean word is a 32 bit fill (either of zeros or ones), a word which is not *clean* is called *dirty*. The basic idea is to encode in such a manner that the clean words are compressed by specifying the type of fill contained in the word and its length; and the dirty words occur verbatim in the encoded string. We explain the basic encoding procedure with an example in Figure 3. In the example, the original bit string is shown in the beginning followed by the compressed string. The original string is 224 bits long and is represented in hexadecimal. As we can see that in the input string the first two blocks are dirty words followed by four clean words each with a fill of zeros and the last word is a dirty word. The encoded string contains two types of 32 bit words namely *marked words* and *verbatim words*. Marked words are sort of headers which carries information regarding the length and positions of fills and verbatim words are verbatim copies of dirty words. The first bit of each marked word represents the type of clean word that is to follow, the next 16 bits of a marked word encodes the length (in words) of the clean words and the final 15 bits encodes the number of dirty words that follows the clean words.

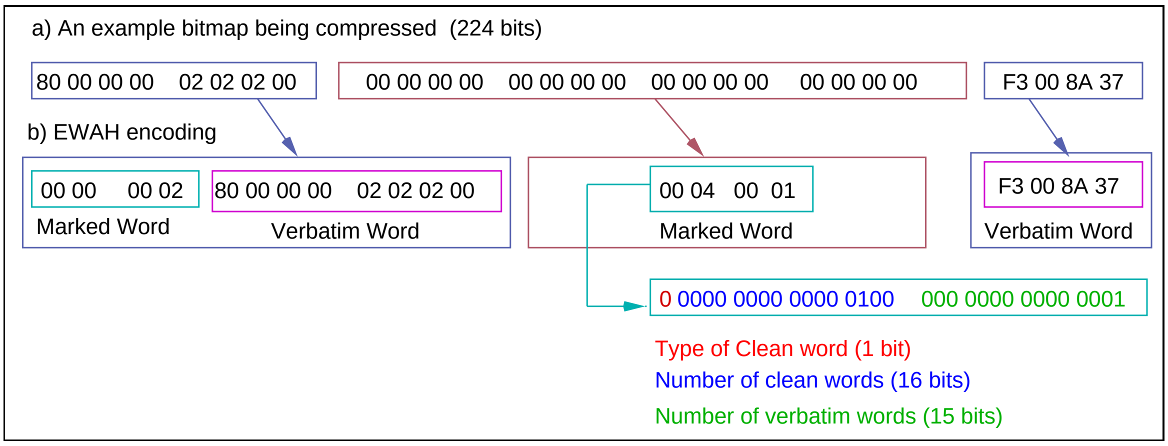


Fig. 3. EWAH compression example.

In our example the first word of the encoded string is a marked word (this is so, for all strings) `0x00 00 00 02`, which means that the marked word is followed by no clean words but two dirty words. Next the following two dirty words are copied verbatim. Next the marked word `0x00040001` occurs, which means that this marked word would be followed by four clean words each of zero which would be followed by one dirty word. At the end of this marked word the dirty word is written verbatim. This procedure can be easily generalized.

Also, it is not difficult to see that as the EWAH works with a granularity of 32 bits, hence bitwise operations can be easily performed on the compressed words and the compressed results can be directly obtained. In [13] one can find the algorithms to perform logical operations on RLE encoded bitmaps. These procedures can be easily extended for EWAH compression scheme.

7.1 RDAS2-cmp: RDAS2 with Compression

The changes required in RDAS2 to incorporate bitmap compression are as follows:

1. The column R_β[`bitmap`] in the relation R_β would be populated by the EWAH compressed bitmaps. The MAC would also be computed using the compressed bitmaps.
2. The query translation procedure and the response procedure remains same as that of RDAS2 (without compression).
3. The response procedure also remains the same, i.e., the server just answers the q_α and q_β queries.
4. For the verification procedure in RDAS2 with compression it is not required to uncompress the bitmaps, the client verifies the S_α response by the procedure α-Verify in Fig. 2, then it verifies the tags of the individual compressed bitmaps returned in S_β and finally computes the result bitmap using the returned compressed bitmaps, and checks if the result bitmap matches with the compressed bitmap obtained from the response.

The incorporation of compression in RDAS2 has no effect on the security but the storage and communication costs would decrease based on the amount of compression achieved.

The size of R_α in case of RDAS2 with compression would be the same as in RDAS1. Let $\mathsf{sz\_cmp}(t_N[bitmap])$ denote the size of the compressed bitmap for a given attribute and value pair, the size of R_β would be

$$\mathsf{size}(R_\beta) \leq N(s_{\mathsf{Name}} + s_{\mathsf{sk}} + \lg(\mathsf{nT} + N) + \tau + \mathsf{sz\_cmp}(t_N[bitmap]))$$

The size of a server response in case of $\mathsf{RDAS2}$ would be

$$\mathsf{siz}_{\mathsf{RD2}} = \mathsf{siz}_{\mathsf{RD1}} + l \times \mathsf{sz\_cmp}(t_l[bitmap]), \tag{10}$$

where $\mathsf{siz}_{\mathsf{RD1}}$ is the size of the response of $\mathsf{RDAS1}$, as given in Eq. (4).

Our experiments (presented later) clearly shows gains in both communication and computational cost if compression is used.

8 Further Extensions and Improvements

In this section we discuss several other ways in which the RDAS framework can be extended.

8.1 The case of empty replies

One limitation of $\mathsf{RDAS1}$ and $\mathsf{RDAS2}$ is that if the server replies with an empty response in S_β corresponding to a query then there is no way to verify the validity of the response. For example, in case of the relation $R1$, if a query

```
SELECT * FROM R1 WHERE Level = 'L3'
```

is posed, then response to this query would be empty, and there is no way to verify whether the response is correct. It is to be noted that a server may reply an $\mathsf{RDAS2}$ query which has a valid response with an empty response and in such a scenario also $\mathsf{RDAS1}$ provides no mechanism to detect that the response is wrong.

If we assume knowledge of the client regarding the values of the attributes present in the relation then it would be easy for it to verify whether an empty response is the correct one. We discuss two simple ways by which $\mathsf{RDAS1}$ can be extended so that empty responses can be verified. The same steps are valid for $\mathsf{RDAS2}$ also.

1. Recall that the authentication transform $\mathcal{F}$ returns two additional data items M_c and M_s, which are stored in the client and server sides respectively. In $\mathsf{RDAS1}$, $M_c = \mathsf{nT}$ the number of tuples in the original relation, and M_s is empty. As a first solution, we suggest to augment M_c with a list of all values corresponding to each sensitive attribute in the relation. With this additional information, the client can easily verify whether an empty response is correct. This functionality comes with an additional storage cost in the client side. But if we assume that the domain of the attributes have small cardinality this storage would be much less compared to the size of the whole relation. Moreover, with this small change in $\mathsf{RDAS1}$ the other efficiency and security claims remains unchanged.
2. As a second solution we propose to augment M_s with additional information. In particular, we suggest to build a list Lst_a, which contains all distinct values of the attribute a which occurs in the original relation (encoded in an appropriate way). Corresponding to each such list we compute $\mathsf{lstTag}_a = \mathsf{MAC}_{K_1}(\mathsf{Lst}_a)$, and store $(\mathsf{Lst}_a, \mathsf{lstTag}_a)$ for each sensitive attribute a in M_s. Whenever the server returns an empty response corresponding to a query involving a set of attributes S, then the server includes in its response $(\mathsf{Lst}_a, \mathsf{lstTag}_a)$ for all $a \in S$. It is easy to see that with this information the client can verify whether the empty response is the correct one.
   This solution has no extra overhead on the client side storage, but increases the server side storage, which is generally not of much concern. Additionally, it increases the bandwidth requirement for the responses of queries with empty responses. But, for other queries the functionality of $\mathsf{RDAS1}$ along with the security claims remain unchanged.

8.2 Including other query types

Efficient range query processing: Range queries can be handled by RDAS2 by posing a range query with several selects. But this may not be efficient. An efficient way to handle range queries would be to store range bitmaps [3, 4]. The type of bitmaps that we have used so far are called equality bitmaps, and they are good for select queries. Using range bitmaps one can encode range information of the attribute values and thus would be well suited for processing range queries. Adding other bitmap encodings in our system can be done in a straightforward way, and would increase the functionality at the cost of storage

Projection and aggregation queries: Projection and aggregation queries cannot be handled directly by our protocol. But, projections and aggregations can always be done in the client side if these functionalities are required. And this can also be added to the system, without hampering its security properties.

Join queries: We have not discussed the functionality of join processing for our schemes. But basic join processing (like equijoins) can again be obtained in a straightforward way by storing extra information. In particular, join bitmaps [24, 1] can be used for this purpose.

8.3 Multiuser settings

As we have stated earlier, as we use a symmetric key primitive as the main cryptographic object, thus our scheme is restricted to a single user setting, i.e., the data owner is the one who queries the database. We can replace the message authentication scheme with a public key signature scheme to extend the scheme for multiple queriers. We see no problem in doing this, but for a multiuser setting, the security definition that we give for RDAS would no more be valid, and one needs to extend this definition. Using signatures, the computational cost would in general be more than that in using MACs. The functionality of aggregation can also be used (as we used for MACs) to reduce communication costs.

8.4 Dynamic databases

We argued that there exist scenarios in which it may not be necessary to handle data updates, for instance, the case of data warehousing applications. Thus having a scheme which is valid only for static databases is useful. Most work in the literature on authenticated query processing focus on static databases. The proposals for dynamic scenario are quite few [14, 29, 37]. However, our proposal can be extended to dynamic scenarios also; this extension is not straightforward and we would report this separately soon.

9 Experimental Results

In this section we discuss the experimental results and compare the performance of RDAS1 and RDAS2 in various scenarios.

9.1 The Basic Building Blocks

Both RDAS1 and RDAS2 can be implemented with any secure MAC, we chose two MACs for our implementations (a) PMAC instantiated with an AES with 128 bit key (in particular we use the description in [30]) and (b) Polynomial evaluation MAC (which we will further call as PolyMac).

PMAC is a block cipher based MAC where the main operations involved are block cipher calls. The way we implement PolyMac is as follows. Let $X_1||X_2||\dots||X_m = \mathsf{parse}_n(X)$, and let $\mu \in \{0,1\}^n$, we define

$$\mathsf{PolyMac}_{h,k}(X,\mu) = (X_1 h \oplus X_2 h^2 \oplus \mathsf{cpad}_n(X_m)h^m \oplus |X|h^{m+1}) \oplus E_k(\mu),$$

where X is the message and μ a non-repeating quantity associated with each message, $E_k()$ is a block cipher and the additions and multiplications are in the field $\mathbb{F}_{2^n}$. Such polynomial evaluation MACs are known to be secure when the quantity μ is non-repeating. For our implementations we take $n = 128$, and we consider the attribute `Nonce` in R_α and `RowNo` in R_β as the quantity μ. For PolyMac also we choose the block cipher as AES with 128 bit key.

One thing to notice is that PolyMac is not a deterministic MAC. It is a statefull MAC, the quantity μ is a state of the algorithm and repetition of μ completely breaks down its security. As we stated in Section 6.1 only deterministic MACs can be aggregated, thus, Theorem 3 for aggregated MACs does not hold for PolyMac. Thus PolyMac cannot be used in RDAS1-agg and RDAS2-agg.

For implementation of the block cipher in both MACs we use the new Intel dedicated instructions for AES. Finite field multiplications required for the PolyMac were implemented using the `PCLMULQDQ` instruction, which can perform carry-less multiplication of two 64 bit strings. These 64 bit multiplications were combined using the Karatsuba technique to obtain multiplication of two 128 bit strings, the final reduction was performed using a technique described in [8].

9.2 Experimental Settings

All results were obtained by testing the implementations in a machine with the following specifications:

- **CPU:** Four-core i5-2400 Intel processor (3.1GHz).
- **OS:** Ubuntu 12.04.02 LTS.
- **DataBase:** PostgreSQL 9.1.9
- **Compiler:** gcc 4.7.3

We use Census-Income data set [6] to test performance of our schemes. This data contains weighted census data extracted from the 1994 and 1995 population surveys conducted by the U.S. Census Bureau. The number of instances in the data set is 199523. The data contains 42 demographic and employment related variables, the sum of the cardinalities of all the attributes is 103419, and the total size of the dataset is 99.1 MB.

As explained, RDAS1 and RDAS2 work in an environment where one needs to perform computations in both the client and the server side. In our implementation all server-side computations are done in the PostgreSQL database using the PostgreSQL tools. We implemented the client in C, where ever possible we used the Intel SIMD instructions using Intel intrinsics. We designed the server side code in such a way such that all computations can be handled by the default PostgreSQL tools. This specific implementation choice makes our client much more powerful than the server, and also leaves space for a much more optimized implementation. Such an optimized implementation would require an implementation of all database engine functionalities, which we think is beyond the scope of this work. But, we would like to mention that this specific design choice also gives us the opportunity to see how good one can do by adding the authentication functionality to an already existing database system.

The experiments were performed using the set of queries presented in Table 4. Table 4 shows the characteristics of the queries in terms of the number of restrictions, the query type and the size of the query response. The restrictions are all equality conditions aggregated by some Boolean operators. Query Q1-Q5 are disjunctions of equality conditions, whereas the rest of the queries have additional Boolean operators like AND and NOT. The last column shows the percentage of the response size in terms of the whole database size. Note that the number of restrictions corresponds to the number of tuples which would be included in a correct and complete S_β response and the response size would be same as the number of tuples in the S_α result.

Query Id	Number of Restrictions	Query type	Response Size (tuples)	Database Percentage
Q1	10	OR	20115	10
Q2	20	OR	35452	18
Q3	30	OR	92791	46
Q4	40	OR	106065	53
Q5	50	OR	198869	99
Q6	3	OR, AND	4016	2
Q7	3	OR, AND, NOT	10354	5
Q8	4	OR, AND	24722	12
Q9	3	OR, AND	64028	32

Table 4. Summary of the different queries used for performance testing

In Table 5 we summarize the queries that can be handled by our different proposed schemes.

RDAS1 and all its variants only can manage the queries Q1-Q5, because these are queries in which the Boolean connectives are disjunctions. On the other hand RDAS2 can handle all the queries Q1-Q9, because it is designed to work with queries involving all kinds of Boolean connectives.

RDAS1-agg and RDAS2-agg are implemented only using PMAC as PolyMac cannot be used as an aggregate MAC. As stated before, we implemented the aggregation at the server side with PostgreSQL XOR function and at the client side with the Intel SIMD instruction for xor, this of course has performance implications that we discuss later.

Only RDAS2 is implemented with compression, we name the variant as RDAS2-cmp. As in RDAS1 explicit bitmaps are neither stored nor transmitted, hence the compressed bitmap version is not applicable in case of RDAS1. For compressing the bitmaps and applying logic operations on them we used the Lemire library [12]. This library only implements OR, AND, XOR, operations over compressed bitmaps. This is the reason why report only results for queries Q1-Q9 except Q7 for RDAS2-cmp. Though using these basic operations as provided by the library one can implement other logic operations, but we have not done this, as we feel that for a proof of concept the query classes that we handle would be enough.

Scheme	Queries
RDAS1 PolyMac	Q1-Q5
RDAS1 PMAC	Q1-Q5
RDAS1-agg	Q1-Q5
RDAS2 PolyMac	Q1-Q9
RDAS2 PMAC	Q1-Q9
RDAS2-agg	Q1-Q9
RDAS2-cmp PolyMac	Q1-Q6 Q8-Q9
RDAS2-cmp PMAC	Q1-Q6 Q8-Q9

Table 5. Summary of the different scenarios used for performance testing

9.3 Experimental results on RDAS1 and its Variants

In Table 6 we report the time required for executing the set of queries (Q1,Q2,Q3,Q4,Q5). We report the normal time (i.e., the time for execution without any authentication) along with the times required for RDAS1 with both PolyMac and PMAC, and for RDAS1-agg. All reported times are in milliseconds and is the average of 250 executions of the same query. The reported time involves all steps in the authentication procedure, i.e., time for query translation, time required for the server to respond and the time for client side verification. In Table 6 we also report the extra overhead of each of our schemes over the normal scheme without authentication. The data presented in Table 6 is also presented pictorially in Figure 4.

Query	Normal time	RDAS1 [PolyMac]		RDAS1 [PMAC]		RDAS1-agg [PMAC]	
Id	-	Avg time	Extra Overhead(%)	Avg time	Extra Overhead(%)	Avg time	Extra Overhead(%)
Q1	437.05	525.74	20.29	522.01	19.44	879.69	101.28
Q2	747.58	1062.52	42.13	1048.32	40.23	1743.47	133.22
Q3	1708.08	2668.86	56.25	2652.40	55.29	4025.26	135.66
Q4	1944.71	2895.41	48.88	2877.48	47.97	4357.16	124.05
Q5	3739.53	7568.17	102.38	7564.11	102.27	10357.86	176.98

Table 6. Execution times for OR queries with RDAS1.

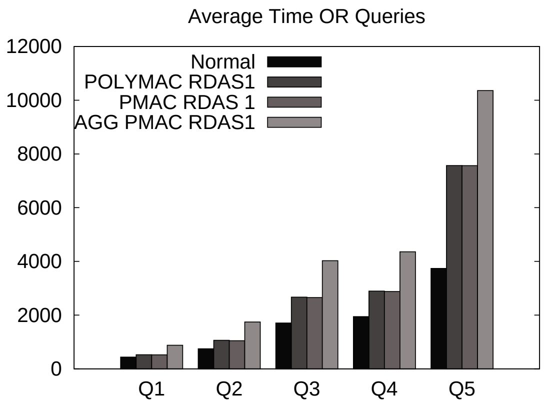


Fig. 4. RDAS1 OR queries times (milliseconds).

The results of Table 6 show that in general PMAC is marginally faster than PolyMac. Even though the aggregated scheme is implemented with PMAC, the results shows that this scheme is very expensive, in general it takes a bit more than the double of the normal time (time to respond the query without authentication). This can be explained by the fact that in case of RDAS1-agg the server and the client need to calculate the aggregated tags, this is done by XORing all the involved tags. We will see in Table 7 that the time for responding the query increases significantly in RDAS1-agg compared to RDAS1.

In Table 7 we present separately the following times:

1. Time required for query translation (the Φ function).
2. Time required by the server to respond to the query (the Ψ function).
3. The time required for verification (the $\mathcal{V}$ function).

Note that out of all the above procedures only the response procedure (Ψ) is executed in the server, and the other two procedures runs in the client. Table 7 clearly shows that the time required for query translation is negligible, the most of the time is spent in the server response and the verification procedures.

The server response procedure for RDAS1-agg is significantly more than that of RDAS1. This is because that in case of RDAS1, the server needs to aggregate the MACs before sending them. Thus, the server needs to compute xors equal to the total number of tuples in S_α and S_β. And this operations are not required to be performed by the server in case of RDAS1. As the server side is implemented using the PostgreSQL, these XORs are expensive. But the verification procedure in case of RDAS1-agg is marginally faster than in RDAS1. Note that the number of MAC computations in the verification process of both RDAS1 and RDAS1-agg are the same. But, in case of RDAS1 the verification process requires to compare each computed tag with the tag received. But in case of RDAS1-agg, individual tag verification is not required, here the computed tags are xor-ed and the final value is compared with the aggregated tag, which is received as a part of the query response. As stated, the verification process is implemented using SIMD instructions, hence the aggregation does not take as much time as individual comparison of the tags.

Query	RDAS1				RDAS1-agg		
Id	ϕ	ψ	$\mathcal{V}$[PolyMac]	$\mathcal{V}$[PMAC]	ϕ	ψ	$\mathcal{V}$ [PMAC]
Q1	.0127	442.78	83.26	77.86	.0170	811.60	76.56
Q2	.0260	843.92	216.33	202.67	.0339	11558.59	201.84
Q3	.0262	1806.30	862.74	844.75	.0422	3209.64	827.17
Q4	.0262	1980.46	922.22	901.82	.0382	3489.79	884.24
Q5	.0419	3983.31	3638.79	3560.35	.0677	6868.57	3532.19

Table 7. Execution times for primitives with RDAS1-OR queries.

The communication overhead of RDAS1 and RDAS1-agg is discussed in Sections 4.2 and 6.1 respectively. Specifically Eq. (4) gives an upper bound on the response size for RDAS1 and Eq. (6) gives the same for RDAS1-agg. In Table 8 we give the numerical values of the response size for the specific queries used in our implementation. In Table 8, columns 2 and 3 represents the size in tuples for S_α and S_β respectively. The values in columns labeled $\mathsf{siz}_{\mathsf{RD1}} - \mathsf{siz}$ and $\mathsf{siz}_{\mathsf{AGG-RD1}} - \mathsf{siz}$ represents the extra size of the response (in bytes) in case of RDAS1 and RDAS1-agg respectively. These were calculated using equations (4) and (6) respectively. For these calculations we assume the tag size to be 16 bytes, the size for the `Nonce` and `RowNo` as 4 bytes, and $s_{\mathsf{Name}} + s_{\mathsf{sk}} = 200$ bytes. Table 8 clearly shows that RDAS1-agg has significantly lower communication cost compared to RDAS1.

Query Id	Size S_α	Size S_β	$\mathsf{siz}_{\mathsf{RD1}} - \mathsf{siz}$	$\mathsf{siz}_{\mathsf{AGG-RD1}} - \mathsf{siz}$
Q1	20115	10	404500	82532
Q2	35452	20	713440	145920
Q3	92791	30	1862420	377316
Q4	106065	40	2130100	432452
Q5	198869	50	3988380	805708

Table 8. Object Verification Extra-Size for RDAS1 in bytes.

9.4 Experimental results on RDAS2 and its Variants

In this section we present the results of the variants of RDAS2 in the same way as we did in the previous section for RDAS1. In Table 9 we report the time required for executing the set of queries (Q1-Q9) with RDAS2, RDSAS2-agg and RDAS2-cmp. In Table 10 we report the time taken for various sub-processes involved in the query execution. In Figures 5, 6 we present the data of Table 9 pictorically. In Table 11 we report data corresponding to the size of the response.

Query	Normal	RDAS2 [PolyMac]		RDAS2 [PMAC]		RDAS2-agg [PMAC]		RDAS2-cmp [PolyMac]		RDAS2-cmp [PMAC]	
Id	time	Avg time	Over-head(%)	Avg time	Over-head(%)	Avg time	Over-head(%)	Avg time	Over-head(%)	Avg time	Over-head(%)
Q1	437.05	515.06	17.85	508.88	16.43	870.48	99.17	496.03	13.49	496.82	13.67
Q2	747.58	971.57	29.96	969.90	29.74	1671.36	123.57	963.94	28.94	959.39	28.33
Q3	1708.08	2316.89	35.64	2311.90	35.35	3719.12	117.74	2302.87	34.82	2178.18	27.52
Q4	1944.71	2504.30	28.78	2502.96	28.68	3988.83	105.11	2454.10	26.19	2284.95	17.49
Q5	3739.53	6331.86	69.32	6326.28	69.17	9130.61	144.16	6298.63	68.43	6266.40	67.57
Q6	108.64	184.93	70.22	175.72	61.75	322.71	197.04	175.67	61.70	166.92	53.64
Q7	182.37	286.44	57.06	276.79	51.77	488.53	167.87	-	-	-	-
Q8	374.05	572.92	53.17	558.98	49.44	954.73	155.24	570.75	52.59	545.83	45.93
Q9	784.72	1179.85	50.35	1162.56	48.15	1874.71	138.90	1166.25	48.62	1142.06	45.54

Table 9. Execution times with RDAS2.

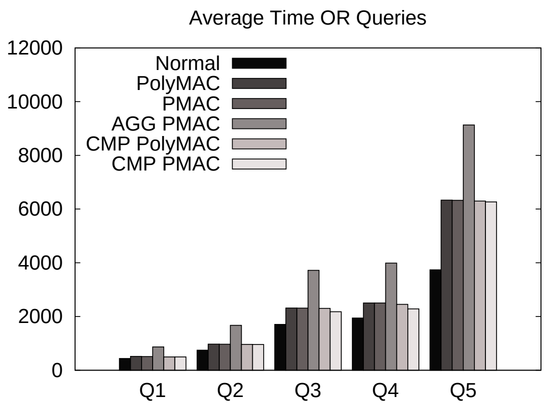
Query	RDAS2				RDAS2-agg[PMAC]			RDAS2-cmp			
Id	ϕ	ψ	$\mathcal{V}$ [PolyMac]	$\mathcal{V}$ [PMAC]	ϕ	ψ	$\mathcal{V}$	ϕ	ψ	$\mathcal{V}$ [PolyMac]	$\mathcal{V}$[PMAC]
Q1	0.0117	457.20	55.62	55.09	0.2367	806.88	60.51	0.0128	451.98	54.87	54.79
Q2	0.0227	857.55	109.95	108.99	0.2033	1534.68	120.56	0.0255	841.26	114.26	115.11
Q3	0.0280	1815.84	500.97	498.63	0.1583	3177.27	513.94	0.0304	1805.03	498.38	499.64
Q4	0.0254	1995.49	514.39	513.00	0.1618	3451.78	523.04	0.0291	1949.91	512.31	511.50
Q5	0.0413	3962.19	2368.64	2355.91	0.2354	6875.67	2400.21	0.0442	3970.33	2369.98	2362.77
Q6	.0048	163.83	20.19	14.90	0.1663	301.95	20.07	0.0052	152.19	14.37	13.59
Q7	.0057	252.35	33.49	27.88	0.1662	451.71	33.90	-	-	-	-
Q8	.0085	497.18	70.75	61.29	0.2183	878.69	69.25	0.0097	490.65	60.42	60.34
Q9	.0061	1026.55	146.92	136.15	0.1671	1725.26	145.86	0.0065	1008.72	134.41	135.49

Table 10. Execution times for primitives with RDAS2.

Query Id	Size S_α	Size S_β	$\mathsf{siz}_{\mathsf{RD2}} - \mathsf{siz}$	$\mathsf{siz}_{\mathsf{AGG-RD2}} - \mathsf{siz}$	$\mathsf{siz}_{\mathsf{CMP-RD2}} - \mathsf{siz}$
Q1	20115	10	653910	331942	524520
Q2	35452	20	1212260	644740	796649
Q3	92791	30	2610650	1125546	1933237
Q4	106065	40	3127740	1430092	2187503
Q5	198869	50	5235430	2052758	4086144
Q6	4016	3	155803	91531	129932
Q7	10354	3	282563	116883	277908
Q8	24722	4	569923	199500	562899
Q9	64028	3	1356043	331579	1338472

Table 11. Object Verification Extra-Size in bytes.

From the Tables we can infer the following:



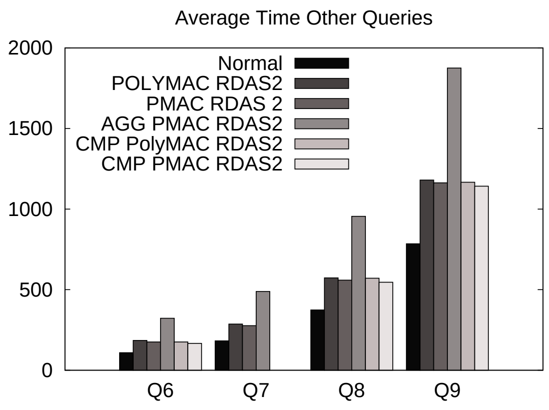


Fig. 5. RDAS2 OR queries times (milliseconds).

Fig. 6. RDAS2 queries times (milliseconds).

1. RDAS2-cmp has the best performance in terms of time for all queries, and RDAS2-agg has the worst performance. The reason for good performance of RDAS2-cmp is probably due to the fact that as compression reduces the size of the bitmaps, hence operations on them can be performed much more efficiently. The performance of RDAS2-agg can be further improved by optimizing the server. Table 10 clearly shows that for RDAS2-agg most of the time is consumed by the query response procedure (Ψ).
2. All versions of RDAS2 performs better than the corresponding versions of RDAS1. This is because in all versions of RDAS1 the verification process has to create the bitmaps from the responses. This can take up some time. But in case of RDAS2, the bitmaps are already there as part of the response.
3. From Table 11, we can see that RDAS2-agg have the best performance in terms of communication cost, followed by RDAS2-cmp and RDAS2. But all versions of RDAS2 have more communication cost compared to the corresponding versions of RDAS1.

10 Discussions and Comparisons

As stated earlier there are other schemes available in the literature which achieves the functionality of authenticated query processing. The main novelty of our scheme is in the use of bitmaps and ability to work on queries other than range queries, moreover we try to analyze security in formal terms and also link the security of the scheme with the security of the message authentication code.

In this section we aim to provide a comparison of our scheme with the other existing schemes, and also point out the limitations in our scheme. Providing an experimental comparison of our scheme with the existing ones is beyond the scope of this work. Some authors (for example [37]) compare the storage and communication cost of the protocols in asymptotic terms, which also do not reveal the complete picture. Here we would point out some of the differences of our framework in comparison to the others with respect to functionality and costs.

As stated earlier, most existing schemes use an additional authenticated data structure to ensure completeness. The additional data structure is usually a tree, in its barest form a Merkle hash tree is used [5, 16]. In some cases variants of B+ trees have also been used as the additional data structure [14, 27, 26]. The other significant direction is use of signatures. By signing the individual tuples with a secure signature scheme, as was done in [20, 27] one can ensure correctness but not completeness. To attain completeness only with signatures a method in [21] was proposed, where chains of signatures are constructed based on a specific sort order of the attribute values. Such types of signatures allow verification of completeness for range queries, without the use of additional data structures. Signature aggregation has also been used to reduce the size of the responses [20, 19, 21, 26]. In what follows we summarize some of the salient features of the previous works available in the literature and compare and contrast it with the RDAS framework.

Type of queries: In the literature, all basic schemes are designed to handle mostly range queries on numerical attributes. In most cases, to handle additional query types, drastic changes in the basic schemes are required. For most tree based schemes, even handling tuples with repeated values of attributes needs special treatment [21, 26]. In some tree based schemes join processing is achieved in rather a straight forward way [26, 35, 14, 29], which leads to significant increase in the size of the query responses and also the stored data in the server side. In [28] a special tree structure called authenticated aggregation B tree was specifically designed to handle aggregation queries, other tree based schemes cannot handle aggregation queries as a part of the scheme.

The signature based schemes like in [21, 26] which uses chain signatures are mainly designed for range queries and are not efficient for other types of queries. It has been claimed in [26] that joins can be handled, but here too the computational costs and the size of the responses would be prohibitively large.

RDAS2 can handle selects involving arbitrary Boolean operations. These query types can be modified to handle range queries, without any extra overhead. In the recent years other efficient bitmap encodings, like the range encodings [3, 4] have been proposed. Such bitmap encoding can provide functionality of range queries in a more efficient way. Such extensions are straightforward and can be easily implemented. Projections and aggregations cannot be handled directly by the protocol, but they can be implemented in the client side without any difficulties. Moreover, simple join queries can also be accommodated if some additional information is stored through join bitmaps [24, 1]. Hence the spectrum of query types that can be handled by the RDAS framework is comparable to the existing schemes.

In the following discussion we will denote the number of rows in a database by n and the number of attributes by m. By n_q we shall mean the number of tuples included in a correct and complete response of a query q, and l_q will denote the number of bitmaps involved in a query q.

Storage Costs: Tree based schemes need to store a tree for each attribute. Asymptotically, a tree based scheme (such as the one described in [14, 26]) requires $O(mn)$ extra storage. Whereas in case of RDAS1 we require $O(n + N)$ storage, where N is the cardinality of all the attributes present. We know that $N \leq mn$, but in most scenarios N is smaller than mn, and in certain scenarios $N << mn$. In case of the basic RDAS2, we require $O(n + Nn)$ storage as we require to store the bitmaps also corresponding to each attribute and its value. Assuming $N = O(mn)$, we have the asymptotic storage requirement for RDAS2 as $O(mn^2)$, but the constants involved in this are much smaller than in the case of tree based solutions, and this is an extreme overestimate for databases with low cardinality attributes. Moreover, in case of RDAS2 the bitmaps can be compressed. It is difficult to give a proper estimate of the amount of compression that can be achieved in general. But, it is clear that when the attribute cardinalities increases, the bitmaps become more sparse (in the sense that they would have lower Hamming weights), which would allow better compression. In case $N = mn$, each bitmap would have only one 1 and rest zeros, and can be encoded in constant length irrespective of the number of rows in the database. Thus, it is expected that with increase in N the encoded size of the bitmaps decreases. In general terms, it has been said (for example in [33]) that the sizes of the compressed bitmap indices are relatively small compared with the typical B-tree indices. This is true even for attributes with very high cardinalities.

The schemes based on signature chains or aggregate signatures like [29, 21] also uses $= O(mn)$ storage.

Query execution costs: The tree based schemes (as [14]) do not have any extra computational overhead in query execution. This is also true for the basic RDAS1 and RDAS2. In case of RDAS2 − agg, for query execution the tags are to be aggregated which requires a number of xor operations which grows linearly with the response size. But in tree based schemes, answering a query requires traversing the tree to find the relevant node, and for multiple attributes this need to be done in multiple trees, and there exists no trivial way to combine the trees involving different attributes. In RDAS, query execution is simpler, moreover the bitmaps stored in case of RDAS2 can even act as indexes and thus make query execution further efficient. In case of schemes using signature chains [29, 21], the signatures are also pre computed, hence additional computation in query execution is not required. But schemes which uses aggregated signatures signatures are required to be aggregated for responding queries. In such a scenario, the server needs to aggregate $O(n_q)$ signatures. It is to be noted that aggregating signatures is much more costly than aggregating MACs. For example if RSA signatures are used then to aggregate two signatures one requires a modular multiplication modulo the RSA modulus (which should be at least 1024 bits long).

Query Verification costs: The schemes based on trees structures ([14]) requires to compute $O(mn_q)$ hashes plus one signature to verify a query. In the case of [26], $O(mn_q)$ signatures are required to be computed, which are very expensive. On the other hand, RDAS1 and RDAS2 requires the computation of $O(n_q + l_q \cdot n)$ tags, where l_q is the number of bitmaps involved in the query. RDAS2 only differs from RDAS1 in that it is necessary to build the involved bitmaps. In case of schemes using signature chains ([21, 29]) the cost is similar to the tree approach.

Communication costs: In RDAS1 and RDAS2, the extra communication cost is $O(n_q + n \cdot l_q)$. Once more the difference between RDAS1 and RDAS2 it is that in RDAS2 the communication costs grows because bitmaps are also sent as part of the response. The extra cost can be reduced by compression as in RDAS2-cmp. In case of RDAS1-agg the extra overhead is constant, since just two aggregated tags are sent irrespective of the response size. Finally in RDAS2-agg the object verification size has a bound of $O(n \cdot l_q)$ where l_q is the number of bitmaps involved in the query. The schemes based on trees ([14]) has as communication cost of $O(\log n)$ hashes that need to be sent. The scheme in [26] has a constant communication cost, as only an aggregate signature is sent here irrespective of the response size. The schemes described in [29, 21] needs to send $O(mn_q)$ hashes plus one aggregated signature.

11 Conclusion

We presented RDAS a generic framework for authenticated query processing and provided the syntax and security definition of a RDAS. We also provided two concrete constructions RDAS1 and RDAS2 which uses bitmap indices and message authentication codes in a novel way. We discussed several ways in which the basic schemes can be extended to accommodate various functionalities. In particular we described the use of aggregate message authentication codes and compressed bitmaps. We reported extensive experimental data on our schemes, our results suggests that RDAS can be a viable option for authenticated query processing in real life.

A serious limitation of the proposed schemes is that they do not allow database updates. We are currently working on this problem, to extend this framework for dynamic databases.

References

1. Ladjel Bellatreche, Rokia Missaoui, Hamid Necir, and Habiba Drias. A data mining approach for selecting bitmap join indices. *JCSE*, 1(2):177–194, 2007.
2. Ladjel Bellatreche, Rokia Missaoui, Hamid Necir, and Habiba Drias. Selection and pruning algorithms for bitmap index selection problem using data mining. In Il Yeal Song, Johann Eder, and Tho Manh Nguyen, editors, *DaWaK*, volume 4654 of *Lecture Notes in Computer Science*, pages 221–230. Springer, 2007.
3. Chee Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD Conference*, pages 355–366. ACM Press, 1998.
4. Chee Yong Chan and Yannis E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD Conference*, pages 215–226. ACM Press, 1999.
5. Premkumar T. Devanbu, Michael Gertz, Charles U. Martel, and Stuart G. Stubblebine. Authentic data publication over the internet. *Journal of Computer Security*, 11(3):291–314, 2003.
6. A. Frank and A. Asuncion. UCI machine learning repository, 2010.
7. Michael T. Goodrich, Roberto Tamassia, and Nikos Triandopoulos. Super-efficient verification of dynamic outsourced databases. In Malkin [15], pages 407–424.
8. Shay Gueron and Michael E. Kounavis. Efficient implementation of the galois counter mode using a carry-less multiplier and a fast reduction algorithm. *Inf. Process. Lett.*, 110(14-15):549–553, 2010.
9. Hakan Hacigümüs, Sharad Mehrotra, and Balakrishna R. Iyer. Providing database as a service. In *ICDE*, pages 29–. IEEE Computer Society, 2002.
10. Jonathan Katz and Andrew Y. Lindell. Aggregate message authentication codes. In Malkin [15], pages 155–169.
11. Arie Shoshani Kesheng Wu, Ekow Otoo and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.
12. Daniel Lemire. lemurbitmaindex. url:http://code.google.com/p/lemurbitmapindex/, 2013.
13. Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. *CoRR*, abs/0901.3751, 2009.
14. Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *SIGMOD Conference*, pages 121–132. ACM, 2006.
15. Tal Malkin, editor. *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*, volume 4964 of *Lecture Notes in Computer Science*. Springer, 2008.
16. Charles U. Martel, Glen Nuckolls, Premkumar T. Devanbu, Michael Gertz, April Kwong, and Stuart G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.
17. Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.
18. Kyriakos Mouratidis, Dimitris Sacharidis, and HweeHwa Pang. Partially materialized digest scheme: an efficient verification method for outsourced databases. *VLDB J.*, 18(1):363–381, 2009.
19. Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Signature bouquets: Immutability for aggregated/condensed signatures. In Pierangela Samarati, Peter Y. A. Ryan, Dieter Gollmann, and Refik Molva, editors, *ESORICS*, volume 3193 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 2004.
20. Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. *TOS*, 2(2):107–138, 2006.
21. Maithili Narasimha and Gene Tsudik. DSAC: integrity for outsourced databases with signature aggregation and chaining. In Otthein Herzog, Hans-Jörg Schek, Norbert Fuhr, Abdur Chowdhury, and Wilfried Teiken, editors, *CIKM*, pages 235–236. ACM, 2005.
22. Maithili Narasimha and Gene Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In Mong-Li Lee, Kian-Lee Tan, and Vilas Wuwongse, editors, *DASFAA*, volume 3882 of *Lecture Notes in Computer Science*, pages 420–436. Springer, 2006.
23. Glen Nuckolls. Verified query results from hybrid authentication trees. In Sushil Jajodia and Duminda Wijesekera, editors, *DBSec*, volume 3654 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2005.
24. Patrick E. O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.
25. Bernardo Palazzi, Maurizio Pizzonia, and Stefano Pucacco. Query racing: Fast completeness certification of query results. In Sara Foresti and Sushil Jajodia, editors, *DBSec*, volume 6166 of *Lecture Notes in Computer Science*, pages 177–192. Springer, 2010.
26. HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. Verifying completeness of relational query results in data publishing. In Fatma Özcan, editor, *SIGMOD Conference*, pages 407–418. ACM, 2005.
27. HweeHwa Pang and Kian-Lee Tan. Authenticating query results in edge computing. In Z. Meral Özsoyoglu and Stanley B. Zdonik, editors, *ICDE*, pages 560–571. IEEE Computer Society, 2004.

28. HweeHwa Pang and Kian-Lee Tan. Verifying completeness of relational query answers from online servers. *ACM Trans. Inf. Syst. Secur.*, 11(2), 2008.
29. HweeHwa Pang, Jilian Zhang, and Kyriakos Mouratidis. Scalable verification for outsourced dynamic databases. *PVLDB*, 2(1):802–813, 2009.
30. Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In Pil Joong Lee, editor, *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.
31. Kurt Stockinger. Bitmap indices for speeding up high-dimensional data analysis. In Abdelkader Hameurlain, Rosine Cicchetti, and Roland Traunmüller, editors, *DEXA*, volume 2453 of *Lecture Notes in Computer Science*, pages 881–890. Springer, 2002.
32. R. Wrembel and C. Koncilia. *Data warehouses and OLAP: concepts, architectures, and solutions.* Gale virtual reference library. IRM Press, 2007.
33. Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. On the performance of bitmap indices for high cardinality attributes. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 24–35. Morgan Kaufmann, 2004.
34. Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.
35. Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. Authenticated join processing in outsourced databases. In *SIGMOD Conference*, pages 5–18, 2009.
36. Yin Yang, Stavros Papadopoulos, Dimitris Papadias, and George Kollios. Spatial outsourcing for location-based services. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *ICDE*, pages 1082–1091. IEEE, 2008.
37. Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese. Efficient query integrity for outsourced dynamic databases. In Ting Yu, Srdjan Capkun, and Seny Kamara, editors, *CCSW*, pages 71–82. ACM, 2012.