Efficient Consistency Proofs for Generalized Queries on a Committed Database*

Rafail Ostrovsky
UCLA
rafail@cs.ucla.edu

Charles Rackoff
University of Toronto
rackoff@cs.toronto.edu

Adam Smith
MIT
asmith@csail.mit.edu

July 20, 2004

Abstract

A *consistent query protocol* (CQP) allows a database owner to publish a very short string c which *commits* her and everybody else to a particular database D, so that any copy of the database can later be used to answer queries and give short proofs that the answers are consistent with the commitment c. Here *commits* means that there is at most one database D that anybody can find (in polynomial time) which is consistent with c. (Unlike in some previous work, this strong guarantee holds even for owners who try to cheat while creating c.) Efficient CQPs for membership and one-dimensional range queries are known [5, 17, 22]: given a query pair $a, b \in \mathbb{R}$, the server answers with all the keys in the database which lie in the interval $[a, b]$ and a proof that the answer is correct.

This paper explores CQPs for more general types of databases. We put forward a general technique for constructing CQPs for any type of query, assuming the existence of a data structure/algorithm with certain inherent robustness properties that we define (called a *data robust algorithm*). We illustrate our technique by constructing an efficient protocol for *orthogonal range queries*, where the database keys are points in $\mathbb{R}^d$ and a query asks for all keys in a rectangle $[a_1, b_1] \times \ldots \times [a_d, b_d]$. Our data-robust algorithm is within a $O(\log N)$ factor of the best known standard data structure (a range tree, due to Bentley [2]).

We modify our protocol so that it is also *private*, that is, the proofs leak no information about the database beyond the query answers. We show a generic modification to ensure privacy based on zero-knowledge proofs, and also give a new, more efficient protocol tailored to hash trees.

Keywords: Commitment, zero-knowledge sets, authenticated data structures, range queries, database privacy.

*Preliminary work done during the summer of 2000 when all authors were visiting/working at Telcordia Technologies. Preliminary version appeared as MIT LCS Technical Report TR-887, Feb. 2003 [27]. Work of the first author at UCLA is partially supported by a gift from Teradata. Conference version in *ICALP 2004*.

Contents

1 Introduction

Informally, a *consistent query* protocol (CQP) allows a database owner to publish a short string c which *commits* her to a particular database D, so that she can later answer queries and give short proofs that her answers are consistent with D. Here *commits* means that she cannot change her mind about D — there is at most one database she can find (in polynomial time) which is consistent with c (e.g. c could be a secure hash of D). Similarly, she can only find valid proofs for query answers which are consistent with D. The challenge is to make both the commitment and the proofs of consistency as short and simple as possible.

One may also require *privacy* – that is, the proofs of consistency should not leak any information on the database beyond the query answers. Privacy is important, for example, in settings in which query answers are sold individually, or in which the database contains personal data. Adding this requirement to a CQP brings it much closer to the traditional cryptographic notion of a commitment scheme.

Below, we discuss relevant related work and then describe our results in detail.

1.1 Related Work

We discuss the related work in the context of cryptographic commitment protocols. These have been studied extensively, and part of our contribution is to tie them in to an algorithmic point of view. A commitment protocol allows Alice to put a value a in a virtual envelope and hand it to Bob. Bob learns nothing about the value (*hiding*), but Alice can later open the envelope, without being able to reveal a different value a' (*binding*).

Commitment Schemes for Large Datasets. The notion of commitment has been generalized considerably to allow revealing only partial information about the committed data, using very little communication. Merkle [24] proposed the following protocol for committing to a list of N values $a_1, ..., a_N$: Pick a collision-resistant hash-function[1] H (say from $2k$ bits to k bits), pair up inputs $(a_1, a_2), \ldots, (a_{N-1}, a_N)$ and apply H to each pair. Now, pair up the resulting hash values and repeat this process, constructing a binary tree of hash values, until you get to a single root of length k. If the root of the tree is published (or sent to Bob by Alice), the entire collection of values is now committed to, though not necessarily hidden—we discuss hiding further below. To reveal any particular value a_i, Alice can reveal a path from the root to a_i together with all the siblings along the path. This requires only $k \log N$ bits. This idea has many cryptographic applications, including efficient signature schemes [24, 6], efficient zero-knowledge arguments [16, 1] and computationally sound proofs [21].

Recently Buldas, Laud and Lipmaa [4], Kilian [17] and Micali and Rabin [22] independently generalized this idea to allow committing to a *set* of values. The server produces a short commitment to her set of $(\mathsf{key}, \mathsf{value})$ pairs which is made public. When a client makes a *membership query* (i.e. "do you have an entry with key x?"), the server returns the answer along with a short proof of consistency. (We call a scheme for this task a CQP for membership queries.) A very similar data structure (again, a Merkle tree) also allows one to also answer one-dimensional *range queries*, e.g. "What keys lie between x and y?" [5, 17, 22]. Merkle trees were subsequently modified to allow efficient updates by changing the structure to resemble a skip list [18]. Our work generalizes these ideas to more complex queries and data structures, and provides rigorous proofs of security.

Protocols with a Trusted Committer—Authenticated Data Structures. There is substantial work on *authenticated data structures* [25], which allow one to guarantee the consistency of many replicated copies of a database. That work tackles a different problem from ours, since it assumes that the commitment phase is always performed honestly. The extra assumption is appropriate in many situations (e.g. certificate revocation with a trusted certification authority[2]) and seems to allow greater efficiency: there are authenticated data structure protocols for answering range queries in time $O(N \log^{d-1} N)$ [13], as opposed to the $O(N \log^d N)$ time taken by the protocols of this paper. Indeed, our generic construction can be viewed as a more robust, but possibly less efficient, version of the generic constructions of authenticated data structures [25, 19, 8, 13].

Despite the greater efficiency it affords, the assumption of a trusted committer is problematic. As argued in [4], a dishonest certification authority could not easily be taken to task for providing inconsistent answers to revocation queries. There are other reasons to distrust the party generating the commitment. With a pricing database, one may want guarantees against price discrimination by the database owner. Peer-to-peer systems, where no individual processor in the network is fully trusted, provide further motivating examples [18].

[1] A hash function family $H_\kappa(\cdot)$ is *collision-resistant* if no poly-time algorithm given κ can find a pair of inputs that map to the same output for a randomly chosen key κ (see Section 2).

[2] More generally, cryptographic commitment schemes with an honest committer have been useful in a variety of contexts, from simplifying requirements on hash functions for signature schemes [26] to combined signature and encryption protocols [9].

Privacy for Committed Databases—Zero-Knowledge Sets. Micali, Rabin and Kilian [20] show how to prove consistency of answers to membership queries while also hiding information about unanswered queries. They require that consistency proofs leak nothing about the database except the query answer—not even the size of the database. (They call the primitive a *zero-knowledge set*.) They give an efficient protocol based on the DDH assumption, with proof length $O(k \log M)$ where M is an upper bound on the set size (k is the output length of the hash function). We show how to achieve the same result with $poly(k)$ communication, under more general assumptions and for more general types of queries. Subsequent to our work, [15] achieved the results of [20] based on general assumptions.

1.2 Our Contributions

This paper considers CQPs for types of queries beyond simple membership and range queries. We give a general framework for designing such protocols based on query algorithms with a certain robustness property, and illustrate our paradigm for *orthogonal range queries*, constructing protocols with an $O(k \log N)$ overhead over the fastest known standard query alogrithms. We also show how to make the protocols *private* without too much loss of efficiency.

A general paradigm for CQPs. We introduce *data-robust algorithms* (DRAs). These are search algorithms (paired with data structures) which are robust against corruptions of the data by an unbounded, *malicious* adversary: for any input—essentially, an arbitrary string— the algorithm will answer all queries consistently with one (valid) database. Although this is trivial for data structures which incorporate no redundancy, it becomes more challenging for more complex structures. (We do not want the algorithm to have to scan the entire data structure each time it is run.)

Assuming the existence of collision-resistant hash functions, any DRA which accesses memory via pointers can be transformed into a consistent query protocol whose (non-interactive) consistency proofs have length at most $O(kT)$, where k is the output size of the hash function and T is the running time of the DRA.

DRAs provide a connection between special data structures and cryptographic protocols. A previous connection was given by a Micciancio [23], for data structures which forget the order in which a sequence of updates was performed. This property is quite different from the one we require.

CQP for Orthogonal Range Queries. We present a consistent query protocol scheme that allows efficient orthogonal range queries in d dimensions. That is, the database consists of tuples $(\mathsf{key}_1, ..., \mathsf{key}_d, \mathsf{value})$, a query consists of d intervals $[a_1, b_1], \ldots, [a_d, b_d]$, and an answer is the set of all database elements whose keys lie inside the corresponding hypercube. The server not only proves that it has provided all the points in the database which match the query, but also that no others exist.

Our consistency proofs have size $O(k(m+1) \log^d N)$, where N is the database size, k is the security parameter, and m is the number of keys in the database satisfying the query (the computation required is $O((m+1) \log^d N)$ hash evaluations). For range queries on a single key, our construction reduces essentially to that of [5, 22, 17].

Our protocol is obtained by first constructing a DRA based on range trees, a classic data structure due to Bentley [2]. Existing algorithms (in particular, the authenticated data structures of [19]) do not suffice, as inconsistencies in the data structure can lead to inconsistent query answers. Instead, we show how local checks can be used to ensure that all queries are answered consistently with a single database. For d-dimensional queries, the query time is $O((m+1) \log^d N)$, where m is the number of hits for the query and N is the number of keys in the database. This is within $\log N$ of the best known (non-robust) data structure.

Privacy for Consistent Query Protocols—Generic Techniques. Consistent query protocols will, in general, leak information about the database beyond the answer to the query. It is possible to add privacy to any CQP using generic techniques: one can replace the proof of consistency π with a zero-knowledge proof of knowledge of π. Surprisingly, this leads to schemes with good asymptotic communication complexity, namely $O(poly(k))$. This generic transformation can hide the size of the database, as in [20]. In particular, this means that one can build *zero-knowledge set* protocols [20] (i.e. CQP's for membership queries) based on general assumptions: for interactive protocols, it is sufficient to assume one-way functions and for non-interactive protocols, it is sufficient to assume that non-interactive zero-knowledge proof systems exist.

Privacy for Consistent Query Protocols—Efficient Constructions. The generic constructions just mentioned are ungainly—the use of NP reductions and probabilistically checkable proofs means that the advantages only appear for extremely large datasets. We give a simpler zero-knowledge protocol tailored to Merkle trees, which does not hide the size of the database. The crux of that protocol is to avoid NP reductions when proving zero-knowledge statements about values of the hash function, and so the result is called an *explicit-hash Merkle tree*. As a sample application, we show how this protocol can be used to add privacy to one-dimensional range trees.

Organization. Section 2 formally defines CQPs. Section 3 explains data-robust algorithms, and the transformation from DRAs to CQPs. Section 4 gives our DRA for orthogonal range queries. Section 5 discusses techniques for making CQPs private. Due to lack of space, all proofs are deferred to the full version.

2 Definitions

We denote by $y \leftarrow A(x)$ the assignment of the (possibly randomized) output of algorithm A on input x to variable y. A function $f(k)$ is *negligible* in a parameter k if $f(k) \in O(\frac{1}{k^c})$ for all integers $c > 0$.

A major component in all our constructions is a collision-resistant hash function family (CRHF). This is a family of length-reducing functions (say from $3k$ bits to k bits) such that it is computationally infeasible to find a collision, i.e. $x \neq y$ with $h(x) = h(y)$ for a random member h of the family. Such functions can be constructed assuming the hardness of the discrete logarithm or factoring. Formally, a family of functions $\{h_{s,k} : \{0,1\}^* \to \{0,1\}^k\}$ is a CRHF if the functions $h_{s,k}$ can be evaluated in time polynomial in k, and there is a probabilistic polynomial time (PPT) key generation algorithm Σ such that for all polynomial-size, randomized circuit families $\{A_k\}$, the quantity $\Pr[s \leftarrow \Sigma(1^k); (x,y) \leftarrow \mathcal{A}_k(1^k, s) :\ h_{s,k}(x) = h_{s,k}(y)]$ is negligible in k.

2.1 Consistent Query Protocols

A query structure is a triple $(\mathcal{D}, \mathcal{Q}, Q)$ where $\mathcal{D}$ is a set of *valid* databases, $\mathcal{Q}$ is a set of possible queries, and Q is a rule which associates an answer $a_{q,D} = Q(q, D)$ with every query/database pair $q \in \mathcal{Q}, D \in \mathcal{D}$.

In a CQP, there is a server who, given a database, produces a commitment which is made public. Clients then send queries to the server, who provides the query answer along with a proof of consistency of the commitment. There may also be a public random string to be provided by a trusted third party. In most of our protocols, the third party is only required for choosing the collision-resistant hash function (and even this can be done by the client if he is available before the database commitment is generated).

Definition 1. A (non-interactive) *query protocol* consists of three probabilistic polynomial-time (PPT) algorithms: a server setup algorithm $\mathcal{S}_s$, an answering algorithm for the server $\mathcal{S}_a$, and a client $\mathcal{C}$. In some settings, there may also be an efficient algorithm Σ for sampling any required public randomness.

- The setup algorithm $\mathcal{S}_s$ takes as input a valid database D, a value 1^k describing the security parameter, and the public information $\sigma \leftarrow \Sigma(1^k)$. It produces a (public) commitment c and some internal state information *state*. Subsequently, $\mathcal{S}_a$ may be invoked with a query $q \in \mathcal{Q}$ and the setup information *state* as input. The corresponding output is an answer/proof pair (a, π), where $a = Q(q, D)$.

- The client $\mathcal{C}$ receives as input the unary security parameter 1^k, the public string σ, the commitment c, a query q and an answer/proof pair (a, π). $\mathcal{C}$ "accepts" or "rejects" the proof π.

Definition 2. A query protocol is *consistent* if it is complete and sound:

- Completeness: For every valid database D and query q, if setup is performed correctly then with overwhelming probability, $\mathcal{S}_a$ outputs both the correct answer and a proof which is accepted by $\mathcal{C}$. Formally, for all $q \in \mathcal{Q}$ and for all $D \in \mathcal{D}$,

$$\Pr[\sigma \leftarrow \Sigma(1^k); (c, \mathit{state}) \leftarrow \mathcal{S}_s(\sigma, D); (a, \pi) \leftarrow \mathcal{S}_a(q, \mathit{state}) : \\ \mathcal{C}(\sigma, c, q, a, \pi) = \text{"accept" and } a = Q(q, D)] \ \geq \ 1 - negl(k)$$

- (Computational) Soundness: For every non-uniform PPT adversary [3] $\tilde{\mathcal{S}}$: run $\tilde{\mathcal{S}}$ to obtain a commitment c along with a list of triples (q_i, a_i, π_i). We say $\tilde{\mathcal{S}}$ *acts consistently* if there exists $D \in \mathcal{D}$ such that $a_i = Q(q_i, D)$ for all i for which π_i is a valid proof. The protocol is *sound* if all PPT adversaries $\tilde{\mathcal{S}}$ act consistently. Formally:

$$\Pr[\sigma \leftarrow \Sigma(1^k); \big(c, (q_1, a_1, \pi_1), \ldots, (q_t, a_t, \pi_t)\big) \leftarrow \tilde{\mathcal{S}}; b_i \leftarrow \mathcal{C}(\sigma, c, q_i, a_i, \pi_i) : \\ \exists \tilde{D} \text{ such that } (a_i = Q(q_i, \tilde{D}) \text{ or } b_i = 0) \text{ for all } i] \ \geq \ 1 - negl(k)$$

  In fact, it is even more natural to require that the adversary "know" the database $\tilde{D}$, say by requiring that $\tilde{D}$ be extractable in polynomial time from the description of the adversary. This is a more subtle property to capture—we refer the reader to the discussion of proofs of knowledge in [10].

2.1.1 Privacy

Informally, we require that an adversarial client interacting with an (honest) server learn no more information from the answer/proof pairs he receives than what he gets from the answers alone. specifically, a simulator who has access only to the query answers should be able to give believable-looking proofs of consistency. The definition comes from [17, 22, 20], though we use a cleaner formulation due to [15].

Definition 3 (Computational privacy). A consistent query protocol for $(\mathcal{D}, \mathcal{Q}, Q)$ is private if there exists a PPT simulator Sim, such that for every non-uniform PPT adversary $\tilde{\mathcal{C}}$, the outputs of the following experiments are computationally indistinguishable:

$$\begin{array}{l|l} \sigma \leftarrow \Sigma(1^k), & \sigma', c', \mathit{state}_{\mathsf{Sim}} \leftarrow \mathsf{Sim}(1^k), \\ (D, \mathit{state}_{\tilde{\mathcal{C}}}) \leftarrow \tilde{\mathcal{C}}(\sigma), & (D, \mathit{state}_{\tilde{\mathcal{C}}}) \leftarrow \tilde{\mathcal{C}}(\sigma'), \\ (c, \mathit{state}) \leftarrow \mathcal{S}_s(\sigma, D), & \\ \text{Output } z \leftarrow \tilde{\mathcal{C}}^{\mathcal{S}_a(\cdot, \mathit{state})}(c, \mathit{state}_{\tilde{\mathcal{C}}}) & \text{Output } z \leftarrow \tilde{\mathcal{C}}^{\mathsf{Sim}(\cdot, \mathit{state}_{\mathsf{Sim}}, Q(\cdot, D))}(c', \mathit{state}_{\tilde{\mathcal{C}}}) \end{array}$$

Here $\tilde{\mathcal{C}}^{\mathcal{O}(\cdot)}$ denotes running $\tilde{\mathcal{C}}$ with oracle access to $\mathcal{O}$. The simulator Sim has access to a query oracle $Q(\cdot, D)$, but *asks only queries which are asked to* Sim *by* $\tilde{\mathcal{C}}$.

Hiding Set Size. In general, a private protocol should not leak the size of the database [20]. Nonetheless, for the sake of efficiency we will sometimes leak a *polynomial* upper bound T on the database size, and call the corresponding protocols *size-T-private* [17]. This can be reflected in the definition by giving the simulator an upper bound T on the size of D as an additional input. One essentially recovers the original definition by letting T be super-polynomial, e.g. $T = 2^k$.

[3] One could imagine protecting against *all* adversaries and thus obtaining perfect soundness. We consider computational soundness since much greater efficiency is then possible.

Interactive proofs. The definitions extend to a model where consistency proofs are interactive (although the access of the simulator to the adversarial client is more tricky).

3 Data-Robust Algorithms and Consistent Query Protocols

In this section, we describe a general framework for obtaining secure consistent query protocols, based on designing efficient algorithms which are "data-robust". That is for any static data structure – even adversarially corrupted – the algorithm will answer all queries consistently with one (valid) database. Assuming the availability of a collision-resistant hash function, we show that any such algorithm which accesses its input by "following" pointers can be transformed into a consistent query protocol whose (non-interactive) consistency proofs have complexity at most proportional to the complexity of the algorithm. (In fact, the transformation works for arbitrary algorithms at an additional multiplicative cost of $\log N$, where N is the size of the database).

3.1 Data-Robust Algorithms

Suppose a programmer records a database on disk in a static data structure which allows efficient queries. The data structure might contain redundant information, for example to allow searching on two different fields. If the data structure later becomes corrupted, then subsequent queries to the structure might be mutually inconsistent: for example, if entries are sorted on two fields, some entry might appear in one of the two lists but not the other. A data-robust algorithm prevents such inconsistencies.

Suppose we have a query structure $(\mathcal{D}, \mathcal{Q}, Q)$. A data-robust algorithm (DRA) for these consists of two polynomial-time[4] algorithms (T, A): First, a setup transformation $T : D \rightarrow \{0,1\}^*$ which takes a database D and makes it into a static data structure (i.e. a bit string) $S = T(D)$ which is maintained in memory. Second, a query algorithm A which takes a query $q \in \mathcal{Q}$ and an arbitrary "structure" $\tilde{S} \in \{0,1\}^*$ and returns an answer. The structure $\tilde{S}$ needn't be the output of T for any valid database D.

Definition 4. The algorithms (T, A) form a *data-robust algorithm* for $(\mathcal{D}, \mathcal{Q}, Q)$ if:

- **Termination** A terminates in polynomial time on *all* input pairs $(q, \tilde{S})$, even when $\tilde{S}$ is not an output from T.
- **Soundness** There exists a function $T^* : \{0,1\}^* \rightarrow \mathcal{D}$ such that *for all* inputs $\tilde{S}$, the database $D = T^*(\tilde{S})$ satisfies $A(q, \tilde{S}) = Q(q, D)$ for all queries q.

  (There is no need to give an algorithm for T^*; we only need it to be well-defined.)
- **Completeness** For all $D \in \mathcal{D}$, we have $T^*(T(D)) = D$.

  (That is, on input q and $T(D)$, the algorithm A returns the correct answer $Q(q, D)$.)

We only allow A *read* access to the data structure (although the algorithm may use separate space of its own). Moreover, A is *stateless*: it shouldn't have to remember any information between invocations.

The running time of A. There is a naive solution to the problem of designing a DRA: A could scan the corrupted structure $\tilde{S}$ in its entirety, decide which database D this corresponds to, and answer queries with respect to D. The problem, of course, is that this requires at least linear time *on every query* (recall that A is stateless). Hence the task of designing robust algorithms is most interesting when there are natural *sub-linear* time algorithms; the goal is then to maintain efficiency while also achieving robustness. In our

[4] We assume for simplicity that the algorithms are deterministic; this is not strictly necessary.

setting, efficiency means the running-time of the algorithm A on *correct* inputs, in either a RAM or pointer-based model. On incorrect inputs, an adversarially-chosen structure could, in general, make A waste time proportional to the size of the structure $\tilde{S}$; the termination condition above restricts the adversary from doing too much damage (such as setting up an infinite loop, etc).

Error model. Although the design of DRAs is an algorithmic question, the error model is a cryptographic one. Much work has been done on constructing codes and data-structures which do well against randomly placed errors, or errors which are limited in number (witness the fields of error-correcting codes, fault-tolerant computation and fault-tolerant data structures). However, in this setting, there are no such limitations on how the adversary can corrupt the data structure. We only require that the algorithm answer consistently for any given input structure. Our error model is more similar to that of property testing and probabilistically checkable proofs, where the goal is to test whether a given string is close to a "good" string; however, we only require computational soundness, which allows us to use different (and simpler) techniques.

3.2 Constructing Consistent Query Protocols From DRAs

Given a DRA which works in a pointer-based memory model, we can obtain a cryptographically secure consistent query protocol of similar efficiency. Informally, a DRA is pointer-based if it operates by following pointer in a directed acyclic graph with a single source (see Section 3.2.1 for details). Most common search algorithms fit into this model.

Theorem 1. *Let (T, A) be a DRA for query structure $(\mathcal{D}, \mathcal{Q}, Q)$ which fits into the pointer-based framework of Section 3.2.1. Suppose that on inputs q and $T(D)$ (correctly formed), the algorithm A examines b memory blocks and a total of s bits of memory, using t time steps. Assuming the availability of a public collision-resistant hash function, there exists a consistent query protocol for $(\mathcal{D}, \mathcal{Q}, Q)$ which has proof length $s + kb$ on query q. The server's computation on each query is $O(s + t + kb)$.*

To get a consistent query protocol from a DRA, we essentially build a Merkle tree (or graph, in fact) which mimics the structure of the data, replacing pointers with hashes of the values they point to. The client runs the query algorithm starting from hash of the unique source in the graph (that hash value is the public commitment). When the query algorithm needs to follow a pointer, the server merely provides the corresponding pre-image of the hash value. Details are in Section 3.2.2. If we run this transformation on data-structures which are not data-robust, we still obtain an intersting guarantee: the resulting protocol is secure as long as the server generating the commitment is honest. This is essentially the transformation of [13, 19].

The remainder of this section contains the details and proof of Theorem 1. We first specify what we mean by a pointer-based framework and then proof the theorem.

3.2.1 Pointer-Based Algorithms

We say a pair of algorithms (T, A) is *pointer-based* if

1. A expects its input data structure $S = T(D)$ to be a *rooted* directed graph of memory blocks. That is, the output of the setup algorithm T is always the binary representation of a directed graph. Each node in the graph has a list of outgoing edges as well as some associated data.

2. A accesses its input S and uses node names in a limited way:

   - A can get the contents of a node u in the graph by issuing the instruction $\mathtt{get}(u)$. This returns the associated data $\mathtt{data}_u$ and a list of outgoing edges $v_{1,u}, v_{2,u}, \ldots, v_{n_u,u}$.

- A always starts out by getting the contents of the root of the graph by issuing the instruction $\mathsf{getroot}()$.
- The only operations A performs on node names are (a) getting the contents of a node, and (b) comparing two node names for equality.
- The only node names which A uses are those obtained from the outgoing edge lists returned by calls to $\mathsf{getroot}()$ and $\mathsf{get}(\cdot)$.

For example, S could be a sequence of blocks separated by a distinguished character, $S = b_1\#\ldots\#b_n$. Each block b_i would consist of some data (an arbitrary string) and "pointers", each of which is the index (in the string S) of the start of another block b_j. The root of the graph could be the first block by convention.

Finally, we need some simple robustness properties of this graph representation (which can be satisfied by the example representation above). We assume:

3. The binary representation of the graph is such that when A is fed an improperly formed input $\tilde{S}$ (i.e. one which is not an output of T), then the behaviour of $\mathsf{get}(\cdot)$ and $\mathsf{getroot}$ is not "too bad":

- When $\mathsf{get}(u)$ or $\mathsf{getroot}()$ is called, if the corresponding part of the input string is not well-formed (i.e. is not a tuple of the form $(\mathsf{data}_u, v_{1,u}, v_{2,u}, \ldots, v_{n_u,u})$), then the call will return a distinguished value $\perp$.
- Both $\mathsf{get}(\cdot)$ and $\mathsf{getroot}()$ always terminate in time linear in the length of the corrupted structure $\tilde{S}$.

Many common search algorithms can be cast in this pointer-based framework. For example, the algorithm for searching in a binary tree takes as input a tree, which it explores from the root by following pointers to right and left children of successive nodes. Indeed, almost all search algorithms for basic dynamic data types can be viewed in this way. Moreover, any algorithm designed for a RAM machine can also be cast in this framework at an additional logarithmic cost: if the total memory space is N, simply build a balanced tree of pointers of height $\log N$, where the i-th leaf contains the data stored at location i in memory.

3.2.2 Proof of Theorem 1

Let (T, A) be a DRA for query structure $(\mathcal{D}, \mathcal{Q}, Q)$ which fits into the pointer-based framework described above. For simplicity, suppose that a correctly formed structure (i.e. an output of T) never contains a pointer cycle, that is, the resulting graph is acyclic.[5]

Proof. The idea is to construct a "hash graph" which mimicks the data structure $T(D)$, replacing pointers with hash values from the CRHF. Let H be a publicly available, randomly chosen member of a CRHF with security parameter k. Depending on the setting, we can either assume that H is common knowledge (in which case there is no need for public randomness), or ask explicitly that a trusted third party output a description of H (in which case the distribution $\Sigma(1^k)$ is the key generator for the CRHF).

Setup algorithm. The server setup algorithm $\mathcal{S}_s$ is as follows: on input D, run T to get $S = T(D)$. View S as a directed graph, with memory blocks as nodes and pointers as edges. This graph can be topologically sorted (by assumption: no pointer cycles). There is a single source, the query algorithm's starting memory block (i.e. the root of the graph)[6]. Now proceed from sinks to the source by adding a hash value (called h_u) at each node u: For a sink, attach the hash of its binary representation; this is basically $h_u = H(\mathsf{data}_u)$. When

[5] This restriction is not necessary. General graphs can be handled at a logarithmic cost by building a tree over the memory structure.

[6] There could in principle be other sources, but by assumption on how A operates it will never access them, so $\mathcal{S}$ can safely ignore them.

u is an internal node, replace each of its pointers $v_{i,u}$ by the hash values of the nodes they point to and then set h_u to be the hash of the binary representation of the transformed block $h_u = H(\mathsf{data}_u, h_{v_{1,u}}, \ldots, h_{v_{n_u,u}})$. At the end, one obtains a hash h_{root} for the source. The server publishes the commitment $c = h_{root}$, and stores S and the associated hash values as the internal variable *state*.

Query algorithm. Given a query q and the setup information *state*, the server $\mathcal{S}_a$ runs the robust algorithm A on the data structure S, and keeps track of all the memory blocks (i.e. nodes) which are accessed by the algorithm (by looking at calls to the $\mathsf{get}(\cdot)$ instruction). Denote the set of accessed nodes by S_q. The answer a is the output of A; the proof of consistency π is the concatenation of the "transformed" binary representations $(\mathsf{data}_u, h_{v_{1,u}}, \ldots, h_{v_{n_u,u}})$ of all the nodes $u \in S_q$,as well as a description of S_q and where to find each node in the string π.

Consistency check. On inputs c, q, a, π (where π consists of a the description of a set of nodes S_q as well as their transformed representations), the client $\mathcal{C}$ will verify the answer by running A, using the proof π to construct the necessary parts of S.

The first step is to reconstruct the subgraph of memory blocks corresponding to the set of accessed nodes S_q. The client $\mathcal{C}$ checks that :

- π is a sequence of correctly formed "transformed" binary representations of memory blocks along with associated hash values.
- S_q forms a subgraph entirely reachable from the root (since A starts from the root and follows pointers, this holds when the server is honest).
- the hash values present are consistent: for each node u, and for each neighbor $v_{i,u}$ of u which is in S_q, check that the value $h_{v_{i,u}}$ attached to u is the hash of the transformed representation of $v_{i,u}$.
- the value h_{root} constructed from the input π is indeed equal to the public commitment c.

Next, $\mathcal{C}$ runs A on this reconstructed S_q. It checks that all the nodes requested by A are in S_q and that A returns the correct value a.

Since the hash function is collision-resistant, there is only one such subgraph S_q which can be revealed by the server. More precisely, there is one overall graph – the committed data structure – such that the server can reveal (reachable) parts of the graph[7]. Thus the server is committed to a data structure $\tilde{S}$ which is bounded in size by the server's memory. By the properties of the data-robust algorithm, an honest server will always be able to answer a query and provide a valid proof of correctness, whereas a malicious server can (at most) answer queries with respect to the database $T^*(\tilde{S})$. □

4 Orthogonal Range Queries

In the case of join queries, a database D is a set of key/value pairs (entries) where each key is a point in $\mathbb{R}^d$, and each query is a rectangle $[a_1, b_1] \times \cdots \times [a_d, b_d]$. Note that these are also often called *(orthogonal) range queries*, and we shall adopt this terminology here for consistency with the computational geometry literature. For concreteness, we consider the two-dimensional case; the construction naturally extends to higher dimensions. In two dimensions, each query q is a rectangle $[a_1, b_1] \times [a_2, b_2]$. The query answer $Q(q, D)$ is a list of all the entries in D whose key $(\mathsf{xkey}, \mathsf{ykey})$ lies in q.

In this section we give a simple, efficient DRA for range queries and show how to modify it to make an efficient consistent query protocol.

[7]The proof of this is standard: suppose that the server can produce two graphs consistent with the hash of the root $c = h_{root}$. By induction on the distance from the root at which the two graphs differ, one can find a pair of strings which hash to the same value

Algorithm 1. $A_{\mathsf{1DRT}}(\ [a,b],\ n,\ )$
Input: a target range $[a,b]$, a node n in a (possibly misformed) 1-DRT.
Output: a set of $(\mathsf{key}, \mathsf{value})$ pairs.

1. **if** n is not properly formed (i.e. does not contain the correct number of fields) **then** return $\emptyset$
2. **if** n is a leaf: **if** $a_n = b_n = \mathsf{key}_n$ and $\mathsf{key}_n \in [a,b]$, **then** return $\{(\mathsf{key}_n, \mathsf{value}_n)\}$ **else** return $\emptyset$
3. **if** n is an internal node:

- $l \leftarrow \mathsf{left}_n, r \leftarrow \mathsf{right}_n$
- **if** $a_n = a_l \leq b_l < a_r \leq b_r = b_n$ **then** return $A_{\mathsf{1DRT}}\ (\ [a,b],\ l) \cup A_{\mathsf{1DRT}}\ (\ [a,b],\ r)$
- **else** return $\emptyset$

Figure 1: Data-robust algorithm A_{1DRT} for querying one-dimensional range trees

4.1 A Data-Robust Algorithm for Range Queries

Various data structures for efficient orthogonal range queries exist (see [12] for a survey). The most efficient (non-robust) solutions have query time $O((m+1)\log^{d-1} N)$ for d-dimensional queries. We review *multi-dimensional range trees* (due to Bentley [2]), and show how they can be queried robustly. The query time of the robust algorithm is $O((m+1)\log^{d} N)$. It is an interesting open question to find a robust algorithm which does as well as the best non-robust algorithms.

4.1.1 One-Dimensional Range Trees

Multidimensional range trees are built recursively from one-dimensional range trees (denoted 1-DRT), which were also used by [5, 22, 17]. In a 1-DRT, $(\mathsf{key}, \mathsf{value})$ pairs are stored in sorted order as the leaves of a (minimum-height) binary tree. An internal node n stores the minimum and maximum keys which appear in the subtree rooted at n (denoted a_n and b_n respectively). For a leaf l, we take $a_l = b_l$ to be the value of the key_l key stored at l. Additionally, leaves store the value value_l associated to key_l.

Setup. Given a database $D = \{(\mathsf{key}_1, \mathsf{value}_1), \ldots, (\mathsf{key}_N, \mathsf{value}_N)\}$, the setup transformation T_{1DRT} constructs a minimum-height tree based on the sorted keys. All the intervals $[a_n, b_n]$ can be computed using a single post-order traversal.

Robust queries. It is easy to see that a 1-DRT allows efficient range queries when it is correctly formed (given the root n of a tree and a target interval $[a,b]$, descend recursively to those children whose intervals overlap with $[a,b]$). However, in our setting we must also ensure that the queries return consistent answers even when the data structure is corrupted. The data structure we will use is exactly the one above. To ensure robustness we will modify the querying algorithm to check for inconsistencies.

Assume that we are given a *rooted* graph where all nodes n have an associated interval $[a_n, b_n]$, and all nodes have outdegree either 0 or 2. A *leaf* l is any node with outdegree 0. A leaf is additionally assumed to have to extra fields key_l and value_l. Consider the following definitions:

Definition 5. A node n is *consistent* if its interval agrees with those of its children. That is, if the children are l and r respectively, then the node is consistent if $a_n = a_l \leq b_l < a_r \leq b_r = b_n$. Moreover, we should have $a_n = b_n$ for a node if and only if it is a leaf.

A path from the root to a node is *consistent* if n is consistent and all nodes on the path to the root are also consistent.

Definition 6. A leaf l in a 1-DRT is *valid* if there is a consistent path from the root to l.

In order to query a (possibly misformed) 1-DRT in a robust manner, we will ensure that the query algorithm A returns *exactly* the set of valid leaves whose keys lie in the target range. In a "normal" (i.e. correctly formed) 1-DRT, every leaf is valid, and so the algorithm will return the correct answer. In a corrupted structure, the algorithm will always answer consistently with the database consisting of the set of points appearing at valid leaves. Thus for any string $\tilde{S}$, the database $T^*(\tilde{S})$ consists of the data at all the valid leaves one finds when $\tilde{S}$ is considered as the binary encoding of a graph.

Algorithm 1 ($A_{1\mathsf{DRT}}$) will query a 1-DRT robustly. When it is first called, the argument n will be the root of the graph. Essentially, $A_{1\mathsf{DRT}}$ runs the ordinary (non-robust) search algorithm, checking all nodes it passes to ensure that they are consistent (Definition 5). It also checks that it never visits the same node twice (in such a case, it must be that the graph the algorithm receives as input is not a tree).

The algorithm $A_{1\mathsf{DRT}}$ operates in the "pointer-based" model. Thus the first node on which the algorithm is called is obtained through a call to $\mathtt{getroot}()$. The neighbours of an internal node n are its two children left_n and right_n. For clarity of the algorithm, we have not explicitly included calls to $\mathtt{get}(\cdot)$ in the description of the algorithm.

The following lemma proves that one-dimensional range trees, along with the algorithm $A_{1\mathsf{DRT}}$, form a DRA for range queries.

Lemma 2. *The algorithm $A_{1\mathsf{DRT}}$ will return exactly the set of valid leaves whose keys are in the target range. In the worst case, the adversary can force the queries to take time $O(s)$ where s is the total size of the data structure. Conversely, given a collection of N entries there is a tree such that the running time of the algorithm is $O((m+1)\log N)$, where m is the number of points in the target range. This tree can be computed in time $O(N \log N)$ and takes $O(N)$ space to store.*

Proof. On one hand, the algorithm is complete, since in a correctly formed tree every node will pass the consistency checks, and so the algorithm will return exactly the set of leaves whose keys are in the target range.

Before proving robustness, it is important to note that there are some kinds of misformed data we don't have to worry about. First, we can assume that all nodes are correctly formed (i.e. have the correct number of fields and the correct types of data) since incorrectly formed nodes will be ignored by the algorithm. Thus we can assume that the algorithm is indeed given some kind of graph as input, although it isn't necessarily a tree. Moreover, we can assume all nodes in the graph have outdegree either 2 or 0.

The proof of robustness follows from the properties of consistent nodes, which in turn follow from the definitions. For any node n which is on a consistent path from the root:

1. The consistent path from the root is unique.

2. No valid leaves *inside* n's subtree have keys *outside* n's interval.

3. If another node n' is on a consistent path from the root, and $[a_{n'}, b_{n'}] \cap [a_n, b_n] \neq \emptyset$, then n' is either an ancestor or a descendant of n (thus one of the two intervals includes the other).

A corollary of these properties is that *no node will be visited twice by the algorithm*. This is because the algorithm expects intervals to shrink at each recurisve step, and so it will never follow a link which leads to a node earlier on in the current recursion stack. Moreover, there can never be two distinct paths by which the algorithm arrives at a node n: because the algorithm is always checking for consistency, the two ancestors

n' and n'' of n would have to be consistent nodes with overlapping intervals, contradicting the properties above.

Hence, the algorithm will visit valid leaves at most once, and never visit invalid leaves. Moreover, it will visit all the valid leaves in the target interval (by inspection). Thus running A_{1DRT} on a string $\tilde{S}$ procudes answers consistent with $T^*_{\mathsf{1DRT}}(\tilde{S})$, the set of data points stored at valid leaves in the graph represented by $\tilde{S}$. □

4.1.2 Two-Dimensional Range Trees

Here, the database is a collection of triples $(\mathsf{xkey}, \mathsf{ykey}, \mathsf{value})$, where the pairs $(\mathsf{xkey}, \mathsf{ykey})$ are all distinct (they need not differ in both components). The data structure, a two-dimensional range tree (denoted 2-DRT), is an augmented version of the one above. The skeleton is a 1-DRT (called the *primary* tree), which is constructed using the xkey's of the data as its key values. Each node in the primary tree has an attached 1-DRT called its *secondary* tree:

- Each leaf l of the primary tree (which corresponds to a single xkey value $a_l = b_l$) stores all entries with that xkey value. They are stored in the 1-DRT tree_l which is constructed using ykey's as its key values.
- Each internal node n (which corresponds to an interval $[a_n, b_n]$ of xkey's) stores a 1-DRT tree_n containing all entries with xkey's in $[a_n, b_n]$. Again, this "secondary" tree is organized by ykey's.

The setup algorithm T_{2DRT} creates a 2-DRT given a database by first sorting the data on the key xkey, creating a *primary* tree for those keys, and creating a secondary tree based on the ykey for each of nodes in the primary tree. In a 2-DRT, each point is stored d times, where d is its depth in the primary tree. Hence, the total storage can be made $O(N \log N)$ by choosing minimum-height trees.

Searching in a 2-DRT. The natural recursive algorithm for range queries in this structure takes time $O(\log^2 N)$ [12]: Given a target range $[a^{(x)}, b^{(x)}] \times [a^{(y)}, b^{(y)}]$ and an internal node n, there are three cases: if $[a^{(x)}, b^{(x)}] \cap [a_n, b_n] = \emptyset$, then there is nothing to do; if $[a^{(x)}, b^{(x)}] \supseteq [a_n, b_n]$, then perform a search on the second-level tree attached to n using the target range $[a^{(y)}, b^{(y)}]$; otherwise, recursively explore n's two children.

Based on the natural query algorithm, we can construct a DRA A_{2DRT} by adding the following checks:

- All queries made to the 1-D trees (both primary and secondary) are made robustly following Algorithm 1 (A_{1DRT}), i.e. checking consistency of each explored node.
- For every point which is retrieved in the query, make sure it is present and valid in all the secondary 1-D trees which are on the path to the root (in the primary tree).

The following definition captures *validity*, which is enforced by the checks above:

Definition 7. A point $p = (\mathsf{xkey}, \mathsf{ykey}, \mathsf{value})$ in a (corrupted) 2-DRT is *2-valid* if

1. p appears at a valid leaf in the secondary 1-DRT tree_l belonging to a *leaf* l of the primary tree with key value $\mathsf{xkey} = a_l = b_l$.
2. For every (primary) node n on the path to l from the root of the primary tree, n is consistent and p is a valid leaf in the (one-dimensional) tree tree_n.

Now given a (possibly corrupted) 2-DRT and a point $p = (\mathsf{xkey}, \mathsf{ykey}, \mathsf{value})$, it is easy to check whether or not p is 2-valid: one first searches for a leaf l with key xkey in the primary tree, exploring only consistent nodes. Then, for each node n on the path from l to the root (including l and the root), one checks to ensure that p appears as a valid leaf in the tree_n.

Algorithm 2. $A_{\mathsf{2DRT}}(\ [a^{(x)}, b^{(x)}] \times [a^{(y)}, b^{(y)}],\ n)$
Input: a target range $[a^{(x)}, b^{(x)}] \times [a^{(y)}, b^{(y)}]$, a node n in a 2-DRT.
Output: a set of $(\mathsf{xkey}, \mathsf{ykey}, \mathsf{value})$ triples.

1. **if** n is not properly formed (i.e. does not contain the correct number of fields), **then** return $\emptyset$.
2. Check for consistency (if check fails, return $\emptyset$):
   - **if** n is a leaf **then** check $a_n = b_n = \mathsf{key}_n$
   - **if** n is an internal node, **then** check $a_n = a_{\mathsf{left}_n} \le b_{\mathsf{left}_n} < a_{\mathsf{right}_n} \le b_{\mathsf{right}_n} = b_n$
3. (a) **if** $[a_n, b_n] \cap [a^{(x)}, b^{(x)}] = \emptyset$ **then** return $\emptyset$
   (b) **if** $[a_n, b_n] \subseteq [a^{(x)}, b^{(x)}]$ **then**
   - $B \leftarrow A_{\mathsf{1DRT}}(\ [a^{(y)}, b^{(y)}],\ \mathsf{tree}_n)$
   - Remove elements of B for which $\mathsf{xkey} \notin [a_n, b_n]$
   - **if** n is an internal node:
     For each point p in B, check that p is 2-valid in either left_n or right_n.
     If the check fails, remove p from B.
   - Return B

   (c) Otherwise
   - $$\begin{aligned} B \ \leftarrow\ & A_{\mathsf{2DRT}}\Big(\ ([a^{(x)}, b^{(x)}] \cap [a_{\mathsf{left}_n}, b_{\mathsf{left}_n}]) \times [a^{(y)}, b^{(y)}],\ \mathsf{left}_n\Big) \\ & \cup A_{\mathsf{2DRT}}\Big(\ ([a^{(x)}, b^{(x)}] \cap [a_{\mathsf{right}_n}, b_{\mathsf{right}_n}]) \times [a^{(y)}, b^{(y)}],\ \mathsf{right}_n\Big) \end{aligned}$$
   - Remove elements of B which are not valid leaves of tree_n.
   - Return B

Figure 2: Data-robust algorithm A_{2DRT} for querying two-dimensional range trees

For robust range queries, we obtain Algorithm 2 (A_{2DRT}). As before, the idea is to return only those points which are 2-valid. Thus, for an arbitrary string $\tilde{S}$, the induced database $T^*_{\mathsf{2DRT}}(\tilde{S})$ is the collection of all 2-valid points in the graph represented by $\tilde{S}$. The following lemma shows that the algorithms $(T_{\mathsf{2DRT}}, A_{\mathsf{2DRT}})$ form a DRA for two-dimensional range queries with query complexity $O((m+1)\log^2 N)$ (where m is the number of points in the target range).

Lemma 3. *Algorithm 2 (A_{2DRT}) will return exactly the set of 2-valid points which are in the target range. On arbitrary inputs, A_{2DRT} terminates in worst-case time $O(L)$, where L is the total size of the data structure.*

Conversely, given a collection of N entries there is a tree such that the running time of the algorithm A_{2DRT} is $O((m+1)\log^2 N)$, where m is the number of points in the target range. This tree can be computed in time $O(N \log^2 N)$ and takes $O(N \log N)$ space to store.

Proof. (sketch) As in the one-dimensional case, the algorithm will never explore the same node twice, and so we may think of the corrupted input to the algorithm as a tree. Moreover, since the algorithm is checking for proper formatiing of nodes, we can assume that this graph consists of a number of "primary" nodes with secondary trees dangling off them. Finding the running time of the algorithm on well-constructed inputs is a straightforward exercise.

On one hand, one can see by inspection that any 2-valid point in the target range will be output by the algorithm, since all the checks will be passed. Moreover, no valid point outside the target range will be output.

On the other hand, consider any point that is output by the algorithm. It must have appeared in the set B at stage 3(b) of the algorithm for some node n. Thus it is a valid leaf in tree_n. Moreover, it must be valid in either left_n or right_n, because of the checks made at step 3(b). This means there is a leaf l which is a descendant of n such that p is a valid point in $\mathtt{tree}_l$ and in all the trees of the nodes on the path from n to l. Finally, as the recursion exits (in step 3(c)), the algorithm will verify that p appears at a valid leaf in all the nodes on the path from the root. to n. Thus p must be a 2-valid point. □

Remark 1. As mentioned above, more efficient data structures and algorithms for planar orthogonal queries exist [12], but it is not clear how to make them robust without raising the query time back to $O((m+1)\log^2 N)$. This is an interesting open question.

One can use similar ideas to make robust range queries on d-dimensional keys, where $d \geq 2$. The structure is built recursively, as in the 2-D case. Although the algorithm is polylogarithmic for any fixed dimension, the exponent increases:

Lemma 4. *There exists a* DRA *for d dimensional range queries such that queries run in time $O((m+1)\log^d N)$, and the data structure requires $O(N\log^d N)$ preprocessing and $O(N\log^{d-1} N)$ storage.*

4.2 Efficient Query Protocol

Given this algorithm, the (non-private) query protocol can be constructed as in Section 3.2: the server creates a tree as in the previous section. For each key/value pair, he computes a hash value h_{key}. He now works his way up through the various levels of the tree, computing the hash values of nodes as the hash of the tuple (min, max, left child's hash value, right child's hash value). A given key will appear roughly $\log N$ times in the tree; the same value h_{key} should be used each time.

To answer a range query, the server runs the algorithm of the previous section. He need only send the hash values and intervals of nodes on the "boundary" of the subgraph (in memory) which was explored, i.e. the leaves and the siblings of the nodes on their paths to the root (the information corresponding to the interior nodes can be reconstructed from the boundary nodes). This yields the following:

Theorem 5 (Two dimensions). *Assuming the existence of collision-resistant hash functions, there is a consistent query protocol for two-dimensional range queries with commitment size k and non-interactive consistency proofs of length at most $O(k(m+1)\log^2 N)$, where m is the number of keys in the query range, and k is the security parameter (output size of the hash function).*

For higher dimensions, our construction yields proofs of length $O(k(m+1)\log^d N)$.

5 Privacy for Consistent Query Protocols

5.1 Privacy Via Generic Techniques

One can construct private CQPs (Definition 3) with good asymptotic complexity using generic techniques, as follows. *Universal arguments*, due to Barak and Goldreich [1], allow one to give an interactive, zero-knowledge argument of knowledge of an NP statement of arbitrary polynomial length, using only a fixed, $poly(k)$ number of bits of communication. This allows one to handle arbitrary query structures (as long as answering queries takes at most polynomial time): the server sends the answer to a query, and then proves interactively that it "knows" a string $\tilde{\pi}$ which the client would accept as a valid proof of consistency. This approach even hides the set size of the database as in [20], since the universal argument leaks only a super-polynomial bound on the length of the statement being proven. Unfortunately, the known construction of universal arguments is cumbersome, even by the standards of theoretical cryptography, since it uses the machinery of probabilistically checkable proofs.

One can gain some simplicity and efficiency by starting from a (non-private) efficient CQP, and replacing each proof of consistency π with an ordinary zero-knowledge argument of knowledge (ZKAK) of π (for example, see Goldreich [10], Chapter 4.7.3). If a public random string is available, one can also use non-interactive zero-knowledge proofs of knowledge (NIZKPK).

This approach will typically leak some bound on the size N of the database, since both Ordinary ZKAK's and NIZKPK's may leak a polynomial upper bound on the lenght of the statement being proven. One can avoid that leakage if the original proofs take time and communication $poly(\log N)$, as with membership and orthogonal range queries. Replacing N with the upper bound 2^k, we once again again get $poly(k)$ communication.

We summarize this discussion in Theorem 6. If we consider the specific case of CQP's for membership queries, then the theorem says that *zero-knowledge set* protocols [20] can be constructed based on general assumptions, such as the existence of non-interactive zero-knowledge proof systems. A different proof of this specific statement was later given by Healy et al. [15].

Theorem 6. *(a) Assume that there exists a collision-resistant hash family. For any query structure with polynomial complexity, there exists a* private CQP *with a constant number of rounds of interaction and* $poly(k)$ *communication.*

(b) Given a public random string, any CQP *with proofs of length* $\ell(N)$ *can be made size-*N*-private with no additional interaction at a* $poly(k\ \ell(N))$ *multiplicative cost in communication, assuming non-interactive zero-knowledge proof systems exist.*

5.2 Explicit-Hash Merkle Trees in Brief

Although the asymptotics of Theorem 6 are good, the use of generic NP reductions means that the advantages only appear for large datasets. We therefore construct simpler protocols tailored to Merkle trees.

The basic Merkle tree commitment scheme leaks information about the committed values, since a collision-resistant function cannot hide all information about its input.[8] At first glance, this seems easy to resolve: one can replace the values a_i at the leaves of the tree with hiding commitments $C(a_i)$. This doesn't work, since there is may be additional structure to the values $a_1, ..., a_N$ which is revealed when one reveals a path in the tree. For example, in CQPs for range queries, the entries are stored in sorted order. Revealing the path to a particular value then reveals its rank in the data set. The problem gets even more complex when we want to reveal a subset of the values, as we have to hide not only whether paths go left or right at each branching in the tree, but whether or not different paths overlap.

A generic solution is to provide a hiding commitment to the description of each node on the path, and then give a zero-knowledge proof that the committed string is consistent with the public hash value (the root of the hash tree). The main bottleneck is in proving that $y = H(x)$, given commitments $C(x)$ and $C(y)$. It is not known how to do that without going through either general NP reductions or oblivious circuit evaluation protocols, both of which are extremely inefficient when applied to a circuit as complex as a hash function. This seems to be a fundamental problem with privacy of Merkle-tree commitments: revealing the hash values reveals structural information about the tree, and not revealing them and instead proving consistency using generic ZK techniques kills efficiency.

[8] There are limited ways in which hash functions may hide information, as discussed by Canetti, Micciancio and Reingold [3]. That definition of privacy is not strong enough for our setting.

The challenge, then, is to provide zero-knowledge proofs that a set $a'_1, ..., a'_t$ is a subset of the committed values, without going through oblivious evaluation of such complicated circuits. We present a modification of Merkle trees where one reveals all hash-function input-output pairs explicitly, yet retains privacy. We call our construction an *Explicit-Hash Merkle Tree*. The construction is explained below, in Section 5.3.

Lemma 7. *Assuming the existence of collision-resistant hash families and homomorphic perfectly-hiding commitment schemes,* explicit-hash Merkle trees *allow proving (in zero-knowledge) the consistency of t paths (of length $d = \log N$) using $O(d \cdot t^2 \cdot k^2)$ bits of communication, where k is the security parameter. The protocol uses 5 rounds of interaction. It can be reduced to a single message in the random oracle model.*

To illustrate the technique, we apply it to one-dimensional range queries. The main drawback of the resulting protcol is that the server needs to maintains state between invocations; we denote by t the number of previous queries.

Theorem 8. *There exists an efficient,* size-N-private *consistent query protocol for 1-D range queries. For the t-th query to the server, we obtain proofs of size $O((t + m) \cdot s \cdot k^2 \cdot \log N)$, where s is the maximum length of the keys used for the data, and m is the total number of points returned on range queries made so far. The protocol uses 5 rounds of interaction and requires no common random string. The protocol can be made non-interactive in the random oracle model.*

The remainder of this section gives the details of the results above. The proof of Lemma 7 can be found in Section 5.3. The final subsection (Section 5.4) gives a proof of Theorem 8.

5.3 Explicit-Hash Merkle Trees in Detail

As mentioned above, Merkle trees allow one to commit to a large number of values via a short commitment, and to reveal some subset $a'_1, ..., a'_t$ of those values very efficiently, by showing a path from the root to that particular value. We explain how to modify that scheme to hide the remaining committed values, while leaving the hash function evaluations explicit, i.e. without going through oblivious evaluation of such complicated circuits. The goal of this section, then, is to prove Lemma 7.

Server storage. Let $C(\cdot)$ be a non-interactive commitment scheme to messages of arbitrary length. It will be convenient to assume that $C(\cdot)$ is homomorphic, that is given commitments to m_1 and m_2 it is possible to produce a commitment to $m_1 + m_2$ ([9]). Such schemes exist based on a number of assumptions, such as the hardness of discrete logarithm extraction (e.g. Pedersen's scheme [28]). Let H be selected from a collision-resistant hash function family.

We will build a hash tree based on commitments to nodes, that is the server will actually commit to commitments of the nodes in the tree. Moreover, rather than store explicit hash values in the tree we will store commitments to those values. Specifically, for each node n in the tree, we will define three values:

- The basic string representation: x_n is the information stored at the node n.
- A hash pre-image for n: c_n is a particular commitment to the value x_n via the commitment shceme $C(\cdot)$.
- The corresponding hash value: $y_n = H(c_n)$ is the hash value for n which we will store at the parent of n.

For a leaf l, we have $x_l = a_l$, and c_l is a commitment $C(a_l)$. For an internal node n, we have $x_n = (H(c_{\mathsf{left}_n}), H(c_{\mathsf{right}_n}))$, and c_n is a component-wise commitment to x_l using $C(\cdot)$, i.e.
$c_l \leftarrow (C(H(c_{\mathsf{left}_n})), C(H(c_{\mathsf{right}_n})))$.

The public commitment is the value $y_{root} = H(x'_{root})$.

Definition 8. For two strings x and y, we say $y \lhd x$ if y is the hash of some valid commitment to x, i.e. if there are random coins ω such that $y = H(C(x; \omega)$.

[9]In fact, we only need to be able to prove the equality of two committed strings without revealing them.

Protocol outline. Suppose the server now wants to reveal t values from the tree. Let $d = \log N$ be the depth of the tree. For each leaf l to be revealed, the server finds the corresponding path $n_1, ..., n_d$ where n_1 is the root and n_d is l. He sends to the client the data a_l, plus fresh commitments to the values x_{n_i} and y_{n_i}. He then proves that these form a consistent path in two stages.

1. For each of the t paths, Server sends $u_1 = C(x_{n_1}), ..., u_d = C(x_{n_d})$ and $v_1 = C(y_{n_1}), ..., v_d = C(y_{n_d})$.
2. The server proves that each of the pairs u_i, v_i is a commitment to a pair x_i, y_i such that $y_i \lhd x_i$.
3. The server proves that the committed nodes actually form a path, that is for every $i > 1$, the server shows that one of the y_i appears as one of the components of x_{i-1}.
4. The server proves that the first node is indeed the root by opening the commitment v_1 to reveal the public commitment string y_{root}.

The first proof is the trickiest, since we wish to use only explicit hash function evaluation (never oblivious) but also not reveal any information on possible relations between the various paths.

Proving that $y_i \lhd x_i$. There are t paths of length d for which this must simultaneousely be proven. At the very least, the server will have to reveal the hash pre-images for all the nodes in those t paths. However, depending on how the paths overlap, there may be far fewer than td such nodes (and hence hash pre-images), and any repetitions will be easy to detect. Thus, the server will additionally send enough "dummy pre-images" so that the total number of committed nodes claimed to be in the hash tree is exactly td. The dummy values are other hash pre-images present in the hash tree. Formally:

1. 1. Let $\{n^{(1)}, ..., n^{(s)}\}$ be the union of the nodes on all t paths ($s \leq td$). We pad this set with $td - s$ other nodes $n_{s+1}, ..., n_{td}$ (arbitrary nodes will work) to get a set of td nodes.
   Let $c^{(1)}, ..., c^{(s)}$ be the corresponding pre-images, i.e. $c^{(j)} = c_{n^{(j)}}$.
   2. Server sends $\{c^{(1)}, ..., c^{(td)}\}$ to the client in random order.
3. Repeat the following cut-and-choose protocol k times:

   1. Server chooses a permutation $\pi \leftarrow S_{td}$, and sends fresh commitments $c'_{n^{(j)}} = C(x_{n^{(j)}})$ to all td nodes $n^{(j)}$, as well as commitments $C(y_{n^{(j)}})$ to the hash values $y_{n^{(j)}} = H(c^{(j)})$. These commitments are permuted according to π before sending.
   2. Client answers with a challenge bit $b \leftarrow \{0, 1\}$.
   3. If $b = 0$, the server:
      1. Sends π proves that for each of the td nodes $n^{(j)}$, $c'_{n^{(j)}}$ and $c^{(j)}$ are commitments to the same value. (This is easy since the commitment scheme is homomorphic.)
      2. opens all commitments to $y_{n^{(j)}}$ (client verifies $y_{n^{(j)}} = H(c^{(j)})$).

      If $b = 1$, the server:

      1. Shows that each of the commitments u_i is equivalent to one of the commitments $c'_{n^{(j)}}$ and that the commitment v_i is equivalent to the corresponding committed hash value $C(y_{n^{(j)}})$.

At the end of this proof, the client should be convinced that each of the commitment pairs (u_i, v_i) corresponds to one of the values $c^{(j)}$, and that the underlying pair x_i, y_i satisfies $y_i \lhd x_i$.

Proving that the path is consistent. We now have pairs of commitments u_i, v_i which hide valid pairs x_{n_i}, y_{n_i}, where $y_{n_i} = H(C(x_{n_i}))$ for some valid commitment of x_{n_i}. We can easily prove that u_1, v_1 corresponds to the root by opening v_1 and checking it is equal to the public commitment y_{root}.

The server must now prove that for each $i < d$, either:

- n_{i+1} is the left child of n_i, which means that $(y_{n_{i+1}} = y_{\mathsf{left}_{n_i}})$, or:
- n_{i+1} is the right child of n_i, which means that $(y_{n_{i+1}} = y_{\mathsf{right}_{n_i}})$.

To prove this, one uses a classic cut-and-choose proof: the server commits to a permutation of $y_{\mathsf{left}_{n_i}}$ and $y_{\mathsf{right}_{n_i}}$. Depending on the client's challenge, the server either proves that the two values were a correct permutation of the real values (this requires only showing equality, which is easy with homomorphic commitments), or proves that one of the values is $y_{n_{i+1}}$. Repeating this k times will lower the soundness error of the proof to 2^{-k}.

5.3.1 Complexity of the Proofs

One can see by inspection that the communication complexity of this proof is dominated by the proofs that $y_i \lhd x_i$. Each phase of the cut-and-choose protocol requires transmitting $O(tdk)$ bits, and so the overall communication complexity is $O(t^2dk^2)$ bits.

Round complexity. The protocol consists of a number of k-round cut-and-choose proofs. Because these proofs are not interdependent, we can run them all in parallel without losing zero-knowledge, so long as we use the same random coins for each of the proofs, i.e. at each round the client sends only a single challenge bit, which is used in all the proofs. (This is not true of ZK proofs in general, but it is true for our protocol.) Thus, we easily obtain a k-round protocol.

This can actually be improved substantially. As a first observation, we can collapse the k rounds of interaction together in order to obtain a 3-round protocol—that is, all challenges are sent simultaneously. This protocol is no longer provably zero-knowledge, but does retain *witness-indistinguishability* (this property is preserved by parallel repetition of protocols, see [10]).

Next, we can use various transformations to obtain a zero-knowledge proof.

5-round zero-knowledge based on perfect trapdoor commitments One can use standard folklore techniques to transform the 3-round, public coin witness-indistinguishable proof of knowledge into a ZK proof of knowledge. This increases the complexity to 5 rounds, and requires an additional assumption of perfectly hiding trapdoor commitment schemes (which exist based on the discrete log assumption and the hardness of factoring). In the first round, the server sends the parameters for a perfectly-hiding trapdoor commitment scheme. The client responds with a commitment to the challenges he will use in the protocol. They then run the 3-round protocol, using the committed challenges. Along with his response to the challenges, the server sends the trapdoor information for the commitment scheme.

Non-interactive zero-knowledge based on a random oracle If a random oracle is available, then we can in fact use the Fiat-Shamir technique to remove interaction completely without losing zero-knowledge, since our underlying proofs require only public coins. The idea is to replace the verifier's challenges with the output of the call to the random oracle on the first message of the protocol. We refer the reader to [11] for a discussion of the transformation and its limitations (in the context of signature schemes).

As a final note, it is *not* sufficient to transform our protocol to obtain a zero-knowledge proof of the existence of a witness – since the commitments involved are only computationally sound, a proof of knowledge is necessary.

5.4 Efficient Privacy for Range Queries—Theorem 8

Given the efficient consistent query protocols for join queries described in Section 3 and Section 4, privacy can be achieved by applying generic witness-indistinguishable or zero-knowledge proofs of knowledge, as described in Section 5.1. However, even for our efficient protocols these will be very complex, as they will require as the least oblivious evaluation of the circuit for hash function H.

Instead, we present efficient, private consistent query protocols for 1-D range queries, based on the explicit-hash technique of Section 5.3. The main drawback is that our protocol is not memoryless: the server must remember what queries have been made so far in order to ensure that no information is leaked from a proof.

The main tool used in the construction is a sub-protocol which, given commitments to values $C(a)$ and $C(b)$, allows the server to prove that $a < b$.

The first step is to modify the range tree so that *all* consistency proofs have length exactly $d = \lceil \log N \rceil$. Subsequently, we show how to achieve privacy efficently for membership queries, and finally for range querires.

Modified range tree. We start from the basic consistent query protocol for membership and range queries, based on range trees. First we modify the data structure slightly so that the length of a proof of consistency can be calculated exactly from the number of data points returned on a given query. Specifically, we ensure that *all* consistency proofs have length exactly $d = \lceil \log N \rceil$, and that the ranges of the children of a node n form a partition of $[a_n, b_n]$ about the splitting point $split_n$.

- Instead of storing at each internal node n the minimum and maximum keys which appear in the subtree rooted at that node, we store a larger interval $[a_n, b_n]$, which nonetheless has the property that all keys key in the subtree satisfy $a_n < \mathsf{key} < b_n$.

  At each branching we require that the children's intervals partition that of their parent, and the point at which they cut the parent's interval is stored at the parent and denoted $split_n$. Thus, the consistency check of Algorithm 1 becomes $a_l < b_l = split_n = a_r < b_r$. If n is a leaf, the consistency check becomes $a_n < \mathsf{key}_n < b_n$.

- For simplicity, we assume that keys are all integers in a known interval $\{1, ..., 2^s - 2\}$. The values $0, 2^s - 1$ are set aside as special values, denoted $-\infty$ and ∞, respectively.

- In order to ensure that it is always possible to split intervals so that $a_n < \mathsf{key}_n < b_n$ at the leaves, we can require that all keys be even numbers (this at most increases the size bound s by 1).

- In every tree, we insert the values $-\infty + 1 = 1$ and $\infty - 1 = 2^s - 2$, so that the range stored at the root is always in fact $[-\infty, \infty]$.

- We assume that the number of leaves in the tree is a power of 2 so that all leaves are at the same depth. This means $N = 2^d - 2$ for some integer d. This at most doubles the number of points we must store in the database.

The consistency proof for a membership query in this new structure will always consist of exactly d nodes (where $N = 2^d - 2$), even for queries which return "key not present". Consistency proofs for range queries comprise $m + 2d$ nodes, where m is the number of data points in the range.

Privacy for membership queries. We first describe how to achieve privacy for membership queries, and then explain how to generalize the technique for range queries.

The protocol outline is the same as for explicit hashing, except that additional range information is stored at the internal nodes. However, in the case of range trees the proof that the path is consistent is considerably more complex, since it involves proving statements of the form $a < b$.

Server storage. This is the same as in the explicit hashing protocol, except that the string x_n contains additional information: for internal nodes it contains a_n, b_n and $split_n$. For leaves, we add the range a_n, b_n, plus the values key_n and value_n (note that for efficiency, value_n can be the hash of the value stored at the leaf).

Moreover, all the range bounds are committed to *bit-by-bit* instead of as a monolithic string. This will be necessary to get fast consistency checks. If all keys are integers less than 2^s, then each number will require sk bits to be committed.

Proving $y_i \lhd x_i$. As before, the server commits to nodes and their hash values via d pairs u_i, v_i. The goal is to prove that these correspond to pairs x_i, y_i where $y_i \lhd x_i$. This is where the protocol requires the server to have memory. As before, the server will send a set of possible hash pre-images for the nodes in the path, and prove that each node in the path corresponds to at least one of these hash pre-images. The problem lies in choosing that set of possible hash pre-images. If the server reveals only those necessary for this path, then two different queries will reveal a lot about how the two different paths overlap. Instead, the server will always send all of the pre-images sent on the previous query, plus d new pre-images (regardless of how many new pre-images are really necessary). Thus, on the t-th query, the server sends td possible pre-images, and runs the same cut-and-choose protocol to show that the committed pairs satisfy $y_i \lhd x_i$.

Proving that the path is consistent. We now have pairs of commitments u_i, v_i which hide valid pairs x_{n_i}, y_{n_i}. We can easily prove that u_1, v_1 correspond to the root by opening v_1 and checking it is equal to the public commitment y_{root}. The basic check which must be performed are essentially the same as in Section 5.3, except that now we must add checks of the form $a < b$. We will show how to prove such statements below. First, we give the outline of the consistency checks.

Suppose that we have a subprotocol for proving that $a < b$ or $a \leq b$ given two commitments $C(a)$ and $C(b)$. Then the server can prove that the path consistent as follows:

- For each $i < d$, we have $a_{n_i} < split_{n_i} < b_{n_i}$.
- For each $i < d$, either:
  - n_{i+1} is the left child of n_i, which means that $(a_{n_{i+1}} = a_{n_i})$ and $(b_{n_{i+1}} = split_{n_i})$ and $(y_{n_{i+1}} = y_{\mathsf{left}_{n_i}})$, or:
  - n_{i+1} is the right child of n_i, which means that $(a_{n_{i+1}} = split_{n_i})$ and $(b_{n_{i+1}} = b_{n_i})$ and $(y_{n_{i+1}} = y_{\mathsf{right}_{n_i}})$.

  To prove this, one uses a classic cut-and-choose proof: the server commits to a permutation of $(a_{n_i}, split_{n_i}, y_{\mathsf{left}_{n_i}})$ and $(split_{n_i}, b_{n_i}, y_{\mathsf{right}_{n_i}})$. Depending on the client's challenge, the server either proves the two triples were a correct permutation of the real values (this requires only showing equality, which is easy with homomorphic commitments), or proves that one of the two triples is equal to $(a_{n_{i+1}}, b_{n_{i+1}}, y_{n_{i+1}})$.

  Repeating this k times will lower the soundness error of the proof to 2^{-k}.
- For the leaf $l = n_d$, we have $a_l < key_l < b_l$.
- For the leaf $l = n_d$, the revealed query answer is correct. If the query was for value key, we must check that $a_l < \mathsf{key} < b_l$ and either $\mathsf{key} = \mathsf{key}_l$ or $\mathsf{key} \neq \mathsf{key}_l$, depending on whether the query answer was positive or negative.

Thus, we need only show how to prove that $a < b$, $a \leq b$ or $a \neq b$) for two committed values $C(a), C(b)$.

Proving $a < b$, $a \leq b$, $a \neq b$. Suppose we have $C(a), C(b)$ for two integers $a, b \in \{0, ..., 2^s - 1\}$. The server wishes to prove to the client that $a < b$. A proof of the statement $a \leq b$ would proceed similarly. The proof that $a \neq b$ is in fact much easier and we leave it as an easy exercise.

1. Let $a_1, ..., a_s$ be the binary representation of a and $b_1, ..., b_s$ be the binary representation of b. Because we asked that the server commit bit-by-bit, we have $C(a_1), ..., C(a_s)$ and $C(b_1), ..., C(b_s)$.

2. Let $C'()$ be a commitment scheme which allows one to commit to one of three values $\{0, 1, *\}$. We only require that it be easy to prove that two commitments are equal. [10]

   Suppose that the first t most significant bits of a and b are equal. Then the server sends fresh commitments to the bits of a and b, except that for the first t bits of each he commits to $*$ instead.

   The problem of verifying that $a < b$ can now be reduced to one of local pattern checking. There are four sequences of committed bits. It must be that $*$'s appear in the two last sequences only when the bits of a, b are equal, and in all other positions the bits are copied faithfully. Moreover, it must be that the first position where $*$'s do not appear has $a_i = 0$ and $b_i = 1$. This means we must check $2s$ patterns, each on four positions.

   However, pattern checking can be done with a cut-and-choose protocol: the server commits to a permutation of all the possible patterns which apply to a given subset of bits (in our setting, there are always less than 20 patterns). Then he either opens all the patterns, or shows that one of them matches the positions he is checking. Repeat k times for soundness error 2^{-k}.

Achieving privacy for range queries. In order to achieve privacy for range queries, we build on the protocol above for membership queries. For each point in the range of the query, the server gives a proof of membership as above. For the two endpoints, the server gives an almost-complete proof of membership: he gives a path to the unique leaf which contains that endpoint, but does not prove any relation between the endpoint and the key at that leaf. Instead, he proves that the answers he has given cover the entire range:

1. The leaves in the range should be contiguous. This can be proven easily by proving $b_l = a_{l'}$ for adjacent leaves l, l'.

2. The endpoints should be proven correct. Suppose the query interval is $[a, b]$. Let l be the leaf corresponding to the left endpoint a. Let l' be the leaf corresponding to the leftmost point in the range. The left endpoint is correct if either

   - $a_l = a_{l'}$ and $a_l < a \leq \mathsf{key}_l$, or
   - $b_l = a_{l'}$ and $\mathsf{key}_l < a \leq b_l$

   This can be proven by a cut-and-choose as before.

   The proof of correctness of the right endpoint is similar.

Note that one can save some of the complexity of the membership proofs by running all the proofs that the various paths are in the hash tree together (see below).

[10] This can be implemented by having each commitment be a pair of bit commitments, where a commitment to $0, \beta$ represents the bit β and a commitment to $1, \beta$ always represents $*$.

5.4.1 Complexity of the Proofs

The communication complexity of the proof of membership can be seen by inspection to be $O(t \cdot d \cdot s \cdot k^2)$, where t is the number of queries so far, d is the depth of the hash tree ($= \log N$), s is the bound on the length of the keys, and k is the security parameter.

As for range queries, the complexity of the proofs can be made $O\left((t+m) \cdot d \cdot s \cdot k^2\right)$, where t is the number of queries so far and m is the total number of points returned from all queries so far. Note: The protocols of Micali et al. [20] for *membership* queries are more efficient than the protocol above. However, their techniques do not generalize to range queries.

Round complexity As in the discussion of explicit-hash Merkle trees, we can obtain *witness-indistinguishability* with a 3-round, public coin protocol, *zero-knowledge* by increasing the complexity to 5 rounds, and we can remove all interactivity if we assume the existence of a random oracle.

Acknowledgements

We thank Leo Reyzin and Silvio Micali for helpful discussions.

References

[1] B. Barak and O. Goldreich. Universal Arguments. In *Proc. Complexity (CCC) 2002*.

[2] J. L. Bentley. Multidimensional divide-and-conquer. *Comm. ACM*, 23:214–229, 1980.

[3] R. Canetti, D. Micciancio and O. Reingold. Perfectly One-Way Probabilistic Hash Functions. In *STOC 1998*, pp. 131-140.

[4] A Buldas, P. Laud and H. Lipmaa. Eliminating Counterevidence with Applications to Accountable Certificate Management. *J. Computer Security*, 2002. (Originally in *CCS 2000*.)

[5] A. Buldas, M. Roos, J. Willemson. Undeniable Replies to Database Queries. In *DBIS 2002*.

[6] I. B. Damgård, T. P. Pedersen, and B. Pfitzmann. On the existence of statistically hiding bit commitment schemes and fail-stop signatures. In *CRYPTO '93*, pp. 22–26.

[7] A. De Santis and G. Persiano Zero-Knowledge Proofs of Knowledge Without Interaction (Extended Abstract). In *Proc. of FOCS 1992*, pp. 427-436.

[8] P. Devanbu, M. Gertz, C. Martel, S. Stubblebine. Authentic Third-party Data Publication. In *DBSec 2000.*, p. 101–112.

[9] Y. Dodis and J. Hea An. Concealment and Its Applications to Authenticated Encryption. In *EUROCRYPT 2003*, May 2003.

[10] O. Goldreich. Foundations of Cryptography, Vol. 1. Cambridge University Press, 2001.

[11] S. Goldwasser and Y. Tauman. On the (In)security of the Fiat-Shamir Paradigm. In *FOCS* 2003.

[12] J. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, 1997.

[13] M. T. Goodrich, R. Tamassia, N. Triandopoulos and R. Cohen. Authenticated Data Structures for Graph and Geometric Searching. In *Proc. RSA Conference, Cryptographers' Track*, 2003.

[14] S. Halevi and S. Micali. Practical and provably-secure commitment schemes from collision-free hashing. In *CRYPTO '96*, p. 201–215.

[15] A. Healy, A. Lysyanskaya, T. Malkin, L. Reyzin. Zero-Knowledge Sets from General Assumptions. Manuscript, March 2004.

[16] J. Kilian. A note on efficient zero-knowledge proofs and arguments. In *24th STOC*, 1992.

[17] J. Kilian. Efficiently committing to databases. Technical report, NEC Research, 1998.

[18] P. Maniatis and M. Baker. Authenticated Append-only Skip Lists. ArXiv e-print cs.CR/0302010, February, 2003.

[19] C. Martel, G. Nuckolls, M. Gertz, P. Devanbu, A. Kwong, S. Stubblebine. A General Model for Authentic Data Publication. Manuscript, 2003. http://www.cs.ucdavis.edu/~devanbu/files/model-paper.pdf.

[20] S. Micali, M. Rabin and J. Kilian. Zero-Knowledge Sets. In *Proc. FOCS 2003*.

[21] S. Micali. Computationally Sound Proofs. *SIAM J. Computing*, 30(4):1253–1298, 2000.

[22] S. Micali and M. Rabin. Accessing personal data while preserving privacy. Talk announcement (1997), and personal communication with M. Rabin (1999).

[23] D. Micciancio. Oblivious data structures: applications to cryptography. In *Proc. STOC 1997*.

[24] R. Merkle A digital signature based on a conventional encryption function. In *CRYPTO '87*, pp. 369–378, 1988.

[25] M. Naor and K. Nissim. Certificate Revocation and Certificate Update. In *7th USENIX Security Symposium*, 1998.

[26] M. Naor, M. Yung. Universal One-Way Hash Functions and their Cryptographic Applications. In *21st STOC*, 1989.

[27] R. Ostrovsky, C. Rackoff, A. Smith. Efficient Consistency Proofs on a Committed Database MIT LCS Technical Report TR-887. Feb 2003. See http://www.lcs.mit.edu/publications

[28] T.P. Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *CRYPTO '91*.