Efficient Consistency Proofs on a Committed Database

Rafail Ostrovsky[*] Charles Rackoff[†] Adam Smith[‡]

February 27, 2003

Abstract

A *consistent query* protocol allows a database owner to publish a very short string c which *commits* her to a particular database D with special consistency property (i.e., given c, every allowable query has unique and well-defined answer with respect to D.) Moreover, when a user makes a query, any server hosting the database can answer the query, and provide a very *short proof* π that the answer is well-defined, unique, and consistent with c (and hence with D). One potential application of consistent query protocols is for guaranteeing the consistency of many replicated copies of D—the owner can publish c, and users can verify the consistency of a query to some copy of D by making sure π is consistent with c. This strong guarantee holds even for owners who try to cheat, while creating c.

The task of consistent query protocols was originally proposed for *membership* queries by Micali and Rabin[18], and subsequently and independently, by Kilian [16]. In this setting a server can prove to a client whether or not a given key is present or not in a database, based only on a short public commitment c.

We strengthen their results in several ways. For membership queries, we improve the communication complexity; more importantly, we provide protocols for more general types of queries and more general relational databases. For example, we consider databases in which entries have several keys and where we allow *range* queries (e.g. we allow a client to ask for all entries within a certain age range *and* a certain salary range).

Towards this goal, we introduce query algorithms with certain inherent robustness properties—called *data-robust algorithms*—and show how this robustness can be achieved. In particular, we illustrate our general technique by constructing an efficient data-robust algorithm for proving consistency of orthogonal range queries (a particular case of a "join"query). The server's proof convinces the client not only that all the matching entries provided are in D, but also that *no others are present*. Our guarantees hold even if the answer is the empty set. In the case of one-dimensional range queries we also show a new data-hiding technique—called *explicit hashing*—which allows us to a execute consistent query protocol π and at the same time protect the privacy of all other information in the database *efficiently*. In particular, we avoid the NP reductions required in a generic zero-knowledge proof.

[*]Telcordia Technologies, Morristown, NJ, USA.

[†]University of Toronto, Toronto, Ontario, Canada

[‡]MIT LCS, 200 Technology Square, NE43-446, Cambridge, MA 01239, USA.

1 Introduction

MERKLE TREES. Merkle [19] proposed the following protocol for committing to a list of n values $a_1, ..., a_N$: Pick a collision-resistant hash-function[1] H, pair up inputs $(a_1, a_2), (a_3, a_4), \ldots, (a_{n_1}, a_N)$ 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, the entire collection of values is now committed (though not necessarily hidden). To reveal any particular value a_i, one can reveal a path from the root to a_i together with all the siblings along the path. Very often, one must actually reveal a subset of the committed values a_i. The advantage of Merkle trees is that as long as the number of leaves revealed is small, the total *communication complexity* of revealing is proportional to the number of leaves revealed times $k \log N$, which in many settings is considerably smaller than N. Indeed, this idea is used throughout modern cryptography, including efficient signature schemes [19, 8], efficient zero-knowledge arguments [15], computationally-sound proofs [17], and many other applications.

CONSISTENT QUERY PROTOCOLS. One significant application of Merkle trees (with additional machinery to enforce consistency) is to proving "consistency of queries' to a committed database. (The topic of commitment protocols (especially efficient ones, has received a lot of attention in the literature, and we build upon that previous work [20, 22, 23, 15, 21, 8, 17, 18, 27, 14, 16, 6, 10, 7]). Especially relevant to our work is the notion of "consistent query commitment" protocol, originally proposed by Micali and Rabin [18], and, subsequently, by Kilian [16] for *membership* queries on a single (key, value) pairs database: suppose there is a server who hosts a very large database which is a collection of $(\mathsf{key}, \mathsf{value})$ pairs. The server produces a small commitment to that database which is then made public[2]. Any time a client asks the database a *membership query* (i.e. "do you have an entry with key x?"), the server returns the answer to the query along with a short proof of consistency with the public commitment. For each key, there should be a unique answer for which the server can provide a proof. An answer could either be "Yes, and the corresponding value is y." or "No, there is no entry with key x". We call a scheme for this task a *consistent query protocol* for membership queries. Both [18] and [16] give efficient solutions assuming only the existence of a collision-free hash function. Not surprisingly, the main constructions of [18, 16] are based on Merkle trees.

MAIN PROBLEM CONSIDERED. In this paper, we consider consistent query protocols for databases in which entries have several incomparable keys (for example, "age", "salary", "rank", etc). The class of queries we consider are joins (i.e. intersections) of range queries on several coordinates. For example, we allow a client to ask whether there is an entry in the database within a certain age range and a certain salary range. The main result of the paper is a novel and efficient commitment scheme that allows such efficient join queries (often called *orthogonal range queries*). The scheme guarantees that the server always answers consistently with a single database. He can prove *both* that all the hits he provides are in the database, *and that no others are present*. We also provide a general framework for constructing consistent query protocols based on query algorithms which are robust against corrupted data. (We remark that our technique is very general and

[1]Recall that a hash function family $H_k(\cdot)$ is called *collision-resistant* if no poly-time algorithm can find a pair of inputs that map to the same output, for k sufficiently large (see Section 2).

[2]This might be done by having the commitment signed by a certification authority, by publishing the commitment along with the server's public key, or by notarizing and time-stamping the commitment.

is applicaible to more general relational databases as well.)

AN APPLICATION. One interesting use of consistent query protocols is in replication of databases. A database owner may publish the short commitment using some reliable but expensive means. A server hosting a copy of the database could then prove the correctness of an answer to a query. Note that the scheme protects even against a malicious database owner—these protocols, in particular, prevent database owners from providing different users with different answers to the same query.

PRIVACY. A natural additional requirement is the server's privacy: informally, a query protocol is *private* if the proof of consistency reveals nothing to the client about the database, beyond what he learns from the answers to his query (and possibly an upper bound on the size of the database). Protecting the contents of the database could be crucial in settings where its contents are sensitive, or where clients are charged for access on a per-query basis.

Both the protocols of Micali-Rabin and Kilian [18, 16] can ensure privacy using standard zero-knowledge techniques. They can be made even more efficient by tailoring the cryptography to the specifics of membership queries.

EFFICIENCY CONSIDERATIONS. In the setting of committed databases, there are essentially two important measures of efficiency: On one hand, the total communication should be low: both the commitment and the proof of consistency should be small compared to the size of the database. On the other hand, the server's and verifier's computations should be efficient: ideally, they should be on roughly the same order as the communication of the protocol. There exists a general — but very inefficient in terms of the server's computation — way to construct consistent query protocols: simply have the server commit to the whole database. When a client sends a query, the server provides the answer along with a zero-knowledge argument of knowledge (ZKAK) that the answer is consistent with the commitment. A ZKAK (interactively) convinces the recipient of a statement's truth without revealing any other information. While this scheme has theoretically very good communication complexity (i.e. $poly(\log N) \cdot poly(k)$, where N is the database size and k is the security parameter), these proofs require enormous (i.e. super-linear in N) computations on the part of the server, as it must construct a so-called *probabilistically checkable proof* [1, 2] for the language of valid consistency proofs (in which the witnesses have size at least N, requiring $poly(N)$-time computations). To achieve practical schemes, we consider more efficient solutions, tailored for this problem.

1.1 Our contributions

CONSISTENT QUERY PROTOCOLS. We show a novel consistent query protocol scheme that allows efficient orthogonal range queries to a database with one, two or more keys associated to each database entry. The size of the commitment and the communication complexity of any join query is much smaller than the size of the database, and for any such query the database can answer (with a correct proof) in only one way. In particular, the server cannot omit any hits from the answers without being detected. The challenging part is to be able to prove that *no* points other than those in the answer actually appear in the database. gneralizing the work of [18, 16] we show how to do this efficiently (in particular, without having to decommit the entire database or resort to PCP proofs on the entire database).

Our consistency proofs have size $O(k(m + 1) \log^2 N)$, 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 of the server is low: in preprocessing, the server must make $O(N \log^2 N)$ evaluations of a collision-resistant hash function. For each query, the server's computation is on the same order as the communication with the client: $O(k(m+1)\log^2 N + \ell)$, where ℓ is the size of the answer to the query. For the case of *range* queries on a single key, our construction reduces essentially to that of [18, 16]. It produces query answers of size $O(k(m+1+\log N))$. However, in general for d-dimensional queries, we obtain consistency proof of size $O(k(m+1)\log^d N + \ell)$.

A GENERAL PARADIGM FOR CONSISTENT QUERY PROTOCOLS. In order to construct our protocol, we introduce the notion of *data-robust algorithms* (DRA). These are search algorithms which are robust against corruptions of the data by a *malicious* adversary: for any static data structure—even adversarially corrupted—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, since in general we do not want the algorithm to have to scan the entire data structure each time it is run—ideally, we want sublinear running time. Note that *the error model here is adversarial*: although much work in the algorithms and mathematics communities has focused on protecting data against randomly placed errors (or settings where the total number of errors introduced is bounded), the task of protecting against arbitrary malicious inputs is much more cryptographic in flavor.

The notion of a DRA has a significant application: assuming collision-resistant hash functions, any such algorithm can be transformed into a consistent query protocol whose (non-interactive) consistency proofs have complexity at most proportional to the complexity of the algorithm times the security parameter.

The consistent query protocol we give for range queries is obtained by first constructing a DRA based on range trees, a classic data structure due to Bentley [3]. Existing algorithms do not suffice, as inconsistencies in the data structure can lead to inconsistent query answers. Instead, we show how *local* consistency checks can be used to ensure that overall, queries are answered consistently with a single database. For two dimensional queries, the query time on correctly formed inputs is $O((m+1)\log^2 N)$, where m is the number of hits for the query and N is the number of keys in the database.

ACHIEVING PRIVACY EFFICIENTLY. Consistent query protocols will, in general, leak information about the database beyond the the answer to the query. As mentioned above, this problem can be solved by using generic constructions of zero-knowledge proofs, but one can get even greater efficiency by tailoring the protocol to the probem at hand. Given a consistent query protocol, one can transform it to be private by replacing the proof of consistency π with a zero-knowledge proof of knowledge of π. This adds interaction, but reduces the communication cost. For the protocols we consider, the cost is as low as $O(k, poly(\log\log N))$ (since N is polynomial in k, this is esentially $O(k)$). For very large databases and values of the security parameter, this yields very low communication complexity (see the end of Section 4 for details).

Although the asymptotics of this last scheme are good, the use of generic NP reductions and probabilistically checkable proofs means that the advantages only appear for extremely large datasets. We also construct tailored protocols for Merkle trees, which are simpler and more direct.

EXPLICIT-HASH MERKLE TREES. The Merkle tree commitment scheme sketched above may actually leak information about the committed values, since a collision-resistant function cannot hide all information about its input. At first glance, this seems easy to resolve: one can either replace the values a_i at the leaves of the tree with hiding commitments $C(a_i)$, or one can build the hiding property into the hash function itself by randomizing the hash (see, for example, the elegant commitment scheme of Halevi-Micali [14]).

However, there is often some additional structure to the values $a_1, ..., a_N$. For example, they might be stored in sorted order[3]. Revealing the path to a particular value would then reveal the rank of a given value 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.

One naïve solution to this problem is to provide a hiding commitment to the description of each node on the path, and then use a generic zero-knowledge proof (as above) that the committed string is consistent with the public hash value (the root of the hash tree). The main bottleneck of that approach is that it requires proving in zero-knowledge 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, especially when applied to a circuit as complex as a hash function. Indeed, at a first glance, 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.

Thus, the main challenge is to provide zero-knowledge proofs that a set $a'_1, ..., a'_t$ is a subset of the committed values, while leaving the hash function evaluations explicit, i.e. without going through oblivious evaluation of such complicated circuits. In this paper, we show that this is not a problem, and show 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*.

Theorem 1.1. *Assuming the existence of collision-free hash families and homomorphic perfectly-hiding commitment schemes,* explicit-hash Merkle trees *allow proving 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 can be made zero-knowledge with 5 rounds of interaction, witness-hiding with 3 rounds of interaction, and completely non-interactive if one assumes the availability of a random oracle.*

PRIVACY FOR RANGE QUERIES. As an application of explicit hash Merkle trees, we show to how to achieve privacy more efficiently for one-dimensional range queries (thus speeding up the protocols of Micali and Rabin [18] and Kilian [16]).

Theorem 1.2. *There exists an efficient* 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 is provably hiding with 3 rounds of interaction, and can also be made non-interactive in the random oracle model.*

More generally, we can make our higher-dimensional protocols private *private* at the cost of a polynomial blowup of the communication complexity, assuming the existence of trapdoor permutations and the availability of public randomness. Thus we obtain proofs of length $poly(k(m +$

[3]Jumping ahead, we will show one application where this property is crucial.

$1)\log N)$, which is still far smaller than N, the size of the database. One can gain even greater efficiency and security if we allow *interactive* consistency proofs. In that case, we can use the efficient proofs of [15] to get a private protocol with communication complexity $k\cdot(poly(\log(km) + \log\log N))$, which can be a substantial improvement in settings where the database is very large compared to the security parameter.

2 Definitions

We see say that a function $f(k)$ is *negligible* in a parameter k if for all integers $c > 0$, we have $f(k) \in O(\frac{1}{k^c})$. Given a algorithm A, we write $y \leftarrow A(x)$ to denote assigning the (possibly randomized) output of A on input x to variable y.

COLLISION-RESISTANT HASH FUNCTIONS. In our construction as as in those of [18, 16], the main cryptographic tool is collision-resistant hash functions (CRHF). This is a family of input-shrinking functions such that given a randomly chosen function h from the family, it is computationally infeasible to find a collision, i.e. two inputs x, y such that $h(x) = h(y)$. Such functions can be constructed assuming the hardness of the discrete logarithm or factoring. Formally, a family of (efficiently computable) 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 generator Σ such that for all PPT algorithms[4] $\mathcal{A}$, we have that $\Pr[s \leftarrow \Sigma(1^k); (x, y) \leftarrow \mathcal{A}(1^k, s) : h_{s,k}(x) = h_{s,k}(y)]$ is negligible in k.

For our constructions (as for those of [18, 16]), we will assume the availability of a public collision-resistant hash function. Formally, this means we assume that some trusted third party has chosen a hash function $h_{s,k}$ at random from the family (for some publicly agreed parameter k) and published the description of the hash function. In practice, one sometimes also uses a fixed hash function, such as SHA or MD5.

2.1 Consistent query protocols

To formalize the notion of consistent query protocols, we first define a query structure: this 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 $q \in \mathcal{Q}, D \in \mathcal{D}$. For example, in the case of simple membership queries, a valid database D is a set of pairs $\{(\mathsf{key}_1, \mathsf{value}_1), \ldots, (\mathsf{key}_n, \mathsf{value}_n)\}$ where no key appears twice. The set of possible queries is just the set of possible keys, and the rule $Q(\mathsf{key}, D)$ returns value_i if $\mathsf{key} = \mathsf{key}_i$ and a distinguished value $\perp$ otherwise.

In a basic consistent query protocol, 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. One can gain extra power if there is some public randomness which is provided by a trusted third party. While we formulate our definitions in the context of such a trusted third party, *we stress that in some settings our constructions*

[4]For simplicity we state our security definitions in the uniform model, but all the definitions can be stated equally well with respect to non-uniform adversaries

do not require the public randomness; we include it in this formulation, because in settings where such public randomness is available, one can achieve even stronger security properties.

Definition 1. A (non-interactive) *query protocol* consists of three 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, as well the public information $\sigma \leftarrow \Sigma(1^k)$. It produces a commitment c (which is made public), as well as 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). $\mathcal{C}$ outputs "accept" if it accepts the proof π and "reject" otherwise.

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

- **Completeness:** For every valid database $D \in \mathcal{D}$ and query $q \in \mathcal{Q}$, if $\sigma \leftarrow \Sigma(1^k)$ and $(c, \textit{state}) \leftarrow \mathcal{S}_s(\sigma, D)$ then $\mathcal{C}$ will accept (a, π) output by $\mathcal{S}_a(q, \textit{state})$ with overwhelming probability. Moreover, $a = Q(q, D)$ with probability 1. Formally, for all $q \in \mathcal{Q}$ and for all $D \in \mathcal{D}$ we have:

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

- **(Computational) Soundness:** For every PPT adversary[5] $\tilde{\mathcal{S}}$, run $\tilde{\mathcal{S}}$ on inputs 1^k and $\sigma \leftarrow \Sigma(1^k)$, and allow it to output a commitment c along with a (polynomially-long in k) 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. The protocol is *sound* if all PPT adversaries $\tilde{\mathcal{S}}$ act consistently with overwhelming probability whenever the $\mathcal{C}$ accepts all the proofs π_i. Formally, we require:

$$\Pr[\sigma \leftarrow \Sigma(1^k); \big(c, (q_1, a_1, \pi_1), \dots, (q_t, a_t, \pi_t)\big); b_i \leftarrow \mathcal{C}(\sigma, c, q_i, a_i, \pi_i) : \\ b_i = 1 \text{ for all } i \text{ and } \tilde{\mathcal{S}} \text{ acts consistently}] \leq negl(k)$$

Although the previous definitions are stated in terms of *non-interactive* consistency proofs, they generalize naturally to interactive proofs.

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

Remark 1 (Maintaining state). In general, a malicious server may maintain some state between various invocations of the query protocol. We stress, however, that our constructions do not *require* such state; the honest server is essentially stateless (remembering only its initial setup with the variable *state*). Furthermore, as long as the consistency proofs are non-interactive, we can even allow multiple concurrent invocations of the server while still maintaining security.

Remark 2 (Public information). As mentioned above, the trusted third party is only used in some of our constructions. In particular, it is really necessary only when we want to use non-interactive zero-knowledge proofs to achieve privacy (see below). In many of our settings, the only initial shared information information we will require is a public CRHF. As we shall see, *this function can be chosen by a representative of the clients*; it needn't be someone trusted by both parties. Moreover, if a certain fixed function (e.g. SHA) is "collision-resistant enough" for a given application, then we can dispense with initial shared information entirely.

Remark 3 (Hash functions are needed). In order to construct good consistent query protocols, a CRHF is not just helpful—it is necessary. If the size of the commitment to the database is smaller than the database itself, it is an easy exercise to prove that the computational soundness of the protocol implies the existence of collision-free hash functions.

2.1.1 Keyed databases

In practice, databases are often simply sets of $(\mathsf{key}, \mathsf{value})$ pairs, where the clients are restricted to asking for all pairs whose keys fall within some subset of the key-space. (In the example of membership queries, the subsets are just the singletons $\{\mathsf{key}\}$.) We call such databases *keyed databases*.

2.1.2 Privacy

Another property which may be useful (e.g. in settings in which query answers are sold individually, or in which the database contains personal data) is *privacy*. Namely, the answer to a query should reveal little or no information about possible answers to other queries. Thus the server is not giving any information away along with his proof of consistency. In this section we define two levels of privacy: one which hides all information about the database, and the other, specific to keyed databases, which hides only the *values* stored in the database, and not the associated keys. Our definitions of privacy follow those given by Kilian [16] in the context of membership queries.

TOTAL PRIVACY. Intuitively, a protocol is private if it reveals no information about queries other than those already asked by a client.

A simple way to formulate this is: suppose we have two databases D_1, D_2 and a set of queries $\{q_1, \ldots, q_m\}$ such that $Q(q_i, D_1) = Q(q_i, D_2)$ for all $i \in \{1, \ldots, m\}$. Then the distributions on the tuple $(c, \mathcal{S}(q_1), \ldots, \mathcal{S}(q_m))$, given that either (a) the server was initialized with database D_1 or (b) the server was initialized with database D_2, should be *computationally indistinguishable* (a more proper formalization of security allows the adversary *adaptive* access to $\mathcal{S}$ as an oracle). Unfortunately, achieving such a strong notion of privacy is problematic, because it is difficult not to reveal any information about the *size* of the database. What is meant by size can vary depending on the context: in general, it is simply the amount of space required to store the database. Thus *we require that the indistinguishability condition above hold only for databases of the same size.*

Formally, consider an adversary $\tilde{\mathcal{C}}$ who interacts with a server hosting either one of two databases $D_1, D_2 \in \mathcal{D}$. We say $\tilde{\mathcal{C}}$ is (D_1, D_2)-limited if $\tilde{\mathcal{C}}$ only asks queries q such that $Q(q, D_1) = Q(q, D_2)$. We denote by $\tilde{\mathcal{C}}^{\mathcal{S}_a(\sigma,\cdot,\textit{state})}(\sigma, c)$ the result of the interaction between $\tilde{\mathcal{C}}$ and the server with setup information *state*.

Definition 3 (Computational privacy). We say that a consistent query protocol $(\Sigma, \mathcal{S}_s, \mathcal{S}_a, \mathcal{C})$ for $(\mathcal{D}, \mathcal{Q}, Q)$ is private if for every two databases $D_1, D_2 \in \mathcal{D}$ of the same size, and for all PPT adversaries $\tilde{\mathcal{C}}$ which are (D_1, D_2)-limited, we have:

$$\Big| \Pr\left[\sigma \leftarrow \Sigma(1^k); (c, \textit{state}) \leftarrow \mathcal{S}_s(\sigma, D_1) : \tilde{\mathcal{C}}^{\mathcal{S}_a(\sigma,\cdot,\textit{state})}(\sigma, c) = 1\right] \\ - \Pr\left[\sigma \leftarrow \Sigma(1^k); (c, \textit{state}) \leftarrow \mathcal{S}_s(\sigma, D_2) : \tilde{\mathcal{C}}^{\mathcal{S}_a(\sigma,\cdot,\textit{state})}(\sigma, c) = 1\right] \Big| \leq negl(k)$$

The foregoing definition is stated in terms of *computational* privacy. By removing the restriction that $\tilde{\mathcal{C}}$ be polynomial time, we obtain statistical security; by further requiring that the probabilities of producing 1 on D_1 or D_2 be equal, we obtain a definition of perfect privacy.

Note that for keyed databases, a natural definition of "size" is the number of keys present in the database. The protocols we present in Section 5 can be made private with respect to this latter notion of size: for any two databases with the same number of keys, queries which return the same answer give no information on which of the two databases is actually being hosted by the server.

VALUE PRIVACY. In the context of keyed databases, some applications may not require the secrecy of the keys contained in the database, but only of the values associated to various keys. Specifically, a consistent query protocol is *value-private*[6] if the proofs associated to queries whose answers don't contain the key key_i reveal no information about the corresponding data value_i. What is meant by "no information" depends on whether we want computational, statistical or perfect secrecy: the distributions on conversations corresponding to different values of value_i should be perfectly (resp. statistically or computationally) indistinguishable, so long as the pair $(\mathsf{key}_i, \mathsf{value}_i)$ never appears as answer to a query.

One can formalize this as in Definition 3 above by restricting D_1 and D_2 to be databases with the same keys present, that is we allow D_1 and D_2 to differ only in the stored values which are not requested by the client.

As is pointed out in [18, 16], it is easy to se that value-privacy is easy to attain. If the values in the database are replaced by *non-interactive commitments* to those values, then any consistent query protocol easily becomes value-private: run the usual protocol, and when a key appears in the answer to a query, simply accompany the query answer with the appropriate de-commitment string. The resulting protocol is private because until he makes a query which returns key_i, the client will only ever see the commitment to value_i. The strength of privacy obtained depends on the type of commitment used. However, statistical value-privacy is easy to obtain, and very efficient: assuming the availability of a public collision-resistant hash function with output length k, one can construct a statistically-hiding, non-interactive commitment scheme (Halevi and Micali,

[6]This is called $D[t]$ privacy in [16].

[14]) with commitment length k and decommitments of length $\ell + 7k$, where ℓ is the length of the message being revealed.

INTERACTIVE PROOFS. One can extend the definition of consistency to a model where the proof may be interactive, and this will be very useful when we want to achieve privacy without trusting a third party to provide public random strings.

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[7]. This task is most interesting for structures which replicate information in order to allow more efficient queries.

Assuming the availability of a collision-resistant hash function, we show in Section 3.2 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).

We observe that several of the protocols of [18, 16] can be viewed in this light. We further illustrate the paradigm by constructing a consistent query protocol for *low-dimensional orthogonal range queries*. In this model the keys in the database consist of several components $(x_1, \ldots, x_d)$. The goal is to find all entries whose keys simultaneously satisfy a range constraint on each component, i.e. queries are rectangles of the form $[a_1, b_1] \times \cdots \times [a_n, b_n]$. We first modify a classic data structure due to Bentley [3] to make it data-robust. We then use it to get a consistent query protocol with commitment size k and proof size $O(k(m+1)\log^d N)$.[8]

In the next section, we formally define consistent query protocols and data-robust algorithms. We then describe a generic method for constructing consistent query protocols from DRA's. Finally, we describe the specific construction for range queries.

3.1 Data-robust algorithms

When considering consistency of queries, a natural problem is that of designing *data-robust algorithms*. Consider the setting where a programmer records a database on disk in some kind of static data structure which allows efficient queries. Such data structures are often augmented with redundant information, for example to allow searching on two different fields. If the data structure later becomes corrupted, then it could be that subsequent queries to the structure would be mutually inconsistent: for example, if entries are sorted on two fields, some entry might appear in one of the two structures but not the other.

To formalize the notion of data-robust algorithms, we first define a query structure: this 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

[7]Note that despite the algorithmic flavor of the question, the error model is indeed cryptographic (i.e. adversarial).

[8]Note that the most efficient normal algorithm for range queries has complexity $O((m+1)\log^{d-1} N)$ [13]. We do not know how to make it data-robust, however, without increasing the complexity by a factor of $\log N$.

which associates an answer $a_{q,D} = Q(q, D)$ with every query/database pair $q \in \mathcal{Q}, D \in \mathcal{D}$. [9]

Suppose we have a query structure $(\mathcal{D}, \mathcal{Q}, Q)$. A data-robust algorithm (DRA) for these consists of two polynomial-time[10] algorithms (T, A): First, a setup transformation $T : D \to \{0,1\}^*$ 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. Note that 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 $T^* : \{0,1\}^* \to \mathcal{D}$ such that for all (adversarially chosen) structures $\tilde{S}$, the database $D = T^*(\tilde{S})$ satisfies $A(q, \tilde{S}) = Q(q, D)$ for all queries q.

  (Note that 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 $S = T(D)$, the algorithm A returns the correct answer $Q(q, D)$.)

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

THE RUNNING TIME OF A. Note that there is a naive solution to the problem of designing a DRA: A could simply 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 algorithms which query *sub-linear* amounts of memory; the goal is then to maintain that efficiency while also achieving robustness. Note that in this setting, efficiency means the running-time of the algorithm A on *correct* inputs[11]. 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 significantly worse (such as setting up an infinite loop, etc).

ERROR MODEL. Although the design of DRA's seems to be an algorithmic question, the error model—that of *adversarially* placed errors—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 rate (witness the entire 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.

[9]For example, in the case of simple membership queries, a valid database D is a set of pairs $\{(\mathsf{key}_1, \mathsf{value}_1), \ldots, (\mathsf{key}_n, \mathsf{value}_n)\}$ where no key appears twice. The set of possible queries is just the set of possible keys, and the rule $Q(\mathsf{key}, D)$ returns value_i if $\mathsf{key} = \mathsf{key}_i$ and a distinguished value $\perp$ otherwise.

[10]We assume for simplicity that the algorithms are deterministic, though this is not strictly necessary.

[11]We assume either a RAM or a pointer-based memory model here; sublinear time is not very powerful when access to memory is sequential.

3.2 Constructing consistent query protocols from DRA's

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. Most common search algorithms fit into this model. Essentially, we obtain a consistent query protocol by creating a Merkle tree (or graph) which mimics the structure of the DRA's data structure.

Proposition 3.1. *Let (T, A) be a DRA for query structure $(\mathcal{D}, \mathcal{Q}, Q)$ which fits into the pointer-based framework described above. Suppose that on inputs q and $T(D)$ (correctly formed), the algorithm A examines $b(q, D)$ memory blocks and a total of $s(q, D)$ bits of memory, using $t(q, D)$ 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 $s(q, D) + kb(q, D)$ on query q. The server's computation on each query is $O(s(q, D) + t(q, D) + kb(q, D))$.*

The details of this statement and its proof form the remainder of this section.

3.2.1 Pointer-based algorithms

Given a DRA which works in a pointer-based memory model (to be specified below), we show how to transform it into a cryptographically secure consistent query protocol. In section Section 4, we show how to extend this protocol to gain greater efficiency, as well as privacy.

Specifically, 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 $\mathsf{get}(u)$. This returns the associated data data_u as well as a list of outgoing edges $v_{1,u}, v_{2,u}, \dots, v_{n_u,u}$.
   - 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, $S = b_1\# \dots \# b_n$. 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 simply be the first block by convention. [12]

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

[12]It should be stressed that 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

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 $\tilde{S}$.

3.2.2 A general construction

Let (T, A) be a DRA for query structure $(\mathcal{D}, \mathcal{Q}, Q)$ which fits into the pointer-based framework described above. Moreover, suppose that a correctly formed structure (i.e. an output of T) never contains a pointer cycle (that is, the resulting graph is acyclic)[13].

Proposition 3.2 (same as Proposition 3.1). *Suppose that on inputs q and $T(D)$ (correctly formed), the algorithm A examines $b(q, D)$ memory blocks and a total of $s(q, D)$ bits of memory, using $t(q, D)$ 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 $s(q, D) + kb(q, D)$ on query q. The server's computation on each query is $O(s(q, D) + t(q, D) + kb(q, D))$.*

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 simply 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)[14]. Now proceed from sinks to the source by adding a hash value (called h_u) at each node u: For a sink, simply attach the hash of its binary representation; this is basically $h_u = H(\mathsf{data}_u)$. When 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

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.

[13] This restriction is not necessary. One can handle general graphs at an additional logarithmic cost by superimposing a tree on the memory structure

[14] 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.

accessed by the algorithm (by looking at calls to the $\mathtt{get}(\cdot)$ 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 $(\mathtt{data}_u, h_{v_{1,u}}, \ldots, h_{v_{n_u,u}})$ 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 indeed a sequence of correctly formed "transformed" binary representations of memory blocks and 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 will be the case 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[15]. 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 Achieving privacy for the general construction

For a formal definition of privacy for consistent query protocols, see Section 2.1.2. Informally, we are interested in two kinds of privacy: *total privacy*, in which the client cannot distinguish between commitments to two different datbases without asking a question which distinguishes the two, and *value privacy*, in which we only wish that the data associated to the keys in the database remain secure. This latter notion is in fact quite easy to obtain, as we describe at the end of this section.

We can also extend the consistent query protocols of the previous section to provide *total privacy*, i.e. to protect against any extra information about the database leaking out through the proof of consistency. Instead of sending the consistency proof π, the server provides a witness-hiding proof of knowledge of π: this convinces the client both that such a π exists *and that the server knows it*. Interestingly, a simple proof of membership of the existence of π doesn't suffice: because the protocols in question are only computationally sound, it may be that consistency proofs

[15]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

exist for all sorts of invalid query answers; we can only rely on the assumption that for all but one answer, those proofs are (computationally) difficult to find.

Note that this solution is a *vast* improvement over the generic solution (described in the introduction). Whereas in the generic case the server must prove statements whose length is at least that of the *whole* database, here it must only prove a statement roughly as long as (the verification circuit for) the consistency proof π. In the case of the protocols of Section 5, this is a quasi-exponential improvement in efficiency, since the resulting proofs are poly-logarithmic in the size of the database.

Let $(\Sigma_{db}, \mathcal{S}_s, \mathcal{S}_a, \mathcal{C})$ be a consistent query protocol for some query structure $(\mathcal{D}, \mathcal{Q}, Q)$, such that the client (i.e. verification algorithm) $\mathcal{C}$ is *deterministic*. We assume that the commitment to the database is shorter than the database itself, and thus that a collision-free hash function is available, either as common knowledge to all parties or explicitly as part of the public information output by Σ_{db}.

Let C_{com} be the commitment function for the protocol of Halevi and Micali [14], i.e. C_{com} takes a message to commit and outputs the commitment along with the random coins used for commitment. If the message has length ℓ, the scheme uses 7ℓ random bits. Note that this protocol requires a publicly available hash function. However, in all constructions of consistent query protocols where the commitment is smaller than the database, a CRHF must already exist, so there is no need here to provide it explicitly. Also note that the decommitment information from this protocol is simply the committed message and the random coins used to commit; the verification consists of ensuring that running the commitment algorithm on the given coins and message does indeed yield the commitment.

Let $\mathcal{P}(\cdot,\cdot)$ be the (deterministic) polynomial-time relation given by

$$\mathcal{P}\big((\sigma_{db}, c', q, a), (\pi, c, \omega)\big) = 1 \quad \text{iff} \quad c' = C_{com}(c, \omega) \text{ and } \mathcal{C}((\sigma_{db}, c, q, a, \pi) = \text{“accept”}$$

The corresponding language $L_{\mathcal{P}}$ is the set of tuples for which there exists a proof of consistency:

$$L_{\mathcal{P}} = \big\{(\sigma_{db}, c, q, a) : \ \exists \pi : \mathcal{P}\big((\sigma_{db}, c, q, a), \pi\big) = 1\big\}$$

Let (Σ_{zk}, P, V) be an *adaptive, multiple-theorem, non-interactive, witness-indistinguishable proof of knowledge system* (NIZKPK) for the relation $\mathcal{P}$. See [9] for a definition of this primitive. Note that in our setting, we don't require that the public randomness used by the NIZKPK be uniform on all strings of a given length. Thus, such a system can be constructed based on the existence of any trapdoor permutation family. Of course, having the public randomness be simple coins is handy, and such proof systems exist as long as there exists a *dense cryptosystem* [9].

Consider a modified consistent query protocol $(\Sigma', \mathcal{S}'_s, \mathcal{S}'_a, \mathcal{C}')$ which uses the NIZKPK to prevent partial leakage of information:

- The public coin generation algorithm $\Sigma'(1^k)$ returns $(\Sigma_{db}(1^k), \Sigma_{zk}(1^k))$.
- The setup algorithm $\mathcal{S}'_s(\sigma_{db}, \sigma_{zk}, D)$ computes $(c, \textit{state}) \leftarrow \mathcal{S}_s(\sigma_{db}, D)$, and $(c', \omega) = C_{com}(c)$. The setup returns c' as the public commitment and $\textit{state}' = (c, c', \omega, \textit{state})$.
- The server's query algorithm $\mathcal{S}'_a(\sigma_{db}, \sigma_{zk}, q, \textit{state}')$ first computes $(a, \pi) \leftarrow \mathcal{S}_a(\sigma_{db}, q, \textit{state})$.

Next, it runs the NIZKPK prover[16] P: $\Pi \leftarrow P\big(\sigma_{zk}, (\sigma_{db}, c', q, a), (\pi, c, \omega)\big)$. It sends to the client the pair (a, Π).

- The client $\mathcal{C}'(\sigma_{db}, \sigma_{zk}, c', q, a, \Pi)$ simply verifies the NIZKPK: $\mathcal{C}'$ accepts iff $V(\sigma_{zk}, (\sigma_{db}, c', q, a), \Pi)$ accepts.

Lemma 4.1. *The query protocol $(\Sigma', \mathcal{S}'_s, \mathcal{S}'_a, \mathcal{C}')$ from the construction above is a consistent query protocol (Definition 2) which is private (definition Definition 3). The resulting protocol has communication complexity polynomial in the complexity of the original protocol.*

Proof. For brevity, we omit the details, as the properties of the protocol follow in a fairly straightforward manner from the properties of the NIZKPK system. There are some subtleties worth noting: (a) proofs of knowledge truly are necessary, since the statistically binding protcol of Halevi-Micali makes the language $L_{\mathcal{P}}$ basically trivial; (b) an adaptive mutliple-theorem NIZKPK is needed since an adversarial client may tailor his queries to the consistency proofs he has received in the past; (c) we need that the length of the proofs π from the original protocol reveal nothing beyond the length of the database. For protocols with an easy-to-calculate upper bound on the proof length over all databases of a given size, this can be accomplished by padding the proof π out to the appropriate length. The protocols constructed in this paper satisfy this property. $\square$

The protocols of the previous section do indeed have deterministic verifiers, and so we can apply the construction above. 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. Moreover, suppose that a correctly formed structure (i.e. an output of T) never contains a pointer cycle.

Proposition 4.2. *Suppose that on inputs q and $T(D)$ (correctly formed), the algorithm A examines $b(q, D)$ memory blocks and a total of $s(q, D)$ bits of memory, using $t(q, D)$ time steps. Assuming the availability of a public collision-resistant hash function, and the existence of trapdoor permutations, there exists a non-interactive,* private *consistent query protocol for $(\mathcal{D}, \mathcal{Q}, Q)$ which has proof length $poly(s(q, D) + kb(q, D))$ on query q. The server's computation on each query is $poly(s(q, D) + t(q, D) + kb(q, D))$.*

Note that when the original protocol yields proofs polylogarithmic in the size of the database, then so does the modified, private protocol.

EFFICIENT INTERACTIVE PROOFS. More generally, using the efficient ZKPK's of [15], one can prove statements of length n with communication $O(k \log^c n)$, for some constant c, assuming the availability of a collision-resistant hash function. The server-side computation is polynomial in n. Thus our consistent query protocol can be made private, and the resulting communication is $O\left(k \log^c \left(s(q, D) + kb(q, D)\right)\right)$. The drawback to this approach is that the proof of consistency becomes interactive.

VALUE PRIVACY. As mentioned in Section 2.1.2, achieving the more limited *value privacy* (in the case of keyed databases) is potentially more efficient: using the commitment scheme of Halevi and Micali [14], we can get statistical value-privacy at an additional cost of only $7k$ bits per value

[16]In fact, the proof π must be padded out so that all consistency proofs have the same length—this way no information is revealed beyond the size of the database.

which must be revealed. Thus the total communication is bounded above by $s(q,D) + 8kb(q,D)$. Using the Halevi-Micali scheme in this context has several advantages: it requires no additional information or infrastructure, since the hash function is required for the basic protocol. Moreover, the resulting protocol is also computation-efficient: the computation required for the commitment is two evalutions of the hash function. To decommit, one simply needs to reveal the message the random coins used in the commitment.

5 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; our construction naturally extends to higher dimensions (Section 5.2). 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.

5.1 A data-robust algorithm for range queries

Various data structures for efficient orthogonal range queries exist (see [13] for a survey). The most efficient (non-robust) solutions have query time $O((m+1)\log^{d-1} N)$ for d-dimensional queries. In this section we recall the construction of *multi-dimensional range trees* (due to Bentley [3]), 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.

5.1.1 One-dimensional range trees

Multidimensional range trees are built recursively from one-dimensional range trees (denoted 1-DRT), which were (essentially) one of the structures used by [18, 16] for membership queries. 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 (denoted a_n) and maximum (denoted b_n) keys which appear in the subtree rooted at n. 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 an easy exercise to show that a 1-DRT allows efficient range queries when it is correctly formed[17]. However, in our setting we must also ensure that the queries return

[17]Given the root n of a tree and a target interval $[a, b]$, descend recursively to those children whose intervals overlap with $[a, b]$.

```
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

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.

The algorithm A_{1DRT} (Algorithm 1, Figure 1) will query a $\mathsf{1\text{-}DRT}$ robustly. When it is first called, the argument n will be the root of the graph. Essentially, A_{1DRT} 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, there must be that the graph the algorithm receives as input is not a tree).

Note that in fact, the algorithm A_{1DRT} operates in the "pointer-based" model of Section 3.2.1. Thus the first node on which the algorithm is called is obtained through a call to $\mathsf{getroot}()$. 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 $\mathsf{get}(\cdot)$ in the description of the algorithm.

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

Lemma 5.1. *The algorithm A_{1DRT} 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 with 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 *in* 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 $\tilde{S}$. □

```
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 $a_n = a_{\mathsf{left}_n} \leq b_{\mathsf{left}_n} < a_{\mathsf{right}_n} \leq b_{\mathsf{right}_n} = b_n$

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 $\mathsf{left}_n$ or $\mathsf{right}_n$.
           If the check fails, remove $p$ from $B$.
         • Return $B$
   (c) Otherwise
         • $B \leftarrow A_{\mathsf{2DRT}}\Big(\ ([a^{(x)}, b^{(x)}] \cap [a_{\mathsf{left}_n}, b_{\mathsf{left}_n}]) \times [a^{(y)}, b^{(y)}],\ \mathsf{left}_n\Big)$
               $\cup\ A_{\mathsf{2DRT}}\Big(\ ([a^{(x)}, b^{(x)}] \cap [a_{\mathsf{right}_n}, b_{\mathsf{right}_n}]) \times [a^{(y)}, b^{(y)}],\ \mathsf{right}_n\Big)$
         • Remove elements of $B$ which are not valid leaves of $\mathsf{tree}_n$.
         • Return $B$
```

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

5.1.2 Two-dimensional range trees

SETUP. 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. Note that it need *not* store the value associated to an $(\mathsf{xkey}, \mathsf{ykey})$ pair.

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 $O(\log^2 N)$ [13]: 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.
- Additionally, 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 the notion of valiity which is enforced by these checks:

Definition 7. A point $p = (\mathsf{xkey}, \mathsf{ykey}, \mathsf{value})$ in a (possibly 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) $\mathsf{2\text{-}DRT}$ 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.

For robust range queries, the algorithm A_{2DRT} we obtain is described in Figure 2. 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 5.2. *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 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. $\square$

Remark 4. As mentioned above, more efficient data structures and algorithms for planar orthogonal queries exist [13], 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.

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

Lemma 5.3. *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.*

5.2 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). Note that 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. Note that 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 5.4 (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).*

The protocol can be made statistically value-private by at an increased cost of $7km$ bits of communication. The protocol can be made perfectly private. If non-interactive proofs are desired, then we obtain proofs of length $poly(k(m+1)\log N)$, at the cost of requiring public randomness. If we allow interactive proofs, then the resulting communication is $O(k\log^c(k(m+1)) + k\log^c \log N)$ for some constant c.

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

6 Explicit-hash Merkle trees

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 efficienty, by showing a path from the root to that particular value. The goal is 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. In this section we describe the construction of explicit-hash, private Merkle trees.

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 $m_1 + m_2$ ([18]). Such schemes exist based on a number of assumptions, such as the hardness of discrete logarithm extraction (e.g. Pedersen's scheme [25]). 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

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

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 $y = H(C(x; \omega)$.

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 o 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.

The specification and analysis of the protocol, which essentially proves Theorem 1.1, is contained in Appendix A.

6.1 Achieving Privacy More Efficiently

Given the efficient consistent query protocols for join queries described in Section 3 and Section 5, privacy can be achieved by applying generic witness-hiding or zero-knowledge proofs of knowledge, as described in Section 4. 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 6. 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 protocol is specified in Appendix B.

References

[1] S. Arora and M. Safra. Probabilistic Checking of Proofs: A New Characterization of NP. Journal of ACM, 45(1):70–122, 1998.

[2] S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy. Proof Verification and Hardness of Approximation Problems. Journal of ACM, 45(3):501-555, 1998.

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

[4] R. Canetti, O. Goldreich and S. Halevi. The Random Oracle Methodology, Revisited (preliminary version). In *Proceedings of STOC 1998*, pp. 209–218.

[5] R. Cramer and V. Shoup. A Practical Public Key Cryptosystem Provably Secure Against Chosen Ciphertext Attack. CRYPTO '98.

[6] G. Di Crescenzo, Y. Ishai, and R. Ostrovsky. Non-Interactive and Non-Malleable Commitment. STOC '98.

[7] G. Di Crescenzo, J. Katz, R. Ostrovsky, A. Smith: Efficient and Non-interactive Non-malleable Commitment. EUROCRYPT 2001: pp. 40-59

[8] I. B. Damgård, T. P. Pedersen, and B. Pfitzmann. On the existence of statistically hiding bit commitment schemes and fail-stop signatures. In D. R. Stinson, editor, *Advances in Cryptology—CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 250–265. Springer-Verlag, 22–26 Aug. 1993.

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

[10] M. Fischlin and R. Fischlin. Efficient Non-Malleable Commitment Schemes. CRYPTO 2000.

[11] S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract). In *Proc. of STOC 1985*, pp. 291–304.

[12] O. Goldreich and S. Micali and A. Wigderson. Proofs that Yield Nothing But their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *JACM*, 38 (1), pp. 691–729, 1991.

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

[14] S. Halevi and S. Micali. Practical and provably-secure commitment schemes from collision-free hashing. In N. Koblitz, editor, *Advances in Cryptology—CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 201–215. Springer-Verlag, 18–22 Aug. 1996.

[15] J. Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 723–732, Victoria, British Columbia, Canada, 4–6 May 1992.

[16] J. Kilian. Efficiently committing to databases. Technical report, NEC Research Institute, February 1998.

[17] S. Micali. CS proofs (extended abstract). In *35th Annual Symposium on Foundations of Computer Science*, pages 436–453, Santa Fe, New Mexico, 20–22 Nov. 1994. IEEE.

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

[19] R. Merkle A digital signature based on a conventional encryption function. In C. Pomerance, editor, *Advances in Cryptology – CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, 16–20 August 1987. Springer-Verlag, 1988.

[20] M. Naor. Bit commitment using pseudo-randomness (extended abstract). In G. Brassard, editor, *Advances in Cryptology—CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 128–136. Springer-Verlag, 1990, 20–24 Aug. 1989.

[21] M. Naor, R. Ostrovsky, R. Venkatesan, M. Yung: Perfect Zero-Knowledge Arguments for NP Can Be Based on General Complexity Assumptions (Extended Abstract). CRYPTO 1992: 196-214

[22] R. Ostrovsky, R. Venkatesan, and M. Yung. Fair games against an all-powerful adversary. Sequences 91 workshop. (see also *AMS DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 13 *Distributed Computing and Cryptography*, Jin-Yi Cai, editor, pp. 155-169. AMS, 1993.)

[23] R. Ostrovsky, R. Venkatesan, and M. Yung. Secure Commitment Against Powerful Adversary: A Security Primitive based on Averag e Intractability. In Proceedings of 9th Symposium on Theoretical Aspects of Computer Science (STACS-92) (LNCS 577 Springer Verlag Ed. A. Finkel and M. Jantzen) pp. 439-448 February 13-15 1992, Paris, France.

[24] R. Ostrovsky, R. Venkatesan, and M. Yung. Interactive hashing simplifies zero-knowledge protocol design. In *Advances in Cryptology - EUROCRYPT '93*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

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

[26] J. Rompel. One-Way Functions are Necessary and Sufficient for Secure Signatures. STOC '90.

[27] A. Russell. Necessary and Sufficient Conditions for Collision-Free Hashing. J. Crypto. 8(2): 87–100, 1995.

Appendix

A The Explicit Hashing Protocol

This section fleshes out the outline from Section 6.

Proving that $y_i \lhd x_i$ There are t paths of length d for which this must simultaneoseusly 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 just other hash pre-images present in the hash tree. Formally:

1. (a) Let $\left\{n^{(1)}, ..., n^{(s)}\right\}$ 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. $c^{(j)} = c_{n^{(j)}}$.

   (b) Server sends $\left\{c^{(1)}, ..., c^{(td)}\right\}$ to the client in random order.

2. Repeat the following cut-and-choose protocol k times:

   (a) 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.

   (b) Client answers with a challenge bit $b \leftarrow \{0, 1\}$.

   (c) If $b = 0$, the server:

      i. 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.)

      ii. opens all commitments to $y_{n^{(j)}}$ (client verifies $y_{n^{(j)}} = H(c^{(j)})$).

   If $b = 1$, the server:

      i. 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}.

A.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[19], 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). Thus, we easily obtain a k-round protocol.

This can actually be improved substantially. First of all, in our setting we do not need the full power of zero-knowledge, but require only that our proof leak nothing about the other data values contained in the hash tree. Since our commitment schemes are information-theoretically hiding, there exists a witness for every possible setting of the other values and thus, we need only that the proofs have witness-hiding proofs. This in fact allows us to collapse the protocol to a 3 rounds, witness-hiding proof of knowledge.

Finally, one can use standard folklore techniques to transform the 3-round witness-hiding proof of knowledge into a ZK proof of knowledge is simulatability is truly desired. This increases the complexity to 5 rounds, and requires the additional assumption of perfectly hiding trapdoor commitment schemes (which exists 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.

Note that 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. Note that if a random oracle is available, then we can in fact use the

[19]This is not true of ZK proofs in general, but it is true for our protocol.

Fiat-Shamir technique to remove interaction completely without losing zero-knowledge (since our underlyingproofs are require only public coins). Formally:

Theorem A.1. *The explicit-hash Merkle tree above allows proving the consistency of t paths of length d using $O(d \cdot t^2 \cdot k^2)$ bits of communication, where k is the security parameter. The protocol can be made provably zero-knowledge with 5 rounds of interaction, witness-hiding with 3 rounds of interaction, and completely non-interactive if one assumes the availability of a random oracle.*

B Achieving Privacy More Efficiently

Given the efficient consistent query protocols for join queries described in Section 3 and Section 5, privacy can be achieved by applying generic witness-hiding or zero-knowledge proofs of knowledge, as described in Section 4. 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 6. 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 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 an interval $[a_n, b_n]$ such 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 $-\infty = 0$ and $\infty - 1 = B - 2$, 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 simply 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 coomitted 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 6, except that now we must add checks of the form $a < b$. We will show how to prove such statemtents 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 $(y_{n_{i+1}} = y_{\mathsf{right}_{n_i}})$.

  This can done via a cut-and-choose protocol as in Section 6. 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 $a, b \in \{0, ..., B-1\}$. 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. [20]

   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 chekcing 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}.

[20] 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 $*$.

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).

B.1 Complexity of the consistency 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. Note that in fact both Micali-Rabin and Kilian [18, 16] gave protocols for private membership queries which were more efficient. However, their technqiues *do not generalize to range queries*.

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.

As for explicit-hash Merkle trees, because we are using perfectly-hiding commitments, we only need witness-indistinguishability and so we can reduce the protocol to 3 rounds. As before, we can obtain a truly zero-knowledge protocol by increasing to 5 rounds, and we can remove all interactivity if we asusume a random oracle.

Contents

1 Introduction **1**
1.1 Our contributions . . . 2

2 Definitions **5**
2.1 Consistent query protocols . . . 5
2.1.1 Keyed databases . . . 7
2.1.2 Privacy . . . 7

3 Data-robust algorithms and consistent query protocols **9**
3.1 Data-robust algorithms . . . 9
3.2 Constructing consistent query protocols from DRA's . . . 11
3.2.1 Pointer-based algorithms . . . 11
3.2.2 A general construction . . . 12

4 Achieving privacy for the general construction **13**

5 Orthogonal Range Queries **16**
5.1 A data-robust algorithm for range queries . . . 16
5.1.1 One-dimensional range trees . . . 16
5.1.2 Two-dimensional range trees . . . 20
5.2 Efficient query protocol . . . 22

6 Explicit-hash Merkle trees **22**
6.1 Achieving Privacy More Efficiently . . . 23

A The Explicit Hashing Protocol **26**
A.1 Complexity of the proofs . . . 27

B Achieving Privacy More Efficiently **28**
B.1 Complexity of the consistency proofs . . . 31