Implementing a Tamper-evident Database System

Gerome Miklau[1] and Dan Suciu[2]

[1] University of Massachusetts, Amherst
miklau@cs.umass.edu
[2] University of Washington
suciu@cs.washington.edu

Abstract. Data integrity is an assurance that data has not been modified in an unknown or unauthorized manner. The goal of this paper is to allow a user to leverage a small amount of trusted client-side computation to achieve guarantees of integrity when interacting with a vulnerable or untrusted database server. To achieve this goal we describe a novel relational hash tree, designed for efficient database processing, and evaluate the performance penalty for integrity guarantees. We show that strong cryptographic guarantees of integrity can be provided in a relational database with modest overhead.

1 Introduction

Data integrity is an assurance that data has not been modified in an unknown or unauthorized manner.[3] In many settings (e.g. banking, medical information management, or scientific data) preventing unauthorized or inappropriate modification is essential.

In database systems, integrity is typically provided by user authentication and access control. Users are required to authenticate, and are limited in the operations they can perform on columns, tables, or views. Unfortunately, in real-world systems these mechanisms are not sufficient to guarantee data integrity. The integrity vulnerabilities in a modern database system stem from the complexity and variety of security-sensitive components, integration of database systems with application level code, underlying operating system vulnerabilities, and the existence of privileged parties, among others. Consequently, when a data owner uses a database to store and query data, she is forced to trust that the system was configured properly, operates properly, and that privileged parties and other users behave appropriately.

The goal of this paper is to allow a user to leverage a small amount of trusted client-side computation to achieve guarantees of integrity when interacting with a potentially vulnerable database server. We achieve this goal by adapting techniques from hash trees to a client-server relational database. In particular, we describe a novel relational hash tree, designed for efficient database processing, and evaluate the performance penalty for integrity guarantees. Our implementation does not require modification of system internals and can therefore easily be applied to any conventional DBMS. While our implementation is oriented to a relational DBMS, the problem of efficiently generating proofs of integrity has many applications to the secure management of data on the web [3, 2, 15].

Threats to database system integrity The variety of security-sensitive components and the complexity of the modern database system make it very difficult to secure in practice. Database systems figured prominently in a recent list [22] of the 20 most critical internet security vulnerabilities . We review next a range of vulnerabilities in modern database systems that could permit modification of stored data.

- *Authentication and access control* The authentication mechanisms of a database system may be vulnerable to weak passwords, or passwords stored unencrypted in scripts and programs. Access control vulnerabilities include improper enforcement by the system, improper policy specification, or configuration of privileged users as well-known system defaults.
- *Application integration* Many database systems are linked to front-end applications like web interfaces. Developers often add access control features at the application level because database systems lack fine-grained access control. This can lead to inconsistencies and new vulnerabilities. For example, *SQL injection*

[3] Throughout the paper we use this sense of the term *integrity*, appropriate to information security, as distinct from the notion of relational integrity maintained by constraints and familiar to database practitioners.

[23, 1] is an attack on databases accessible through web forms. A successful attacker can use a web form to inject SQL commands that modify data with the access rights of the web application (which are usually high) or to discover passwords. Command injection attacks including SQL injection were ranked as the top security vulnerability for web applications in [18]. SQL injection attacks have been reported for many of the most popular commercial database systems.

- *Database extensions* Almost all modern database applications include extensions in the form of user-defined functions and stored procedures. Stored procedures may be written in general-purpose programming languages, by local users or third-party vendors, and may be executed with the access rights of the calling user or in some cases the program author. Since it is very difficult to evaluate a stored procedure for proper behavior, this constitutes a serious threat.
- *Inherited OS vulnerabilities* Since relations are stored as files managed by the operating system, any threats to the integrity of the operating system and filesystem are inherited by the DBMS, and therefore constitute a threat to data integrity.
- *Privileged parties* The user of a database system is always vulnerable to the incompetence or malicious intentions of certain privileged parties. A recent survey of U.S. institutions found that about half of computer security breaches are traced to members internal to the organization [8]. Database administrators always have sufficient rights to modify data or grant permissions inappropriately, while system administrators have root access to the filesystem and could modify records stored on disk.

The techniques presented in this paper are a safeguard against each these vulnerabilities because they provide the author of data with a mechanism for detecting tampering. When verification succeeds, the author has a strong guarantee that none of the vulnerabilities above has resulted in a modification in their data. Considering the range of vulnerabilities described, it makes sense to implement the protection mechanisms described here even when the database system is controlled by a single organization or institution.

Motivating scenario Imagine Alice manages a software company and records the account information of her customers in a relation Account(acctId, name, city, current-balance). Alice would like to evaluate the following queries over Customer:

Q1 Retrieve the current balance for acctId=234.
Alice would like an assurance that the query result is *correct*: that she authorized creation of the result tuple, and that the current-balance has not been modified by an unauthorized party.

Q2 Retrieve all customers with negative balances.
Alice would like an assurance that the query result is *correct* and *complete*: that the attributes of each result tuple are authentic, and in addition, no customers with negative balances have been omitted from the query result.

Q3 Retrieve all customers located in New York City
Alice would like an assurance that the query result accurately reflects a *consistent* state of the database, meaning that tuples that were authentic with respect to a prior state of the database cannot be presented as part of the current database. Since customers may change city, this prevents mixing tuples that were correct at different points in time.

We formalize correctness, completeness, and consistency in Sec 2. To achieve these assurances of integrity, Alice will perform some computation locally using her own trusted computing resources, and store locally only a small bit string acting as a certificate of authenticity for her data. The local computation is performed by a middleware component which receives Alice's operations on her database (e.g. query, insert, update, delete), performs a straightforward translation, and submits them to the database server for processing.

The database server is largely oblivious to the fact that Alice is taking special steps to ensure the integrity of her data. The server stores a modified schema for the database which, in addition to the base data, also includes integrity metadata. The integrity metadata consists of one or more specially-designed tables each representing a hash tree [12, 13, 5]. Alice's database queries are rewritten by the middleware to retrieve the query answer along with some integrity metadata. The middleware performs an efficient verification procedure, returning to Alice the query answer along with notice of verification success or failure.

Contributions and Alternative methods The primary contribution of this work is the design, implementation and performance evaluation of hash trees for use with a client-server relational database. We describe a novel relational representation of a hash tree, along with client and server execution strategies, and show that the *cost of integrity* is modest in terms of computational overhead as well as communication overhead. Using our techniques we are able to provide strong integrity guarantees and process queries at a rate between 4 and 7 times slower than the baseline, while inserts are between 8 and 11 times slower. This constitutes a dramatic improvement over conceivable methods of insuring integrity using tuple-level digital signatures, and also a substantial improvement over naive implementations of hash trees in a database. Since our techniques can easily augment any database system, we believe these techniques could have wide application for enhancing the security of database systems.

We describe hash trees in detail in Sec. 2.2, and our novel design in Sec. 3. Before doing so, we briefly review some alternative integrity strategies and their limitations.

Authenticating tuples Digital signatures or message authentication codes at the level of tuples can prevent modification of data, but cannot resist deletion of tuples from the database or omission of tuples from query answers. Individually signed tuples may also permit inclusion of authentic but out-of-date tuples in the collection. (Using notions we will define shortly, *completeness* and *consistency* are not provided.) In addition, per-tuple signatures are likely to be prohibitively expensive.

Integrity by data duplication Using two (or more) database replicas running on different systems, executing all operations in both systems, and comparing outcomes may increase one's confidence in data integrity since an adversary would be forced to compromise both systems simultaneously. In the worst case, this technique doubles the resource costs, which may be acceptable. However, such a strategy only addresses vulnerabilities related to privileged parties, requiring collusion between parties for successful attacks, and does not guarantee integrity. The other database vulnerabilities described above (authentication weaknesses, application integration, OS vulnerabilities, etc.) will tend to hold for each system and replication does not substantially decrease the risks. Maintaining consistency between replicas is also a challenge. In general, two weakly-trusted systems do not combine to offer a fully trusted system.

Adapting B-Trees to hash trees All major database systems already contain B-tree implementations which efficiently store and update trees over sorted collections of tuples. If a strategy of server modification is followed, it could make sense to reuse this code at the server which is likely to be highly optimized. We are aware however of a number of challenges of adapting B-trees to hash trees. First, while reusing code at the server seems like an easy solution, the precise tree structure (different for each database system) needs to be reproduced at the client, and internal nodes of the tree must be returned by the server which is non-standard. In addition, transaction and locking semantics are substantially different between hash trees and B-trees. B-trees may be a viable option, and require further investigation.

Related work

Hash trees were developed by Merkle and used for efficient authentication of a public file [11, 12] as well as a digital signature construction in [13]. Merkle's hash tree can be described as an authenticated dictionary data structure, allowing efficient proofs (relative to the root hash) of membership or non-membership of elements in the set. Authenticated dictionaries were adapted and enhanced to manage certificate revocation lists in [9, 16].

Authenticated dictionaries were adapted to relations in [5, 4], where algorithms based on Merkle trees and refinements in [16] are proposed for authenticating relations and verifying basic relational queries. In [20] the authors envision authenticated B-trees and the application of commutative hash functions to improve certain performance metrics, but abandon guarantees of completeness for queries, and provide no implementation. To our knowledge, no implementation and thorough performance evaluation of B-tree based techniques has been performed.

Our work draws on the above results (providing the integrity guarantees of [5]) but offers the first design and implementation of an authenticated database using a database management system. The algorithms in existing work were described with a main memory implementation in mind: the data structures are tree-based, and the algorithms require pointer traversals and modifications that are not appropriate for a relational DBMS. Further, the algorithms are evaluated merely in terms of worst case computational and communication complexity, which can hide critical factors in the implementation. For example, we show that minimizing the number of hashes is not a major concern, but a tree organization that minimizes index lookups is critical. Although the connection is not obvious, we propose here a novel structure for a hash tree inspired by techniques for managing intervals in database systems [6, 10].

Authenticating queries using the techniques above may require revealing some data items that are not in the query answer. It is the goal of [14, 17] to provide authenticated answers while also maintaining certain secrecy properties. We focus exclusively on integrity in this work. Recently, substantially different cryptographic techniques were proposed for verifying completeness of relational queries [19]. While promising, efficient performance in practice has not yet been demonstrated.

2 Background

2.1 Threat model and formalizing integrity

We assume the data owner interacts as a client with an untrusted database server. The data owner operates a trusted computing base, but would like to store and process relational data using the untrusted resources of the server. The server may insert false tuples into relations, modify existing tuples, and remove or hide tuples from the database or from query answers. The communication medium between client and server is also untrusted. The adversary may impersonate the client and try to insert or modify tuples at the server. Alternatively, the adversary may impersonate the server and attempt to return spurious tuples as a query result. Our goal is resistance against each of these tampering threats: although we cannot prevent these actions, we can detect them with a negligible probability of failure.

We assume the client trusts the implementation of the techniques described in this work. In practice this is a reasonable assumption. Checking the correctness of our client procedure is much easier than attempting to certify the correctness of an entire database system.

We focus primarily on a single author who is the sole individual with rights to create and modify data. Many parties may issue queries, but integrity is judged with respect to the author. (We discuss extensions of our results to the multiparty case in Section 6.) To formalize integrity guarantees, we assume the author creates a relation R, stores it at the server (and not locally), and asks queries over the database. We further assume that the author updates the relation and that the sequential authentic states of the database are $R_1, R_2, \ldots R_n$. When a user asks a (monotone) query q over R, they receive ans. The key integrity properties we consider are:

Correctness Every tuple was created or modified only by the author:

$$ans \subseteq \bigcup_{1 \leq i \leq n} q(R_i)$$

The union on the right consists of all tuples present in any state of the database. Correctness asserts that the answer set is composed of only these tuples.

Completeness No qualifying tuples are omitted from the query result: for some i, $q(R_i) \subseteq ans$.

Consistency Every tuple in the database or query result is current; i.e. it is not possible for the server to present as current a tuple that has been been removed, or to mix collections of data that existed at different points in the evolution of the database. Consistency means $ans \subseteq q(R_i)$ for some i.

If the author were to sign tuples individually, verification by the client would prove correctness only. Consistency is a stronger condition which implies correctness. The hash tree techniques to be described provide consistency (and thus correctness) upon verification. In some cases hash trees can provide completeness as well. The index i, which identifies particular states of the database, corresponds to a version of the authentication tag which is changed by the author with updates to the database. Proofs of consistency or completeness are relative to a version of the authentication tag.

2.2 Enforcing integrity with hash trees

In this subsection we review the use of hash trees for authenticating relations [5, 12] by illustrating a simple hash tree built over 8 tuples from a relation $\mathsf{R(score,name)}$. We denote by f a collision-resistant cryptographic hash function (for which it is computationally-infeasible to find inputs x and x' such that $f(x) = f(x')$). We build the tree of hash values, shown in Fig. 1, as follows. First we compute the hash for each tuple t_i of the relation by hashing the concatenated byte representation of each attribute in the tuple. Then, to generate a (binary) hash tree, we pair these values, computing f on their concatenation and storing it as the parent.

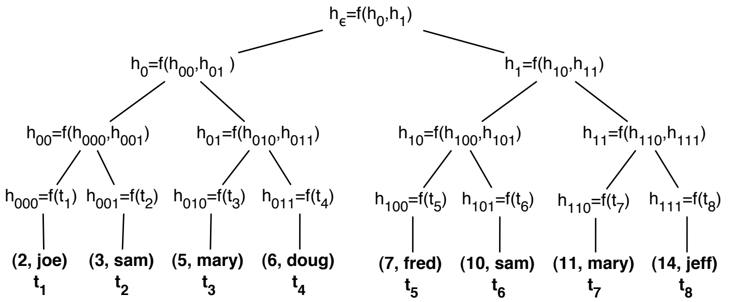

Fig. 1. Hash tree built over a relation containing tuples $t_1, t_2, ...t_8$. f is a cryptographic hash function; comma denotes concatenation when it appears in the argument of f.

We continue bottom-up, pairing values and hashing their combination until a root hash value h_ϵ is formed. The root hash value, h_ϵ, is a short sequence of bytes that depends on each tuple in the database and on a chosen order for the tuples. (The value of the root hash can therefore be used to uniquely identify states of the database, as per the definition of consistency above.)

The computation of a hash tree uses the *public* hash function f and is deterministic, so for a given tree shape it can be repeated by anyone. Alice chooses an order for the tuples in her relation, computes the root hash h_ϵ and stores it locally and securely. She will then store the relation at the vulnerable server. In Fig. 1 the hash tree is perfectly balanced, but this is not required.

Verifying query results The client verifies a query by checking that the query result is consistent with the root hash. To do so, the client must duplicate the sequence of hash computations beginning with the query result, and verify that it ends with the root hash. The tree structure allows the client to perform this computation without recovering the entire relation from the server. For a set of result tuples, the nodes in the tree the server must return are those on the *hash path*, which consist of all siblings of nodes on a path from a result tuple to the root. Successful verification proves integrity under the assumption that it is impossible for the server to find a collision in the hash function f. We illustrate with the example queries below which refer to the database R in Fig. 1 and we assume score is a key for R.

Example 1. Select tuples with score=5. The server returns the answer tuple, t_3. The client can compute h_{010}. In order to complete the computation to the root, the client needs more values from the database, or some nodes internal to the tree. The server returns in addition the hash path consisting of nodes h_{011}, h_{00} and h_1. From these values the client can compute up the tree a new root hash h'_ϵ. Verification succeeds if h'_ϵ equals the root hash h_ϵ stored at the client. Unless the server can find a collision in the hash function, this proves that tuples t_3 is an authentic element of the database, proving consistency and correctness of the query answer.

Example 2. Select tuples with score=8. Since 8 is not present in the database, the query result is empty and correctness holds trivially. To show completeness, i.e. that there are no tuples with score=8 omitted illegally, the server must return the predecessor tuple, t_5, and the successor, t_6, and the hash path $\{h_{11}, h_0\}$.

Example 3. Select tuples with *score* between 4 and 6. The server will return answer tuples t_3 and t_4 along with their hash path $\{h_{00}, h_1\}$ which allows the client to verify correctness and consistency, as above. However there could exist other tuples in the collection matching the search condition, *score* between 4 and 6. Evidence that the answer is in fact complete relies on the fact that the tree is built over sorted tuples. The server provides the next-smallest and next-largest items for the result set along with their hash path in the tree. To prove completeness, the result will consist of t_2, t_3, t_4, t_5 and the hash path is h_{000}, h_{101}, h_{11}.

Example 4. Select tuple with name='Mary'. This query is a selection condition on the B attribute, which is not used as the sort key for this hash tree. The server may return the entire result set $\{t_3, t_7\}$ along with the hash path nodes $\{h_{011}, h_{00}, h_{111}, h_{10}\}$, however in this case only consistency and correctness is proven. The server could omit a tuple, returning for example just t_3 as an answer along with it's verifiable hash path. The author will not be able to detect the omission in this case.

Modifying data: Insertions, deletions, updates The modification of any tuple in the database changes the root hash, upon which the verification procedure depends. Therefore, the client must perform re-computation of the root hash locally for any insertion, deletion, or update. We illustrate with the following example:

Example 5. Insert tuple (12, jack). This new tuple t' will be placed in sorted order between tuples t_7 and t_8. More than one tree shape is possible, but one alternative is to make t' a sibling of t_7 and set $h_{110} = f(t_7, t')$. Since the value of h_{110} has changed, the hashes on the path to the root must be updated, namely h_{11}, h_1 and h_ϵ.

It is critical that the root hash be computed and maintained by a trusted party, and retrieved securely when used during client verification. If the server could compute or update the value of the root hash, it could perform unauthorized modifications of the database without violating the verification procedure. The root hash can be stored securely by the client, or it can be stored by the server if it is digitally signed. Since the root hash changes with any modification of the database, the latter requires re-signing of the root hash with each update operation.

3 The relational hash tree

In this section we describe techniques for implementing a hash tree in a relational database system.

3.1 Overview of design choices

The simplest representation of a hash tree as a relation would represent nodes of the tree as tuples with attributes for parent, right-child, and left-child. With each tuple uniquely identified by its node id, the parent and child fields would contain node id's simulating pointers. There are a number of drawbacks of this simple organization. The first is that in order to guarantee completeness the server must return not just the result set, but the preceding and following elements in the sorted order (as in Ex. 3). Second, traversing the tree requires an iterative query procedure which progresses up the tree by following a sequence of node-ids, and gathers the nodes on the hash path. Finally, the performance of this scheme depends on the tree being balanced which must be maintained upon modification of the database. Even assuming perfect balancing, experiments indicate that the time at the server to execute a simple selection query using this organization is about 12 ms (a factor of 20 times higher than an standard query) and that query times scale linearly or worse with the size of the result.

To simplify the hash path computation, a basic optimization is to store the child hashes with their parent. Then whenever the parent is retrieved, no further database access is required for gathering the children. To simplify the identification of preceding and following elements, we translate our sorted dataset into intervals. The leaves of the hash tree are intervals, and we always search for the containing interval of a point, or the intersecting intervals of a range. This provides completeness as in Ex. 2 and 3. To remove the iterative procedure on the path to the root, we store intervals in the internal nodes of the tree representing the minimum and maximum of the interval boundaries contained in that node's descendants. The result is a relational representation of a hash tree with the only remaining challenges being (i) implementing interval queries efficiently and (ii) keeping the tree balanced.

Interval queries are not efficiently supported by standard indexes. One of the best methods for fast interval queries was presented in [21]. These techniques are a relational adaptation of an interval tree [6] and themselves require representing a tree as relations. Since any tree shape can work as a hash tree, our innovation is to adapt interval trees to design a hash tree, combining both trees into one structure. This serves dual purposes: in addition to supporting very efficient hash path queries, it also maintains a balanced tree for many data distributions.

3.2 Interval Trees

An interval tree [6, 21] is a data structure designed to store a set of intervals and efficiently support intersection queries. We review this data structure here, and then adapt it to a hash tree in the next subsection.

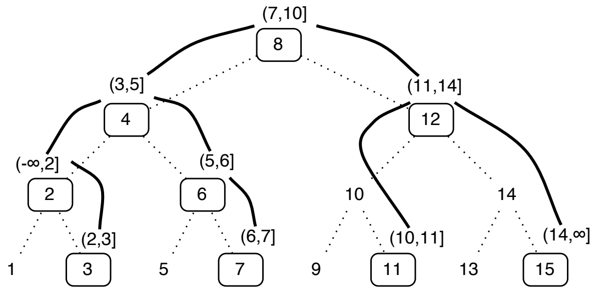

Fig. 2. The *domain tree* T_4 is the complete binary tree, rooted at 8, shown with dotted edges. The *value tree* is represented by circled nodes and solid edges, for intervals derived from $\mathbf{adom} = \{2, 3, 5, 6, 7, 10, 11, 14\}$. The intervals shown in the tree are stored in the Auth table.

The *domain tree* T_k for positive[4] domain $\mathbf{dom} = [0..2^k - 1]$ is a complete binary tree whose nodes are labeled with the elements of $\mathbf{dom} - \{0\} = (0..2^k - 1]$. Its structure is precisely that of a perfectly balanced search tree built over the entire domain. The root of T_k is the midpoint of $\mathbf{dom} - \{0\}$, 2^{k-1}, or in bits 1000...0. It's children are 01000...0 and 1100...0. The tree has height $k-1$ and its leaves are the odd elements of $\mathbf{dom}$. Domain tree T_4 is shown in Fig. 2. We often refer to the nodes of T_k by their labels in $\mathbf{dom}$.

Each node in T_k is the midpoint of its spanning interval, denoted $span(n)$, defined to be the interval containing every element in the subtree rooted at n, including itself. The span of the root is therefore $(0..2^k - 1]$; the span of node 2, shown in Fig. 2 is $(0, 3] = \{1, 2, 3\}$. For a given $\mathbf{dom}$, we denote $-\infty = 0$, and $\infty = 2^k - 1$.

An interval with endpoints in $\mathbf{dom}$ is stored at a unique node in the domain tree as defined by its *fork node*. The fork is the lowest node n such that $span(n)$ contains the interval.

Definition 1 (Fork node of interval). *Let $I = (x, y]$ be an interval with $x, y \in \mathbf{dom}$ and $x < y$. The fork node $fork(I)$ is the unique node $n \in T_k$ such that:*

(i) $I \subseteq span(n)$, and
(ii) for all descendants n' of n, $I \not\subseteq span(n')$.

It follows immediately that $n \in I$, and that n is the highest node in the domain tree such that $n \in I$.

As an example, the fork node in T_4 of interval $(3, 5]$ is 4 as shown in Fig 2. The computation of the fork for interval $I = (x, y]$ can be performed efficiently using bitwise operations on x and y. Since $x < y$, x and y can be written as $x = z0x_0$ and $y = z1y_0$ for (possibly empty) bit strings z, x_0, y_0. Then $\text{fork}(I) = z10^{|x_0|}$. If $x = 3 = 0011$ and $y = 5 = 0101$ then $z = 0$ and $\text{fork}(I) = 0100 = 4$.

3.3 Disjoint interval trees

We now adapt interval trees to our setting. Let $\mathbf{adom} = \{x_1 \ldots x_n\}$ be a set of data values from $\mathbf{dom}$. We always assume the data values are different from $-\infty$ and ∞. Then they partition $\mathbf{dom}$ into a set U of $n+1$ intervals:

$$I_0 = (-\infty, x_1], \quad I_1 = (x_1, x_2] \quad \ldots \quad I_n = (x_n, \infty]$$

for $-\infty = 0$ and $\infty = 2^k - 1$. Since the elements of $\mathbf{adom}$ are distinct, we have $x_i < x_{i+1}$ for all intervals I_i. Each interval of U is stored at its fork node in the domain tree, and we show next how the intervals can be connected to form a tree which overlays the domain tree.

We say a node w in the domain tree is *occupied* if there is an interval in U whose fork is w. The occupied nodes in Fig 2 are circled and labeled with the intervals that occupy them. Recall that the intervals of U are always non-overlapping and cover the entire domain.[5] It follows that each node in the domain tree holds

[4] For illustration purposes only, we describe a relational hash tree over a positive domain, although it is easily generalized to signed integers, floats, or strings. We implemented a signed integer domain with $k = 32$.

[5] Ours is therefore a special case of the structure considered in [6, 10] which is designed to accommodate a general set of intervals.

at most one interval. Further, the root of T_k is always occupied, and we show next that for any such U, the occupied nodes can always be linked to form a unique (possibly incomplete) binary tree, called the *value tree* for U. Let $\textsc{subtree}(x)$ in T_k be the subtree rooted at x.

Property 1. For any node x in domain tree T_k, if there is at least one occupied node in $\textsc{subtree}(x)$, then there is always a unique occupied node y which is the ancestor of all occupied nodes in $\textsc{subtree}(x)$.

For example, in Fig 2, the unique occupied ancestor in $\textsc{subtree}(10)$ is node 11. Property 1 implies that nodes 9 and 11 could never be occupied while 10 remains unoccupied.

Definition 2 (Value tree). *For a set of disjoint intervals U derived from* **adom** *as above, and domain tree T_k, the value tree V_U is a binary tree consisting of the occupied nodes from T_k and defined inductively as follows:*

- *$root(V_U) = root(T_k)$*
- *For any value node x in V_U, its right (resp. left) child is the unique occupied ancestor of the right (resp. left) subtree of x in T_k, if it exists.*

x is a leaf in V_U iff the right and left subtrees of a value node x are both unoccupied.

In Fig. 2 the value tree is illustrated with solid edges connecting the occupied nodes of T_4.
Benefits of the value tree In summary, the domain tree is static, determined by the domain, while the value tree depends on the set of values from which the intervals are derived. The value tree has a number of important properties for our implementation. First, by design, it is an efficient search structure for evaluating range intersection queries (i.e. return all stored intervals that intersect a given interval $I = (x, y]$). Such a query is evaluated by beginning at the root of the value tree, traversing the path in the value tree towards $fork(I)$ and checking for overlapping stored intervals. Secondly, it provides an organization of the data into a tree which we use as the basis of our hash tree. This avoids explicit balancing operations required by other techniques. Finally, the relationship between the domain tree and value tree allows us to avoid expensive traversals of the tree at the server. Instead of traversing the path from a node in the value tree to the root, we statically compute a *superset* of these nodes by calculating the path in the domain tree. We then probe the database for the value tree nodes that exist. The sets defined below are used for query evaluation in Sec. 3.5, and compute all the forks of nodes in the value tree necessary for client verification.

$$\mathsf{Ancestors}(x) = \{n \mid x \in span(n)\}$$
$$\mathsf{rangeAncestors}(x, y) = \{n \mid (x, y] \cap span(n) \neq \emptyset\}$$

For example, $\mathsf{Ancestors}(13) = \{8, 12, 14, 13\}$ and $\mathsf{range\text{-}Ancestors}(6{,}9) = \{8, 4, 6, 7, 12, 10, 9\}$.

Properties of a disjoint interval tree Each set **adom** determines a unique value tree. If **adom** is empty, then the resulting interval set contains only one interval $(-\infty, \infty]$, and the value tree consists of a single node: the root of T_k. At the other extreme, if **adom** = **dom** $- \{-\infty, \infty\}$ then every node of the domain tree will be occupied, and the value tree will be equal to the domain tree. We describe next some properties of the value tree which are implied by sets **adom** derived according to a known distribution.

The depth of a node $n \in T_k$ is the number of edges along the path from the root to n (hence 0 for the root). Recall that the fork node of an interval is the lowest node whose span contains the interval. Since the span of nodes in T_k decreases (by half) as you descend the domain tree, it follows that the width of an interval determines the maximum depth of the interval's fork.

Property 2 (Maximum depth of interval). Let $I = (x, x']$ be an interval and define $j = \lfloor \log_2(x' - x) \rfloor$. Then the depth of fork(i) is less than $k - j$.

This result implies that if **adom** consists of values spread uniformly, then the value tree fills T_k completely from the top down. Alternatively, if **adom** consists of consecutive values, then Prop 1 implies that the value tree fills T_k completely from the bottom up. Formally, this is captured by:

Property 3 (Value tree shape). Let **adom** be a set of $2^m - 1$ elements (for $m < k$). If (1) **adom** is spread uniformly or (2) it consists of consecutive values, then the value tree has height m.

We omit further formal analysis of the properties of the value tree for lack of space. The relevant issue for our implementation is the length of paths in the value tree. Both a uniform distribution and a set of consecutive values result in a minimal height of the value tree. The question of the worst case distribution in this setting remains open. We return to the impact of the data distribution, and domain size, k, in Sec. 5.

3.4 Relational hash tree

In this subsection we describe the relational representation of a hash tree based on interval trees. Fig. 3 contains table definitions, indexes, user-defined functions, as well as the important verification queries. Given a relation $R(A, B_1, ...B_m)$, we choose the sort attribute of the hash tree to be A and assume the domain of A can be represented using k bits. A is not necessarily a key for R, and we let $\mathbf{adom}_A$ be the set of distinct A-values occurring in R. We form disjoint intervals from this domain as described above, build a disjoint interval tree, and encode each node in the *value tree* as a tuple in table $\mathsf{Auth}_{R.A}$. The table Data_R stores each tuple of the original table R, with an added field fork which is a foreign key referencing $\mathsf{Auth}_{R.A}$.fork. We drop the subscripts for Auth and Data when the meaning is clear. For tuple $t \in$ Auth, t.fork is the fork of the interval (t.predA,t.A]. Hash values for the left child and right child of the node are stored in each tuple. In addition, the hash of the node *content* is stored in attribute hashed-content. The *content* of a value tree node is a serialization of the pair (pred,A), concatenated with a serialized representation of all tuples in Data agreeing on attribute A, sorted on a key for the remaining attributes. The hash value at any node in the value tree is computed as f(Lhash, hashed-content, Rhash). For internal nodes of the value tree, Lhash and Rhash are hash values of right and left children of the node. For leaf nodes, Lhash and Rhash are set to a public initialization value. All hash values are 20 bytes, the output of the SHA-1 hash function. Note that in a conventional hash tree data values only occur at the leaves while in our interval hash tree, data values are represented in all nodes.

TABLES
Auth_R(fork bit(k), predA bit(k), A bit(k),
  Lhash byte(20), hashed-content byte(20),
  Rhash byte(20))
Data_R(fork bit(k), A bit(k), B_1 ... B_m)

INDEXES
Auth-index CLUSTERED INDEX on (Auth.fork)
Data-index INDEX on (Data.fork)
⟨additional user indexes on Data not shown⟩

FUNCTIONS
Ancestors(x bit(x)) ⟨defined in Sec. 3.3⟩
rangeAncestors(x bit(x),y bit(x))

(Q1) Selection query on A: R.A = $x

```
SELECT Auth.*
FROM Ancestors($x) as F, Auth
WHERE F.fork = Auth.fork    ORDER BY F.fork
```

(Q2) Range query on A: $x < R.A ≤ $y

```
SELECT Auth.*
FROM rangeAncestors($x,$y) as F, Auth
WHERE F.fork = Auth.fork    ORDER BY F.fork
```

(Q3) Arbitrary query condition: cond(R.A, R.B1 .. R.Bm)

```
SELECT Auth.*
FROM Auth, Data, Ancestors(Data.A) as F
WHERE F.fork = Auth.fork AND
cond(Data.A,Data.B1 .. Data.Bm)    ORDER BY F.fork
```

Fig. 3. Table definitions, indexes, functions, and queries for relational hash tree implementation.

3.5 Authenticated query processing

The client-server protocol for authenticated query and update processing is illustrated in Fig. 4. We describe next server query processing, client verification, and updates to the database.

Server query processing The query expressions executed at the server are shown in Fig. 3. For selection and range queries on the sort attribute, $Q1$ and $Q2$, and arbitrary query conditions, $Q3$. They each retrieve from Auth the result tuples along with paths to the root in the value tree. We avoid iterative traversal in the value tree by computing the sets Ancestors(x) or rangeAncestors(x, y) and performing a semijoin. Note that the computation of the ancestors makes no database accesses. It is performed efficiently as an user-defined procedure returning a unary table consisting of nodes from the domain tree. The following examples illustrate the execution of $Q1$ and $Q2$.

Example 6. SELECT * FROM R WHERE R.A = 13 Referring to Fig. 2, we compute Ancestors(13) which is equal to $\{8, 12, 14, 13\}$. $Q1$ joins these nodes with the value tree nodes in Auth_R. The result is just two tuples representing nodes 12 and 8 in the value tree. Node 12 holds interval $(11, 14]$ which contains the search key 13, proving the answer empty but complete, and node 8 is included since it is on the path to the root.

Example 7. SELECT * FROM R WHERE 6 < A ≤ 9 Computing rangeAncestors$(6, 9)$ yields the set $\{8, 4, 6, 7, 12, 10, 9\}$ since each of these nodes in the domain tree has a span intersecting $(6, 9]$. Of these, only nodes $\{8, 4, 6, 7, 12\}$ are in the value tree, and are retrieved from the database. Note that some intervals stored at these nodes do not overlap the query range $(6, 9]$ (for example, $(3, 5]$ is retrieved with node 4). Nevertheless, node 4 is required for reconstructing this portion of the value tree since nodes 6 and 7 are its descendants. The client will perform a final elimination step in the process of verifying the answer.

Arbitrary queries For queries that include conditions on attributes $B_1, ...B_m$ the interval hash tree cannot be used to prove completeness of the query answer, but can still be used to prove correctness and consistency. Any complex condition on $B_1, ...B_m$ can be evaluated on Data, resulting in a set of values for attribute A. These values will be distributed arbitrarily across the domain and therefore arbitrarily in the value tree. They are fed into the Ancestors function to retrieve all paths up to the root. Duplicates are eliminated, and then these nodes are joined with Auth (as in queries $Q1$ and $Q2$). The resulting query is shown in Fig. 3 as $Q3$.

Example 8. The query in Ex. 4 asked for all tuples from R with *name*='Mary'. The query result consists of tuples with *scores* 5 and 11. To authenticate this query, the fork ancestor set is computed to be Ancestors$(5) \cup$ Ancestors$(11) = \{8, 4, 6, 5\} \cup \{8, 12, 10, 11\} = \{8, 4, 6, 5,\ 12, 10, 11\}$.

Execution plan For queries Q_1, Q_2 and Q_3 the favored execution strategy is to materialize the Ancestors table, and perform an index-nested loops join using the index on Auth_R. The query optimizers in the two database systems we tried chose this execution plan. For Q_1, the Ancestors set is quite small – it is bounded by parameter k of the domain tree (32 for most of our experiments). Thus evaluation of Q_1 consists of not more than k probes of the index on Auth. For range queries, the Ancestors set is bounded by the result size plus $2k$. The execution of range queries can be improved by utilizing the clustering of the index on Auth. This optimization is described in Sec. 4.3.

Client verification In each case above, the server returns a subset of tuples from Auth which represent a portion of the value tree. The client verifies the query answer by reassembling the value tree and recomputing hashes up the tree until a root hash h'_ϵ is computed. To enable the client to efficiently rebuild the tree, the result tuples can be sorted by the server.

Insertion, deletion, and update Updating any tuple in the database requires maintenance of the interval hash tree, namely addition or deletion of nodes in the value tree, and re-computation of hashes along the path to the root from any modified node. These maintenance operations are integrity-sensitive, and can only be performed by the client. Therefore, before issuing an update, the client must issue a query to retrieve the relevant portions of the hash tree, verify authenticity, and then compute the inserts or updates to the database. The insertion protocol between client and server is illustrated in Fig. 4. We focus for simplicity on inserts (deletes are similar, and updates are implemented as deletes followed by insertions).

Example 9. To insert a new tuple (13, ..) the client issues the selection query for $A = 13$. This is precisely the query described in Ex. 6, and retrieves nodes 8 and 12 from the value tree. Node 12 contains interval $(11, 14]$ which must be split, with the insertion of 13, into two intervals: $(11, 13]$ and $(13, 14]$. This requires an update of the tuple representing value tree node 12, changing its interval upper bound from 14 to 13. Interval $(13, 14]$ will be stored at the formerly unoccupied node 14, which requires insertion of a new tuple in Auth representing the new value node. The hashes are recomputed up the tree from the lowest modified node in the value tree. In summary, this requires 1 insert into Auth, and a sequence of updates to the value tree tuples along the path from the inserted tuple to the root.

In general, executing an authenticated insertion involves the cost of an authenticated query, the insertion of one new Auth tuple, and updates to h Auth tuples, where h is the depth of the fork of the new interval created by the insertion. Although each update to Auth can be executed efficiently using the index on node, the large number of updates can cause a severe penalty. We address this problem next by bundling the nodes of the value tree.

4 Optimizations

4.1 Bundling nodes of the value tree

The execution cost of queries and inserts depends on the length of paths in the value tree, which is determined by the data distribution and bounded by the maximum depth in the tree k. Reducing the length of paths in the value tree reduces the number of index probes executed for queries, and reduces the number of tuples modified for insertions. To reduce path length, we propose grouping nodes of the value tree into bundles, and storing the bundles as tuples in the database. For example, we can imagine merging the top three nodes $(8, 4, 12)$ in the tree of Fig. 2 into one node. We measure the degree of bundling by the height of bundles, where a bundle of height b can hold at most $2^b - 1$ value tree nodes ($b = 1$ is no bundling). The schema of the Auth table is modified so that a single tuple can hold the intervals and hashes for $2^b - 1$ nodes in the value tree. and the Ancestors and rangeAncestors functions are generalized to account for bundling.

4.2 Inlining the fork ancestors

Although the Ancestors set is efficiently computed for selection queries on the sort attribute, our experiments show that when evaluating arbitrary query conditions that return a large number of distinct A-values, computation of the Ancestors set can have a prohibitive cost. To remedy this, we have proposed trading off space for computation, and inlining the ancestor values for each tuple in Data. This requires that the schema of the Data table be modified to accommodate $\lceil k/b \rceil$ fork nodes, where k is the parameter of the domain tree and b is the bundling factor.

4.3 Optimizing range queries

The evaluation of range queries can be substantially improved by expressing rangeAncestors(x, y) as the disjoint union of three sets: leftAncestors, Inner, and rightAncestors. Inner(x, y) consists of all fork nodes inside $(x, y]$. leftAncestors(x, y) consists of all fork nodes not in Inner whose span upper boundary intersects $(x, y]$. Likewise, rightAncestors(x, y) contains all nodes not in Inner whose span lower boundary intersects $(x, y]$. The range query $Q2$ is then equivalent to the following union of three subqueries:

```
SELECT * FROM leftAncestors(x,y) as L, Auth
WHERE L.fork = Auth.fork    UNION ALL
SELECT * FROM rightAncestors(x,y) as R, Auth
WHERE R.fork = Auth.fork    UNION ALL
SELECT * FROM Auth
WHERE Auth.fork BETWEEN x AND y
```

The benefit is that the range query in the third SELECT can be evaluated using the clustered index on fork. This optimization is used in [10].

5 Performance evaluation

In this section we present a thorough performance evaluation of the relational hash tree and our proposed optimizations. The client authentication code was written in Java, using JDBC to connect to the database server. Experiments were performed using both PostgreSQL, and Microsoft SQL Server databases.[6] No substantial differences were found between engines, and for each experiment we present numbers for only

[6] Moving between database systems was very easy; the only challenge was adapting to different SQL dialects.

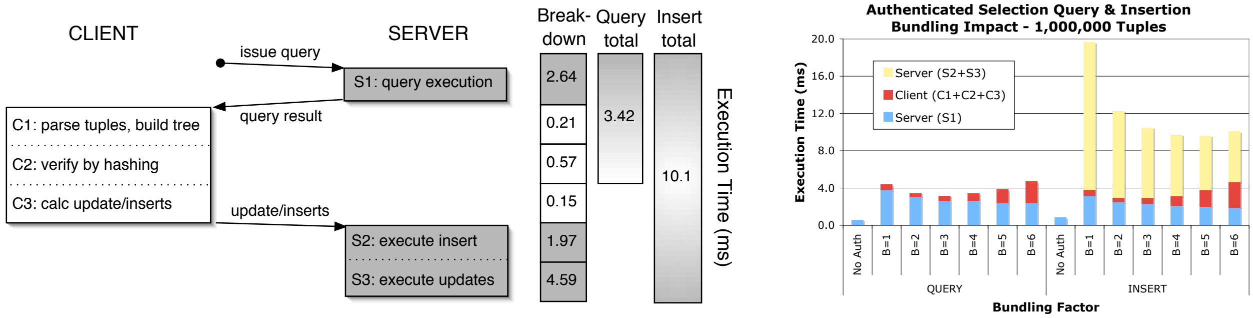

Fig. 4. (Left) Processing diagram for authenticated QUERY and INSERT. C_1, C_2, C_3 are client-side computations, S_1, S_2, S_3 are server-side computations (including communication cost), with execution times broken down by component cost (for bundling = 4). (Right) Impact of bundling on execution times for authenticated query and insert.

one system. Both the client and server machines were Pentium 4, 2.8Ghz machines with 2 GB memory. We used SHA-1 [7] as our hash function. SHA-1 is no longer considered secure. It was broken after these experiments were performed, but moving to SHA-224 is expected to have a negligible impact on the numbers presented here. The performance numbers below do not include the invariant cost of signing the root hash upon update (4.2 ms), and verifying the root hash for queries (0.2 ms). The numbers below represent the average execution time for 200 random query or insert operations, with the 5 lowest and highest values omitted, over a database of random values.

Overview of cost for selection query and insert Our experiments were run on synthetic datasets containing 200-byte tuples. On a database of 1,000,000 tuples, without authentication, a selection query on an indexed attribute takes approximately 0.6ms while an insert takes about 1.0ms. These are the baseline values used for comparison in our analysis.

Figure 4 shows the processing protocol for a simple selection query and an insert, along with times for our best-performing method. An authenticated query consists of execution of the hash path query at the server (quantity S_1) followed by client-side parsing the result tuples and re-building tree structure (C_1) and verification by hashing up the tree and comparing the root hash (C_2). The total time is 3.42ms, of which 77% is server computation and 23% is client time. This verified query therefore executes about 6 times slower than the baseline. Communication costs are included in the server-side costs for simplicity.

Execution of an authenticated insert includes the costs of the query *plus* the additional client computation of the updates and inserts to the relational hash tree (quantity C_3) and server execution of updates and inserts (S_2 and S_3). Overall, authenticated inserts run about 10 times slower than the baseline. The dominant cost is the execution of updates. Our bundling optimization targets this cost, bringing it down from $14ms$ to the value in Fig. 4 of 4.59ms (for bundle height 4). The cost is significant because for a relational hash tree of average height h, approximately h tuples in the Auth table must be updated since the hash values of the nodes have changed. This cost is targeted by our bundling optimization described next.

Impact of domain tree bundling Recall that our bundling optimization was defined in terms of a bundle height parameter b, the tree height of bundles, which is 1 for no bundling. Fig. 4 (right) shows the impact of bundling on authenticated selection queries and inserts. Each bar in the graph also indicates the breakdown between client and server computation. Bundling primarily speeds up server operations by reducing the number of tuples retrieved and/or updated. The impact of bundling is dramatic for inserts, where the time for $b = 4$ is about half the time for $b = 1$. This is a consequence of fewer updates to the bundled value tree nodes in the Auth table (5 instead of about 16 without bundling).

Range and Arbitrary queries The bundle height $b = 4$ is optimal not only for inserts (shown above) but for range and arbitrary queries studied next, and all results below use the bundling technique. Range queries (using the optimization described in Sec. 4.3) run very efficiently, just 2-3 times slower than the baseline case. Arbitrary queries are slower because disparate parts of the tree must be retrieved, and each node retrieved requires and index probe. They run about 20-30 times slower than the baseline, and scale roughly linearly with the result size. Inlining is a critical optimization, as shown in Fig. 5 (left) which improved processing time by about a factor of 5 in our experiments. The figure shows the per-tuple speed-up for processing an arbitrary query. That is, a value of 1 indicates that an arbitrary query returning 1000 tuples takes the same time as 1000 separate selection queries returning 1 tuple.

Scalability Fig. 5 (right) shows that our techniques scale nicely as the size of the database grows. The table presents selection queries, range queries, and inserts for databases consisting of 10^5, 10^6, and 10^7 tuples. Authenticated operations are roughly constant as the database size grows. This is to be expected since at the client the execution time is determined by the result size, and at the server the execution time is largely determined by the length of paths in the relational hash tree which grow logarithmically with the database size.

Impact of domain size and data distribution Recall that parameter k of the domain tree is determined by the number of bits required to represent elements in the domain of the sort attribute A, and was set to 32 in most of our experiments. Since k is the height of the domain tree, k bounds the length of paths in the value tree, and determines the size of the sets returned by Ancestors and rangeAncestors. This means that as k increases, the number of index probes increases with the average depth of nodes in the domain tree. This increases the server query time, but is mitigated by the bundling optimization. Further, the more important factor is the length of paths in the value tree because this determines the number of nodes returned by the server, as well as the number of updates/inserts to the database when hashes are recomputed upon update. The length of paths in the value tree is determined by the database instance size and distribution of values across the sort attribute. Prop. 3 proves an upper bound on the height for two cases. Our experiments confirmed that the value tree is well-balanced and these paths are short. For example, there is no measurable difference in query execution times between a uniform distribution, a database of consecutive values, or distributions derived from compound sort keys. Thus the shape of the interval tree is a very effective alternative to explicit balancing of the hash tree stored at the server.

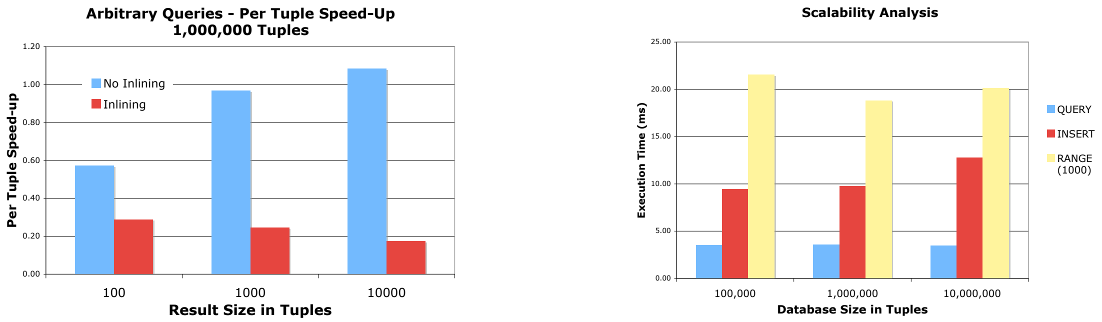

Fig. 5. (Left) Per-tuple speed up for arbitrary condition queries using inlining. (Right) Scalability graph showing execution time for selection queries, inserts, range queries for databases with size $10^5, 10^6$ and 10^7.

Storage and Communication overhead Communication cost is included with the server-side query execution costs in all performance numbers above, and was not substantially increased by our techniques. Storage overhead was also modest: total size on disk (including indexes) for the authenticated database was 321MB compared to 249MB for the baseline.

We consider the costs presented modest, especially since the overhead of our techniques is measured by comparison to some of the fastest operations that can be performed in a database system (index lookups and updates). In a real application, the overhead of integrity could easily be dwarfed by a query that included even a simple table scan.

6 Multiple party integrity

Conventional notions of authenticity typically refer to a single author. There are therefore some basic underlying challenges to formalizing authenticity of data modified by many parties. A straightforward extension of the single party case we have studied so far permits multiple authors, who all trust one another, but do not trust the database server. In this case any author is permitted to update the root hash, and the authors use a shared signature to prevent the server from modifying the root hash. Such a scenario presents some challenges for concurrent processing of updates because there is contention for the root hash value.

Nevertheless, a more realistic model for multiple party integrity has n mutually untrusting authors contributing to a common database which is stored at an untrusted server. In this setting we partition the tuples of the database by author, and ownership of a tuple is indicated by the presence of an author field in our modified schema: R'(author, A, B1 ... Bm). A query over R can be evaluated over R' and the same integrity properties of correctness, consistency, and completeness are relevant in this setting. However, it should be impossible for author α_1 to add, modify, or delete tuples in the name of any other author. Our techniques can be used to provide these integrity guarantees by prepending the author field to whichever sort key is used for the authentication table Auth, and essentially maintaining separate hash trees for each author. This relieves contention for the root hash, and improves concurrency. This model can be extended to support transfer of ownership amongst users, but we leave this as future work.

7 Conclusion

We have described techniques that allow a client – with a modest amount of trusted computational overhead, and a reasonable cost at the server – to verify the correctness, consistency, and completeness of simple queries, even if the database system is vulnerable to tampering. Our work is the first to design and evaluate techniques for hash trees that are appropriate to a relational database, and the first to prove the feasibility of hash trees for integrity of databases. Our implementation shows that some of the metrics optimized in other works are inappropriate and will not lead to better performance in database system. In the future we would like to investigate probabilistic (rather than absolute) guarantees, and additional models of multiple party integrity.

References

1. C. Anley. Advanced SQL injection in SQL server applications. NGSSoftware Insight Security, available from www.ngssoftware.com, 2002.
2. E. Bertino, G. Mella, G. Correndo, and E. Ferrari. An infrastructure for managing secure update operations on xml data. In *Symposium on Access control models and technologies*, pages 110–122. ACM Press, 2003.
3. P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. Flexible authentication of XML documents. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 136–145. ACM Press, 2001.
4. P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic data publication over the internet. *J. of Computer Security*, 11(3):291–314, 2003.
5. P. T. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic third-party data publication. In *IFIP Work. on Database Security*, 2000.
6. H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical report, Technical University of Graz, Austria, 1980.
7. Secure hash standard (SHA). Federal Information Processing Standard Publication 180-2, 2000.
8. L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. 2004 CSI/FBI computer crime and security survey. Computer Security Institute, 2004.
9. P. C. Kocher. On certificate revocation and validation. In *Fin. Cryptography*, pages 172–177, 1998.
10. H.-P. Kriegel, M. Potke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *VLDB Conference*, pages 407–418, 2000.
11. R. C. Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Information Systems Laboratory, Stanford University, 1979.
12. R. C. Merkle. Protocols for public key cryptosystems. In *Symp. Security & Privacy*, 1980.
13. R. C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.
14. S. Micali, M. O. Rabin, and J. Kilian. Zero-knowledge sets. In *FOCS*, 2003.
15. G. Miklau and D. Suciu. Managing integrity for data exchanged on the web. In A. Doan, F. Neven, R. McCann, and G. J. Bex, editors, *WebDB*, pages 13–18, 2005.
16. M. Naor and K. Nissim. Certificate revocation and certificate update. In *USENIX Security Symp.*, 1998.
17. R. Ostrovsky, C. Rackoff, and A. Smith. Efficient consistency proofs on a committed database.
18. The 10 most critical web application security vulnerabilities. OWASP http://aspectsecurity.com/topten/, Jan 2004.
19. H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD Conference*, pages 407–418, 2005.

20. H. Pang and K.-L. Tan. Authenticating query results in edge computing. In *ICDE*, 2004.
21. F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, New York, NY, 1985.
22. The 20 most critical internet security vulnerabilities. SANS Inst. http://www.sans.org/top20/, Oct 2004.
23. SQL injection: Are your web applications vulnerable? SPI Dynamics Inc. White Paper, Retrieved Oct 1, 2004 from: www.spidynamics.com, 2002.