CHAPTER 9

AUTHENTIC THIRD-PARTY DATA PUBLICATION

Premkumar Devanbu, Michael Gertz, Charles Martel
Department of Computer Science, University of California, Davis, CA 95616
{devanbu|gertz|martel}@cs.ucdavis.edu

Stuart G. Stubblebine
CertCo, 55 Broad Street, New York, NY 10004
stuart@stubblebine.com

Abstract Integrity critical databases, such as financial data used in high-value decisions, are frequently published over the Internet. Publishers of such data must satisfy the integrity, authenticity, and non-repudiation requirements of clients. Providing this protection over public networks is costly.

This is partly because building and running secure systems is hard. In practice, large systems can not be verified to be secure and are frequently penetrated. The consequences of a system intrusion at the data publisher can be severe. This is further complicated by data and server replication to satisfy availability and scalability requirements.

We aim to *reduce the trust required* of the publisher of large, infrequently updated databases. To do this, we separate the roles of owner and publisher. With a few trusted digital signatures from the owner, an untrusted publisher can use techniques based on Merkle hash trees to provide authenticity and non-repudiation of the answer to a database query. We *do not* require a key to be held in an on-line system, thus reducing the impact of system penetrations. By allowing untrusted publishers, we also allow more scalable publication of large databases.

1. INTRODUCTION

Consider an Internet financial-data warehouse, with historical data about securities such as stocks and bonds, that is used by businesses and individuals to make important investment decisions. The *owner* (or creator) of such a database might be a rating/analysis service (such as Standard & Poors), or a government agency. The owner's data might be needed at high rates, for example by the user's investment tools. We focus our attention on data which changes infrequently and needs to be delivered promptly, reliably and accurately.

One approach to this problem is for the owner of the information to digitally sign the answers to clients' queries, using a private signing key, sk_O. This signature is verified using the corresponding public key, pk_O. Based on the signature, a client can be sure that the answer comes from the owner, and that the owner can't claim otherwise. However, there are several issues here. First, the owner of the data may be unwilling or unable to provide a reliable and efficient database service to handle the needed data rates. Second, the owner needs to maintain a high level of physical security and system security to defend against attacks. This has to be done to protect the signing key, sk_O, which must be resident at the server at all times to sign outgoing data. In practice, large software systems have vulnerabilities, and keeping secret information on a publicly-accessible system is always risky. Using special hardware devices to protect the signing key will help, as would emerging cryptographic techniques like "threshold cryptography," but these methods do not fully solve the system-vulnerability problem, and can be too expensive in our domain, both computationally and financially.

A more scalable approach uses trusted third-party *publishers* in conjunction with a key management mechanism which allows a certified signing key of a publisher to speak for the owner (see also [3]). The database (or database updates) is provided securely to the publisher, who responds to client queries by signing them with it's own (certified) signing key, sk_P. Presumably, the market for useful databases will motivate publishers to provide this service, unburdening database owners of the need to do so. The owner simply needs to sign the data after each update and distribute it to the publisher. As demand increases, more publishers will emerge, or more capable ones, making this approach inherently scalable. But the approach still suffers from the problem and expense of trying to maintain a secure system accessible from the Internet. Furthermore, the client might worry that a publisher engages in deception. She would have to believe that her publisher was both competent and careful with site administration and physical access to the database. She might worry about the private signing-key of the publisher, which is again vulnerable to attacks. To gain trust, the publisher must adopt meticulous administrative practices, at far greater cost. The need for trusted publishers would also increase the reliance on brandnames, which would limit market competition.

In a summary of fundamental problems for electronic commerce [10], Tygar asks "How can we protect re-sold or re-distributed information … ?" We present a solution to this problem in the context of relational databases.

2. BASIC APPROACH

We allow an **untrusted** *publisher* to provide a **verification-object** $\mathcal{VO}$ to a *client* to verify an answer to its database query. The client can use the $\mathcal{VO}$ to gain assurance that the answer is just what the database *owner* would have provided. The verification-object is based on a **summary-signature** that the owner periodically distributes to the publisher (Figure 1).

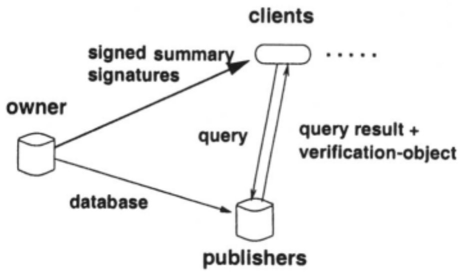

Figure 1. We partition the role of information provider into *owner* and *publisher.* The owner provides database updates to the publisher. The publisher is untrusted. The *client* makes inquiries to the publisher. Its gets responses which can be verified using a returned *verification-object*. Superscripts denote keys known to that party. Only sk_O is secret. The client must be sure of the binding of pk_O to the owner.

The summary-signature is a bottom-up hash computed recursively over B-tree type indexes for the entire set of tuples in each relation of the owner's database, signed with sk_O. Answers to queries are various combinations of subsets of these relations. Given a query, the publisher computes the answer. To show that an answer is correct the publisher constructs a verification-object using the same B-tree that the owner had used to compute the summary-signature. This verification-object validates an answer by providing an unforgeable "proof" which links the answer to the summary-signature. Our approach has several features:

1 Besides its own security, a client needs only trust the key of the owner. The owner only needs to distribute the summary-signature during database updates. So, the owner's private key can be maintained in an "off-line" machine, isolated from network-based attacks. The key itself can be ensconced in a hardware token, which is used only to sign a single hash during updates.

2 Clients need not trust the publishers, nor their keys. In particular, if a particular publisher were compromised, the result would *only* be a loss of service at that publisher.

3 In all our techniques, the verification-object is of size linear in the size of the answer to a query, and logarithmic in the size of the database.

4 The verification-object guarantees that the answer is correct, without any extra or missing tuples.

5 In all of our techniques, the overheads for computing the summary-signature, the $\mathcal{VO}$, and for checking the $\mathcal{VO}$ are reasonable.

6 The approach offers far greater survivability. Publishers can be replicated without coordination, and the loss of one publisher does not degrade security and need not degrade availability.

A correct answer and verification-object will always be accepted by the client. An incorrect answer and verification-object will almost always be rejected, since our techniques make it computationally infeasible to forge a correct verification-object for an incorrect answer. Overall, the approach nicely simplifies the operational security requirements for both owners and publishers.

3. MERKLE HASH TREES

We describe the computation of a Merkle Hash Tree [6] for a relation *r* with *m* tuples and relation schema $R = \langle A_1, \ldots, A_n \rangle$. A more complete description can be found in [4]. Assume that $\mathcal{A} = \langle A_i, \ldots, A_k \rangle$ is a list of attributes from *schema*(*R*). A Merkle Hash Tree, denoted by $MHT(r, \mathcal{A})$, is a balanced binary tree whose leaves are associated with the tuples of *r*. Each node in the tree has a value computed using a collision-resistant hash function *h*:

1 First, compute the *tuple hash* h_t for each tuple $t \in r$, thus

$$h_t(t) = h(h(t.A_1) \; || \; \cdots \; || \; h(t.A_n))$$

The tuple hash (by the collision resistance of the hash function) is a "nearly unique" tuple identifier. We also assume distinct "boundary tuples" t_0, t_{m+1} with artificial attribute values chosen to be smaller (larger) than any real tuple. These are associated with the left (right) most leaves in the tree.

2 Next, compute the Merkle hash tree for relation *r*. We assume that *r* is sorted by the values of $\mathcal{A}$ so that for two distinct tuples $t_{i-1}, t_i \in r$, $t_{i-1}.\mathcal{A} \leq t_i.\mathcal{A}$. Any total order over *r* based on $\mathcal{A}$ will work. We now describe how to compute *V*(*u*) the value associated with a node *u* of $MHT(r, \mathcal{A})$. Let u_i be the leaf associated with t_i.

Leaf-node u_i : $V(u_i) = h_t(t_i)$

Internal node u : $V(u) = h(V(w) \,\|\, V(x))$

where w, x are the children of u. We also refer to w, x as *hash siblings,*

The value of the root is the "root hash" of the Merkle tree. This construction easily generalizes to a higher branching factor $K > 2$, such as in a B^+-tree; however, for our presentation here we use binary trees. If the *owner* and the *publisher* build a *MHT* around index structures that are used in query evaluation, then constructing a $\mathcal{VO}$ is a minor overhead over the query evaluation process itself.

Note that (by the cryptographic assumption of a collision-resistant hash function) if the correct value of the *parent* is known to the client, it is hard for the publisher to forge the value of its children.

Definition 1 (Hash Path)

For a leaf node u_i in $MHT(r, \mathcal{A})$ the nodes necessary to compute the hash path up to the root hash is denoted as $path(t_i)$. Such a hash path always has the length d, the depth of node u_i, $d \leq \lceil \log(m+2) \rceil$. With t_i and the values of siblings of the nodes on the path we can recompute the value at the root. Hash paths can also be provided for non-leaf nodes.

The d values of the siblings of $path(t_i)$ constitute the $\mathcal{VO}$ showing that tuple t_i is actually in the relation. Indeed any interior node within the hash tree can be authenticated by giving a path to the root. Hash paths show that tuples actually exist in a relation; to show that set of tuples is complete, we need to show boundaries. Any non-empty contiguous sequence $q = \langle t_i, \ldots, t_j \rangle$ of leaf nodes in a Merkle Hash Tree $MHT(r,\mathcal{A})$ uses t_{i-1} and t_{j+1} as its *boundary tuples.*

Any non-empty contiguous sequence $q = \langle t_i, \ldots, t_j \rangle$ of leaf nodes in a Merkle Hash Tree $MHT(r, \mathcal{A})$ has a lowest common ancestor $LCA(q)$. This situation is illustrated in Figure 2. Given $LCA(q)$, one can show a hash path $path(LCA(q))$ to the authenticated root hash value. After this is done, (shorter) hash paths from its boundary tuples t_{i-1} and t_{j+1} to $LCA(q)$ and the values of $t_i, \ldots, t_j$ allow us to compute $V(LCA(q))$. We can then compute the root hash using the values of the siblings of $path(LCA(q))$. This lets us verify of that $t_{i-1}, \ldots, t_{j+1}$ are associated with contiguous leaves in our tree.

We finally define desirable properties of the answer set q returned by *publisher*, in terms of the correct answer that would have been returned by *owner.*

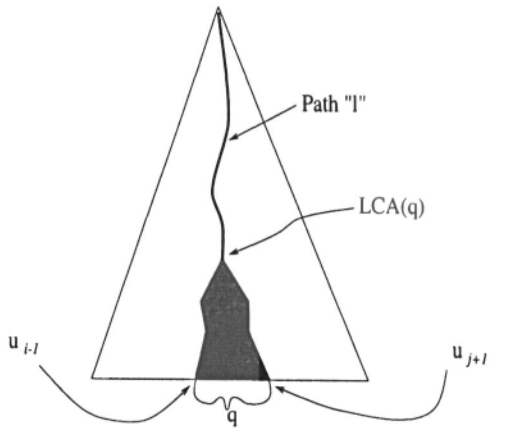

Figure 2. A Merkle tree, with a contiguous subrange $q = \langle t_i, \ldots, t_j \rangle$, with a least common ancestor LCA(q), and upper and lower bounds. Note verifiable hash path "1" from LCA(q) to the root, and the proximity subtrees (thick lines) for the "near miss" tuples u_{i-1} and u_{j+1} which show that q is complete.

Definition 2 *The answer given by* publisher *to a query q is* inclusive *if it contains only the tuples that would have been returned by* owner, *and is* complete *if it contains all the tuples that would have been returned by* owner.

4. BASE LEVEL RELATIONAL QUERIES

In this section we outline the computation of $\mathcal{VO}$ for answers to basic relational queries. We illustrate the basic idea behind our approach for selection and projection queries in Section 4.1 and 4.2, respectively. Slightly more complex types of queries (join queries) and set operators are discussed in Sections 4.3 and 4.4.

4.1. SELECTIONS

Assume a selection query $\sigma_{A_i \Theta c}(r)$ (c $\equiv$ constant) that asks for tuples with attribute values for A_i in a specified range. Assuming that the tree $MHT(r, A_i)$ has been constructed, we can provide compact $\mathcal{VO}$s for the answer q to a query. We consider two cases: when $q = \{\}$, and otherwise. If $q \neq \{\}$, assume a set of answer tuples $t_i, t_{i+1}, \ldots, t_j$ which build a contiguous sequence of leaf nodes in $MHT(r, A_i)$. We simply include a couple of boundary tuples and return the set $t_{i-1}, \ldots, t_{j+1}$, along with the hash paths to t_{i-1} and t_{j+1}. If q is empty, just the boundary tuples are returned. In either case, the size of the $\mathcal{VO}$ is $O(\mid q \mid + \log_2 \mid r \mid)$.

In [4] we present a formal proof that our construction $\mathcal{VO}$s for selection queries is secure:

Lemma 3 *If* publisher *cannot engineer collisions on the hash function or forge signatures on the root hash value, then if* client *computes the right authenticated root hash value using the* $\mathcal{VO}$ *and the answer provided for* selection *queries, then the answer is indeed complete and inclusive.*

4.2. PROJECTIONS

For queries of the pattern $\pi_{\mathcal{A}}(r), \mathcal{A} \subset schema(R)$, the projection operator eliminates some attributes of the tuples in the relation r, and then eliminates duplicates from the set of shortened tuples, yielding the final answer q. A user can choose many different ways to project from a relation R; if this choice is dynamic, it may be best to leave the projection to the *client.* However, the *client* then gets a potentially large intermediate result $\mathcal{I}$; so the $\mathcal{VO}$ will be linear in size $\mid \mathcal{I} \mid$, rather than in the smaller final result $\mid q \mid$. We note that we can, if necessary, mask some of the attributes from the *client;* with just the hash of those attributes in each tuple, the *client* can compute the tuple hash.

Consider, however, an often-used projection $\pi_{\mathcal{A}}(r)$ which projects onto attributes where duplicate elimination will remove numerous tuples. Given the pre-projection tuple set, the *client* would have to do all this work. Now, suppose we have a Merkle tree $MHT(r, \mathcal{A})$, *i.e.,* we assume that the sets of retained attribute values can be mapped to single values with an applicable total order. In this case, we can provide a $\mathcal{VO}$ for the projection step that is linear in the size of the projected result q.

Each tuple t in the result set q may arise from a set $S(t) \subseteq r$ with tuples having identical values for the projected attribute(s) $\mathcal{A}$. We must show that the set q is inclusive and complete:

1 To prove $t \in q$, we show the hash path from *any* witness tuple $y \in S(t) \subseteq r$ to the Merkle Root. However, "boundary" tuples make better witnesses, as we describe next.

2 To show that there are no tuples missing, say between t and t', $(t, t' \in q)$, we just show that $S(t), S(t'), \subseteq r$ are contiguous in the sorted order. Hash paths from two "boundary" tuples $y \in S(t)$ and $x \in S(t')$ that occur next to each other in the Merkle tree can do this.

We observe that both the above bits of evidence are provided by displaying at most $2 \mid q \mid$ hash paths, each of length $\lceil \log_2 r \rceil$. This meets our constraint that the size of the verification object be bounded by $O(\mid q \mid \log_2 \mid r \mid)$.

Constructing Merkle trees to provide compact $\mathcal{VO}$s for duplicate elimination with every possible projection might be undesirable. We might construct trees for only highly selective, common projection attributes, and leave the other duplicate eliminations to the client.

4.3. JOINS

Joins between two or more relations, specially equi-joins where relations are combined based on primary key – foreign key dependencies, are very common. We focus on pairwise joins of the pattern $r \bowtie_C s$ where C is a condition on join attributes of the pattern $A_R \Theta A_S$, $A_R \in schema(R), A_S \in schema(S)$, $\Theta \in \{=,<,>\}$. For Θ being the equality predicate, we obtain the so-called equi-join. We show 3 different approaches, for different situations.

Given a query of the pattern $r \bowtie_C s$, one structure that supports computation of very compact $\mathcal{VO}$s for the query result is based on the materialization (i.e., the physical storage) of the Cartesian Product $r \times s$. This structure supports the three types of joins, which can all be formulated in terms of basic relational algebra operators, i.e., $r \bowtie_{A_R \Theta A_S} s \equiv \sigma_{A_R \Theta A_S}(r \times s)$. Assume $m =| r |, n =| s |$. The verification structure for $r \bowtie_C s$ queries is constructed by sorting the Cartesian Product according to the *difference* between the values for A_R and A_S, assuming such an operation can be defined, at least in terms of "positive", "none" or "negative". This yields three "groups" of leaf nodes in the Merkle Tree: (1) nodes for $t.A_R - u.A_S$ for two tuples $t \in r, u \in s$ is 0, thus supporting equi-joins, (2) nodes where the difference is positive, for the predicate >, and (3) nodes where the difference is negative, for the predicate <. If only simple Θ joins, with $\Theta \equiv =, >$ or $<$ are desired, there is no need to construct binary Merkle trees over the entire cross product—we can just group the tuples in $R \times S$ into the three groups, hash each group in its entirely, and append the three hashes to get the root hash. In this case, the $\mathcal{VO}$s for the three basic Θ queries would consist only of 2 hash values!

For equi-join queries, an optimized structure, presented briefly below, can be used. Rather than the full Cartesian Product $r \times s$, we materialize the *Full Outer Join* $r ⟗ s$ which pads tuples for which no matching tuples in the other relation exist with null values (see, e.g., [2, 8]). The result tuples obtained by the full outer-join operator again can be grouped into three classes: (1) those tuples $tu, t \in r, u \in s$, for which the join condition holds, (2) tuples from r for which no matching tuples in s exist, and (3) tuples from s for which no matching tuples in r exist. Constructing a VO for the result of query of the pattern $r \bowtie_{A_R \theta A_S} s$ then can be done in the same fashion as outlined above.

Suppose R and S have B-tree indices over the join attributes, and these trees have been Merkle-hashed; also suppose, without loss of generality, that of the two relations, r has the smaller number of distinct values, say r^d, and that the size of the answer is q. We can now provide larger VOs of size $O(r^d \log n + r^d \log m + |q|)$ in the worst case[1]. This is done by doing a "verified" merge of the leaves of the two index trees. Whenever the join attributes have the right θ relation, witness hash paths in the trees for R and S are provided to show inclusiveness of the resulting answer tuples; when tuples in r or s are skipped during the merge, we provide a pair of proximity subtrees in R or S respectively to justify the length of the skip. This conventional approach to joins gives rise to larger VOs than the approach described above, but at reduced costs for *publisher* and *owner.*

4.4. SET OPERATIONS

Consider set operations over relations u and v, which may be intermediate query results, u and v may be subsets of some relations r and s respectively, which are each sorted (possibly on different attributes) and have its own Merkle tree $MHT(r, \mathcal{A})$ and $MHT(s, \mathcal{A}')$, the root of which is signed as usual. We consider unions and intersections.

Union. In this case, the answer set is $q = u \cup v$. The *client* is given VOs for u and v, along with VOs for both; *client* verifies both VOs, and computes $u \cup v$. This can be done in $O(u + v)$ using a hash merge. Since $|q|$ is $O(|u| + |v|)$, the overall VO, and the time required to compute and verify the answer, are linear in the size of the answer.

Intersection. The approach for union, however, does not produce compact VOs for set intersection. Suppose $q = u \cap v$ where u and v are as before: note that often $|q|$ could be much smaller than $|u| + |v|$. Thus, sending the VOs for u and v and letting the *client* compute the final result could be a lot of extra work. We would like a VO of size $O(|q|)$. If Merkle trees exist for u and v, we can do inclusiveness in $O(|q|)$: *publisher* can build a VO for q with $O(|q|)$ verification paths, showing elements of q belong to both u and v. Completeness is harder. One can pick the smaller set (say u) and for each element in $u - q$, construct a VO show that it is not in v. In general, if u and v are intermediate results not occurring contiguously in the same Merkle

[1]While we can construct pathological cases that require VOs of this size, practical cases may often be better, being closer to $O(q \log n + q \log m)$. Further empirical study is needed.

tree, such a $\mathcal{VO}$ is linear in the size of the smaller set (say u). A similar problem occurs with set differences $u - v$.

We have not solved the general problem of constructing $\mathcal{VO}$'s linear in the size of the result for intersections and set differences. Indeed, the question remains as to whether (in general) linear-size $\mathcal{VO}$s *can* even be constructed for these objects. However, we have developed an approach to constructing linear-size $\mathcal{VO}$s for a common type of intersection, *range query,* for example, a selection query where the age and salary fall in specified ranges. For these, we use a data structure drawn from computational geometry called a *multi-dimensional range tree.* This also supports set differences over range queries on several attributes.

In d-dimensional computational geometry, when one is dealing with sets of points in d-space, one could ask a d-space range query. Consider a spatial interval $(< x_1^1, x_2^1 > \ldots < x_1^d, x_2^d >)$; this represents an axis-aligned rectilinear solid in d-space. A query could ask for the points that occur within this solid. Such problems are solved efficiently in computational geometry using so-called *Range Trees* (See [5], Chapter 5). We draw an analogy between this problem and a query of the form

$$\sigma_{c_1^1 < A_1 < c_1^2}(r) \bigcap \cdots \bigcap \sigma_{c_d^1 < A_d < c_d^2}(r)$$

where $\{A_1, \ldots A_d\} \subseteq schema(R)$. We use the multi-dimensional range tree (*mdrt*) data structure to solve such queries and provide compact verification objects. For brevity, we omit the full details of our approach. However, in [4], we show how to construct $\mathcal{VO}$s for "d"-way range queries such as the ones shown above. We also argue that the size of these $\mathcal{VO}$s, for a relation with size n, is $O(\mid q \mid + \log^d n)$. The *mdrt* itself uses $O(n \log^{d-1} n)$ space and can also be constructed in time $O(n \log^{d-1} n)$. While the data structure arises out of computational geometry, it can be used with any domain that has enough structure to admit a total order. Full details are discussed in [4]

5. CONCLUSIONS AND FUTURE RESEARCH

We have explored the problem of authentic third party data publication. In particular, we have developed several techniques that allow untrusted third parties to provide evidence of *inclusive* and *complete* query evaluation to clients without using public-key signatures. In addition, the evidence provided is linear in the size of the query answers, and can be checked in linear time. Our techniques can involve the construction of complex data structures, but the cost of this construction is amortized over more efficient query evaluation, as well as the production of compact verification objects. Such pre-computation of views and indexing structures are not uncommon in data warehousing applications.

We now examine some pragmatic considerations in using our approach, as well as related work and future research.

Query processing flexibility. What queries can be handled? A typical SQL "**select . . . from . . . where** ..." can be thought of one or more joins, followed by a (combined) selection, followed by a projection. A multi-dimensional range tree can be used for both efficient evaluation and construction of compact $\mathcal{VO}$s for such queries. Specifically, consider a query that involves the join of two relations R and S, followed by a series of selections and a final projection. Let's assume a Theta-join over attribute A_1 (occurring in both relations), followed by selections on attributes A_2 and A_3, and a final projection on several attributes, jointly represented by A_4. Full details are deferred to [4]. However, to summarize briefly: such a query can be evaluated by constructing a mdrt, beginning with the join attributes, followed by trees for each selection attribute, and perhaps finishing with a tree for some selective projection attributes.

Conventions. It is important to note that all interested parties: *owner*, *publisher* and *client*, share a consistent schema for the databases being published. In addition there needs to be secure binding between the schema, the queries and the query evaluation process over the constructed Merkle trees. A convention to include this information within the hash trees needs to be established. All parties also need to agree on the data structures used for the $\mathcal{VO}$. It is also important that the *publisher* and the *client* agree upon the format in which the $\mathcal{VO}$ together with the query result is encoded and transmitted. Tagged data streams such as XML provide an attractive option.

Recent Query Evaluations. Verifiers must verify that query evaluation is due to an "adequately recent" snapshot of the database and not an old version. We assume the technique of recent-secure authentication [9] for solving this problem. Risk takers (e.g., organizations relying on the accuracy of the data) specify freshness policies on how fresh the database must be. The digital signatures over the database include a timestamp of the last update as well as other versioning information. Clients interpret the timestamps and verify the database is adequately recent with respect to their freshness policies.

Query flexibility. For efficient verification of query answering, we make use of different trees over sorted tuple sets. Without such trees, our approach cannot provide small verification objects. This points to a limitation of our approach—only queries for which Merkle trees have been pre-computed can be evaluated with compact verification objects. Our approach cannot support arbitrary interactive querying with compact verification objects. Arbitrary interactive querying, however, is *quite rare* in the presence of fixed applications at *client* sites. In practice, however, data-intensive applications make use of a fixed set of queries. These queries can still make use of parameters entered by a user and which are typically used in selection conditions. Our approach is compatible with such applications. Essentially, client applications commit a *priori* the queries they wish to execute; the owner and the publisher then pre-compute the required Merkle hash trees to produce short verification objects. While our approach cannot provide compact verification objects in the context of arbitrary interactive database querying, it is quite compatible with the widely-used practice of embedding pre-determined (and parameterizable) queries within data-intensive applications.

References

[1] M. Bellare. Practice-oriented Provable Security. In G. Davida E. Okamoto and M. Mambo (eds.), *Proceedings of First International Workshop on Information Security (ISW 97),* LNCS 1396, Springer Verlag, 1997.

[2] C.J. Date. An Introduction to Database Systems, Addison-Wesley, 1999.

[3] P. Devanbu and S.G. Stubblebine. Software Engineering for Security: a roadmap. In *The Future of Software Engineering,* Special volume published in conjunction with ICSE 2000, ACM Press, 2000.

[4] P. Devanbu, M. Gertz, C. Martel, and S.G¿ Stubblebine. Authentic Third-party Data Publication. Technical Report, `www.db.cs.ucdavis.edu/publications/DGM00.ps`, 2000.

[5] M. D. Berg , M. V. Kreveld, M. Overmars and O. Schwarzkopf. *Computational Geometry.* Springer-Verlag, New York, 2000.

[6] R.C. Merkle. A Certified Digital Signature. In *Advances in Cryptology* (*Crypto '89*), LNCS Vol. 435, Springer Verlag, 218–238, 1989.

[7] M. Naor, K. Nissim. Certificate Revocation and Certificate Update. *Proceedings of the 7th USENIX Security Symposium*, 1998.

[8] A. Silberschatz, H. Korth, S. Sudarshan. Database System Concepts, (3rd edition), McGraw-Hill, 1997.

[9] S. G. Stubblebine. Recent-secure authentication: Enforcing Revocations in distributed systems. IEEE Computer Society Symp. on Security and Privacy, 1995.

[10] J. D. Tygar Open Problems in Electronic Commerce. In *Proc. SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, ACM, 101, 1999.