Optimizing ORAM and Using it Efficiently for Secure Computation

Craig Gentry
IBM Research

Kenny Goldman
IBM Research

Shai Halevi
IBM Research

Charanjit Julta
IBM Research

Mariana Raykova
IBM Research

Daniel Wichs
Northeastern University

April 28, 2013

Abstract

Oblivious RAM (ORAM) allows a client to access her data on a remote server while hiding the *access pattern* (which locations she is accessing) from the server. Beyond its immediate utility in allowing private computation over a client's outsourced data, ORAM also allows mutually distrustful parties to run secure-computations over their joint data with sublinear on-line complexity. In this work we revisit the tree-based ORAM of Shi et al. [20] and show how to optimize its performance as a stand-alone scheme, as well as its performance within higher level constructions. More specifically, we make several contributions:

- We describe two optimizations to the tree-based ORAM protocol of Shi et al., one reducing the storage overhead of that protocol by an $O(k)$ multiplicative factor, and another reducing its time complexity by an $O(\log k)$ multiplicative factor, where k is the security parameter. Our scheme also enjoys a much simpler and tighter analysis than the original protocol.
- We describe a protocol for binary search over this ORAM construction, where the entire binary search operation is done in the same complexity as a single ORAM access (as opposed to $\log n$ accesses for the naive protocol). We then describe simple uses of this binary-search protocol for things like range queries and keyword search.
- We show how the ORAM protocol itself and our binary-search protocol can be implemented efficiently as secure computation, using somewhat-homomorphic encryption.

Since memory accesses by address (ORAM access) or by value (binary search) are basic and prevalent operations, we believe that these optimizations can be used to significantly speed-up many higher-level protocols for secure computation.

Contents

1 Introduction

Oblivious RAM. Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky [9, 16, 11], allows a client to outsource her data to a remote server (e.g., "the cloud") and access it efficiently and privately. In particular, the client can access individual elements of her data without revealing to the server *which* elements she is accessing. The efficiency of the client and server on each such data access should be small and essentially independent of (polylogarithmic in) the size of the entire client data. Using ORAM, a client can privately execute arbitrary RAM computations over her remotely stored data, without having to download the data from the server in its entirety. In particular, the client/server computation and communication is essentially only proportional to the time-complexity of the RAM computation itself (with polylogarithmic overhead). Therefore ORAM offers tremendous savings when the client wants to execute simple computations (e.g., binary search) over huge amounts of data.

ORAM for Multiparty Computation. Another use of ORAM is to speed up *secure multi-party computation (MPC)* protocols (e.g., [21, 10] etc.), where two or more parties want to execute some computation over their inputs without revealing the inputs to each other. When the total size of the input is large (e.g., one of the inputs is a large database), one may often want to compute functions whose running time on a RAM is sublinear in the total size of the input. However, secure computation protocols that use either Boolean or arithmetic circuits as a representation of the evaluated functionality, inherently incur computation complexity at least linear in the total size of the input for each secure evaluation. This is also the case for secure computation based on fully homomorphic encryption [6, 5, 4, 3, 7] that uses arithmetic circuits. Several works [17, 14] have demonstrated that ORAM can be used to construct secure computation protocols whose (amortized) complexity can be sub-linear in the total size of the input. In particular, the efficiency of these protocols is proportional to the time-complexity of the underlying computation on a RAM (which may be sublinear in the input size), in contrast to traditional approaches whose efficiency is proportional to the circuit size of the underlying computation (which cannot be sublinear in the input size).

Efficiency of ORAM. Although there has been much recent progress in constructing progressively more efficient ORAM schemes [18, 12, 20, 13], the concrete computation and storage overhead introduced by ORAM is usually significant in practice. Most ORAM schemes are based on a hierarchical structure of [11]. Despite their asymptotic efficiency, these schemes suffer in practice due to their reliance on fairly complex "sorting networks". (Some constructions also have large "spikes" of complexity, where every so often the client and server need run a maintenance procedure of complexity linear in the entire data set.) A recent solution of Shi et al. [20] (which we describe in Section 2) takes a very different approach over these other schemes. Despite having worse asymptotic performance, it is significantly simpler to implement (and understand conceptually), and is deemed to be more efficient in practice because it does not hide any large constant factors. It has been implemented and used in past projects [14]. The client/server computation and communication needed to perform each operation in this protocol is $O(k \log^2 N)$, and the server storage is $O(kN)$, where N is the size of the client data and k is the "security parameter." (Roughly in every operation we incur at most a 2^{-k}-probability of privacy loss. Clearly to support M operations we need $k \gg \log M$ where M is the number of operations; a typical setting may be $k \in [50, 80]$.)

1.1 Our Results

We propose some significant optimizations to the ORAM construction of Shi et al. [20] and its use in performing private RAM computation over outsourced data and secure multiparty computation. These optimizations improve both asymptotic and concrete efficiency of the scheme. We briefly describe each of these optimizations below.

Reduced Storage and Query Time. One of the main disadvantages of the Shi et al. scheme is the large $O(k)$ multiplicative storage overhead on the server. We show a very simple modification to the scheme that gets rid of this overhead; the server only needs to store $O(N)$ bits for N bits of client data (the hidden constant is no more than 4). Another modification improves the computation and communication complexity by a factor of $\log k$ from $O(k \log^2 N)$ to $O(k \log^2 N / \log k)$, and also greatly simplifies and tightens the analysis. Since we usually assume that $N = k^c$ for some small constant c, our improved time/communication complexity simply becomes $O(k \log N)$. These improvements only require relatively minor changes to the scheme of Shi et al., and do not introduce any hidden complexity to its implementation (in fact they may make it even easier to implement). Together, these changes yield a scheme whose asymptotic efficiency matches the best "hierarchal ORAM" construction, while being significantly simpler and with much better concrete efficiency.

Optimized Binary Search. One of the most commonly used sub-linear time operations over large data is *binary search*. This operation is performed whenever executing any query over an indexed database (or sorted data) and only accesses logarithmic number of elements in the database. A direct execution of binary search using ORAM requires a logarithmic number of ORAM queries. We propose a modification for the ORAM scheme of Shi et al. [20] (or our variant of it) that allows the execution of binary search with the efficiency of a single ORAM query.

Optimizations Using Homomorphic Encryption. Lastly, we show that it is possible to optimize the asymptotic communication complexity and the client's computation time using *homomorphic encryption* (HE) [6]. Moreover, by only relying on very limited low-complexity homomorphic operations, these optimizations can be made practically efficient using *somewhat* homomorphic encryption schemes. In particular, the client's computation and the communication complexity per data access is reduced from at least $O(k \log^2 N / \log k)$ to $O(k \log N)$. This may provide a constant-factor savings even for $N = k^c$, and the saving gets larger for larger value of N.

2 Tree ORAM Construction

In this section we summarize the oblivious RAM (ORAM) construction of Shi et al. [20]. Recall that an ORAM construction allows a client to request storage blocks from a server in a way that hides the access pattern. Roughly, the server should not be able to discern if an access query refers to the same storage block as a previous query, or to a new block that was not queried before. The original protocol of Shi et al. uses $O(k \cdot DN \log N)$ storage on the server to store a database of N elements, each of which is D-bits long, with security parameter k (i.e., with failure probability $\exp(-k)$). The client only needs $O(k \log n(D + \log D))$ memory,[1] and the time complexity of the

[1]The client memory can be made as small as $O(k + D + \log N)$, but in the variant that we describe here the server's messages can be as long as $O(k \log N(D + \log N))$ so the client needs this much memory just to store them.

protocol is $O(k \log^2 N(D + \log N))$ per access.

We note that most prior work assumed a computation model with D-bit word size, so the expressions above become $O(kN \log N)$ and $O(k \log^2 N)$, respectively, but here we stick to the more accurate notation that includes also the D factors. We also note that in [20] they use $k = O(\log N)$, but again here we follow the more standard convention of having a separate security parameter, unrelated to the database size.

2.1 Recursive Structure

At the heart of the construction of Shi et al. is the following recursive structure: Suppose that we have an underlying basic ORAM protocol, where the client stores a *table* of N "addresses" of size $\log N$ bits in order to implement oblivious access to a *database* of N elements of size $D > 2 \log N$ bits (and moreover, a single ORAM access to the database only requires a accessing a single address in the table). Since the client's table is at most half the size of the database, it suggests storing the table itself in the same ORAM structure recursively, reducing the client storage by a factor of two with every level of recursion.

In more detail, in level 0 of the recursion the client "outsources" a database consisting of $N_0 := N$ values of D bits each, using a client-stored table consisting of N_0 addresses of size $\log N_0$ bits each. In each future level, the client outsources the table of the previous level using the underlying scheme. In level $i + 1$ of the recursion, the client outsources a table with N_i entries of size $\log N_i$ bits, by thinking of it as a database of $N_{i+1} = \lceil N_i/2 \rceil$ elements, each of of size $2 \log N_i$ bits. This is done using the underlying basic ORAM construction, in which the client only needs to store a level $i + 1$ table of N_{i+1} entries of $\log N_{i+1}$ bits each. Repeating this for $\log N$ levels of recursion reduces the client table to only a constant number of entries.

To access the j'th entry of the level-i table, the client computes the address of that entry in the level-$(i + 1)$ database (with $N_{i+1} = \lceil N_i/2 \rceil$ elements), specifically that address is $j' = \lfloor j/2 \rfloor$. The client uses the level-$(i + 1)$ basic ORAM to fetch that element, and extracts from it the desired entry. For that instance of the basic ORAM protocol, the client needs to access a single entry of its level-$(i + 1)$ table, and this is done recursively using the level-$(i + 1)$ basic ORAM. Therefore, altogether, the recursive scheme uses at most $\log N$ accesses of the underlying ORAM.

Clearly, the same recursive structure can be applied with any integer factor $\tau \geq 2$ between the size of successive levels. Specifically, we store the i-level client table (with N_i entries) as a database of $N_{i+1} = \lceil N_i/\tau \rceil$ elements of size $\tau \log N_i$ bits, accessing the j'th entry of the level-i table is done by retrieving the element $j' = \lfloor j/\tau \rfloor$ in the corresponding database, and we use $\log_\tau(N)$ levels of recursion. The drawback of using a large value of τ is that the size of elements in the i-level database grows linearly with τ, which impacts the running time of the underlying basic ORAM protocol. Below we assume that τ is a small constant, e.g., $\tau = 4$.

2.2 The Underlying Basic ORAM

The underlying basic ORAM protocol from [20] has the server keeping an N-element database in a complete binary tree of depth $\log N$, where each node in the tree contains a bucket large enough to store k data elements (k is the security parameter). Of course, the content of all the buckets is encrypted under the client's key, in particular the server does not know how many elements are actually stored in each bucket.

Each database element with logical address $v \in [N]$ is associated with a random leaf L_v, and the client keeps an N-entry table of the mapping $v \mapsto L_v$. (I.e., entry v in the table contains the leaf number L_v.)

Denote by d_v the data corresponding to logical address v. Throughout the protocol we maintain the invariant that the triple (L_v, v, d_v) is stored in one of the buckets on the path from the root to the leaf L_v. (Initially it is stored at the leaf itself.) Access to logical address v consists of two subroutines, one for doing the actual access and another one to clean up after the first.

Access. To access the data associated with logical address v, the client looks up L_v in its table and asks the server for the entire path from the root to leaf L_v. Upon receiving all the buckets in this path, the client decrypts them, finds a triple of the form (L_v, v, d_v) in one of the buckets (for some d_v), and this value d_v is the requested data.

The client either leaves the data unchanged (if the operation is a read) or overwrites it with a new value (if it is a write), and we denote the resulting data by d'_v. In either case, it chooses a new random leaf $L'_v \in [N]$ and updates its table with the new L'_v value. The client then removes the triple (L_v, v, d_v) from the bucket where it was found, and instead puts the triple (L'_v, v, d'_v) in the root bucket. Finally it re-encrypts all the buckets and send them back to the server, who replaces all the buckets on the path to L_v by the new encrypted buckets.

Observe that since the new triple is placed at the root, then this operation does not violate the tree invariant that we maintain.

Eviction. To prevent the root bucket from overflowing, the client and server run a "maintenance" subroutine whose goal is to evict triples from their current buckets and push them to buckets lower down the tree. Specifically, the client chooses at random two nodes from every level of the tree except the leaves. For every chosen node, it asks the server for the buckets in that node and its two children. The client decrypts these buckets, and if the parent bucket contains any triples then it chooses one of them at random, which we denote by (L_v, v, d_v). The client removes that triple from the parent bucket, and pushes it to the bucket of one of the two children.

The child to which this triple is evicted is determined by L_v. Namely, the tree invariant tells us that this triple is currently found in some node on the path from the root to leaf number L_v, and the client just pushes it one level down on that path. Clearly, this operation maintains the tree invariant. The client then re-encrypts the buckets of the parent and its two children and send them back to the server, who stores the new buckets in place of the old ones.

Shi et al. proved in [20] that the probability of any bucket overflowing is small. Specifically, the probability of an overflow in m operations is bounded by $O(mN/\exp(k))$. They also proved that as long as no overflow occurs, the view of the server is computationally independent of the access pattern (assuming the security of the encryption scheme).

2.3 Putting it Together

In the complete construction of [20], the server keeps $\ell - 1 = \lceil \log_\tau(N) \rceil - 1$ complete binary trees, with the level-i tree having $\tau^{\ell-i}$ leaves (rounded up to a power of 2). Each node in every tree has a bucket large enough to store k "entries", each entry is assigned a (changing) random leaf of the

tree, and we maintain the invariant that each entry can be found in one of the buckets on the path from the root to "its leaf."

In the largest tree ($i = 0$), each entry corresponds to one logical address $v \in \{0, \ldots, N-1\}$, and it contains the user data for that logical address. For the next tree ($i = 1$), each entry corresponds to a size-τ interval of logical addresses, and it contains the τ leaf-numbers in the largest tree that are currently assigned to the entries of those τ logical addresses. More generally, each entry in the tree at level $i+1$ corresponds to the union of τ level-i intervals (which is itself a size-τ^{i+1} interval of logical addresses), and that entry contains τ leaf-numbers of the level-i tree, namely the leaves that are currently assigned to the entries of those τ level-i intervals. With each entry in every tree we store also the first logical address of the interval of that entry, as well as the leaf that is currently assigned to that entry. Thus each entry is of the form

$$\begin{array}{lllll} \text{level-0}: & \big(& L^*, & v, & \text{user-data} \quad \big) \\ \text{level} > 0: & \big(& L^*, & v, & (L_1, L_2, \ldots, L_\tau) \big) \end{array}$$

where L^* is the leaf currently assigned to that entry, $[v, v+\tau^i)$ is its interval, and $(L_1, L_2, \ldots, L_\tau)$ are the leafs in the next tree that are currently assigned to the τ sub-intervals

$$[v, v+\tau^{i-1}),\ [v+\tau^{i-1}, v+2\tau^{i-1}),\ [v+2\tau^{i-1}, v+3\tau^{i-1}),\ \ldots$$

Of course, all of the buckets in all of the trees are encrypted under a key known to the client.

The "tree at the last level ℓ", which has a single node, is kept by the client. That tree has just a single entry, corresponding to the interval $[0, \tau^\ell)$, and containing τ leaf-numbers of the tree at level $\ell - 1$ that are currently assigned to the entries of the sub-intervals $[0, \tau^{\ell-1}), [\tau^{\ell-1}, 2\tau^{\ell-1}), \ldots$

ORAM Access Query. To access the logical address v, the client looks up in its level-ℓ "tree" for the list $(L_1^{(\ell-1)}, \ldots, L_\tau^{(\ell-1)})$, and determines the level-$(\ell-1)$ sub-interval containing v, namely j such that $(j-1)\tau^{\ell-1} \le v < j\tau^{\ell-1}$. The client sets $v_{\ell-1} = (j-1)\tau^{\ell-1}$ and $L^{(\ell-1)} = L_j^{(\ell-1)}$, chooses at random a new leaf $\hat{L}^{(\ell-1)}$ and replaces $L_j^{(\ell-1)}$ by this new value in the list. Then the client proceeds iteratively for $i = \ell - 1$ down to 0:

1. Request from the server all the buckets on the path from the root of the level-i tree down to the leaf $L^{(i)}$. Decrypt them and find in them an entry of the form $(L^{(i)}, v_i, \mathsf{data})$.
2. If $i > 0$ do the following:
   (a) Parse $\mathsf{data} = (L_1^{(i-1)}, \ldots, L_\tau^{(i-1)})$, choose a new random leaf in the next tree, $\hat{L}^{(i-1)}$.
   (b) Determine the level-$(i-1)$ sub-interval containing v, namely j such that $v_i + (j-1)\tau^{i-1} \le v < v_i + j\tau^{i-1}$. Set $v_{i-1} = v_i + (j-1)\tau^{i-1}$ and $L^{(i-1)} = L_j^{(i-1)}$.
   (c) Replace $L_j^{(i-1)}$ by $\hat{L}^{(i-1)}$ inside data, denoting the result by data'.

   Else ($i = 0$), if this is a write operation then set data' to be the new value. Otherwise (read), set $\mathsf{data}' = \mathsf{data}$.
3. Remove the entry $(L^{(i)}, v_i, \mathsf{data})$ from the bucket where is was found, and instead place in the root bucket the entry $(\hat{L}^{(i)}, v_i, \mathsf{data}')$. Re-encrypt all the buckets and send to the server.

Finally, the client and server run the Eviction subroutine for each of the trees $i = 0, 1, \ldots, \ell - 1$. If this was a read operation then the return value is the data value from the last level $i = 0$.

The complexity of this protocol is essentially linear in the amount of information sent between the client and server. This infomration consists of a constant number of buckest from every level in every tree. Since the $(\ell - i)$'th tree has depth $\log_2(\tau^i) = i \log_2 \tau$, then the number of buckets sent is

$$\log_2 \tau \cdot \sum_{i=0}^{\ell-1} i = O(\log \tau \cdot \ell^2) = O(\log \tau \cdot (\log N / \log \tau)^2) = O(\log^2 N / \log \tau)$$

(with $\log N$ buckets sent in the largest tree). Each bucket has size either $O(k(D + \log N))$ (for the largest tree) or $O(k\tau \log N)$ (for all the smaller trees), hence the total communication (and computation) complexity is

$$O(\log^2 N / \log \tau) \cdot O(k\tau \log N) + \log_2 N \cdot O(k(D + \log N)) = O\big(k \log^2 N (D + \tau \log N / \log \tau)\big).$$

Using a small constant for τ, we get the claimed complexity of $O(k \log^2 N (D + \log N))$.

3 Parameter Optimizations for the Tree ORAM Scheme

In this section we present two optimization for the underlying basic tree-based ORAM construction of Shi et al. [20]. First we describe in Section 3.1 a very simple optimization that saves a factor of $O(k)$ in the storage overhead of the protocol. Then in Section 3.2 we describe a somewhat more involved optimization that reduces the time complexity per access by a factor of $O(\log k)$, and also considerably simplifies and tightens the overflow-probability analysis.

3.1 Reducing the Height of the ORAM Tree

The construction from [20] uses an N-leaf tree to store an N-element database. We first observe that increasing just slightly the capacity of the buckets, one can achieve the same overflow probability for many more elements than leaves. Note that the buckets in [20] are large enough to keep $O(k)$ elements, but the expected number of elements in each bucket in only $O(1)$, which is the reason for the $O(k)$ overhead in storage. We observer, however, that we can increase the expected number of elements per bucket to any desired number m, while keeping the overflow probability at $\exp(-k)$, simply by making the bucket capacity $O(m + k)$, thus reducing the overhead to $O(m/k)$. Hence by setting $m = k$, we reduce the storage overhead to only $O(1)$.

Specifically, by making the capacity of each bucket be $2k$ (rather than k as in [20]), we can use a shallower tree with only n/k leaves to store n elements. We reduce the number of nodes by a factor of k while increasing the bucket size by only a factor of 2, thus getting a factor $k/2$ saving in storage.

For this new setting, it is enough to analyze the overflow probability only in the leaves, since all the internal nodes have the exact same behavior as in the original construction of [20]. For the leaves we have the following:

Lemma 1. *Fix one particular leaf (out of the n/k leaves in the tree) and one particular access operation. With the bucket of that leaf having size $2k$, the overflow probability for this leaf in that operation is at most $e^{-k/3}$.*

Proof. For any fixed access operation, we can model the content of the leaves as having n balls thrown randomly into n/k bins. The expected number of balls in a bin is k, and using Chernoff bound [1] we get that the probability that any particular bin has more than $2k$ balls is upper-bounded by $e^{-k/3} \approx 2^{-k/2}$. ∎

(We remark that to improve the bound from $2^{-k/2}$ to 2^{-k}, it suffices to increase the size of each bucket from $2k$ to just under $2.6k$.)

3.2 Increasing the Degree of the Tree

Our next observation is that instead of using just binary trees as in [20], we can use higher-degree trees. Specifically, we propose increasing the branching factor of the trees to k, and decreasing their depth accordingly to $\log_k(n/k) = (\log n/\log k) - 1$. Clearly, this modification reduces the running time of the Access subroutine from Section 2.2 by a $\log k$ factor (simply because the tree is shallower, and hence the size of the paths that the server sends is decreased). However, this would increase the complexity of the Evict procedure by a factor of k, since each eviction operation now involves a parent and all of its d children (rather than just two children as in [20]). We therefore change the eviction algorithm so as to avoid the increased computation cost, as described next.

3.2.1 A New Eviction Procedure

Notations. Below we denote $[k] = \{0, 1, \ldots, k-1\}$, and use the standard notation where we name each node in a degree-d tree using the path from the root to that node, represented as a sequence of integers in $[k]$. For example the root is the empty sequence (), child number 3 of the root it (3), the child number 2 of that node is (3, 2), etc. Formally, the nodes of a complete k-ary tree of height h are $[k]^{\leq h}$, where each internal node $v \in [k]^{<h}$ is the parent of all the nodes $\{(v, i) : i \in [k]\}$.

With these notations, the *least common ancestor* of two nodes u, v, denoted $LCA(u, v)$, is just the longest common prefix in the names of these two nodes.

We also use the notation $\mathsf{DigitReverse}_k(t)$ to denote the order-reversal of the base-k digits of the integer t, for example in decimal representation we have $\mathsf{DigitReverse}_{10}(1234) = (4, 3, 2, 1)$. Formally, if $t < k^h$, $t = \sum_{j=1}^{h} i_j k^{j-1}$ for unique $i_i < k$, then $\mathsf{DigitReverse}_k(t) = (i_h, \ldots, i_2, i_1) \in [k]^h$.

The push-to-leaf procedure. We replace the randomized eviction procedure of Shi et al. with a deterministic procedure that just picks one leaf L in every time step (in a specific order to be described shortly), and tries to push items down along the path to L as far as they can go. Namely, if somewhere on the path from the root to L we found an item that is assigned to leaf L', then we push that item down the path to the node $LCA(L, L')$, i.e., the least common ancestor where the paths to L and L' diverge.

For example, if we choose leaf $L = (1, 2, 0, 3)$, and in the root we found an item (L', v, data) that is assigned to leaf $L' = (1, 2, 3, 3)$, then we remove that item from the root and push it down to the internal node two levels below $LCA(L, L') = (1, 2)$. We do the same for all the items on the path from the root to L. Notice that during the entire procedure, we only access nodes along this one path.

We observe that if our tree satisfies the tree invariant before the push-to-L procedure (i.e., every item in the tree is found on the path from the root to its assigned node), then it still satisfies that invariant also after the procedure. Moreover no item is ever *pushed up* by that procedure, only

EVICT(tree $= [k]^{\leq h}$, step $t \in \mathbb{N}$):

1. Set $t' = t \bmod k^h$, and $L = \mathsf{DigitReverse}_k(t')$; // L is a leaf
2. For every item $I = (L', v, \mathsf{data})$ in any bucket on the path to L, do: // push-to-L
3. Remove I from its current bucket and push it to the bucket of node $LCA(L, L')$.

Figure 1: Our new eviction procedure

down (or not at all). We also note that it is possible to implement the push-to-leaf procedure in time complexity linear in the size of a root-to-leaf path.

Ordering the leaves. The deterministic order in which we pick the leaves for eviction is just the *digit-reversed lexicographic order*. For example, with $k = 3, h = 2$, we use the order

$$(0,0),(1,0),(2,0),(0,1),(1,1),(2,1),(0,2),(1,2),(2,2), \quad (0,0),(1,0),\ldots$$

Specifically, for a degree-k, height-h tree, we pick at every step $t \in N$ the leaf whose name is $L = \mathsf{DigitReverse}_k(t \bmod k^h)$. The new eviction procedure, using this ordering of the leaves, is described in Figure 1. The main property of this ordering is that, at each level i of the tree, these paths *cycle* through the k^{i-1} nodes of that level periodically every k^{i-1} steps.

3.2.2 Overflow Analysis

The overflow analysis for the leaves does not change from the previous section. For internal nodes, we now prove that if we set the bucket size in the internal nodes to $2k$, then with the eviction procedure from Figure 1 the overflow probability is exponentially small in k. Surprisingly, the analysis for our deterministic evacuation procedure turns out to be a rather easy, quite similar to the analysis for the leaves. In particular we do not need any of the "heavy" queuing-theory tools that were used in the analysis of the original protocol in [20].

Lemma 2. *Fix one particular internal node in the tree, $v \in [k]^{<h}$, and one particular access operation. With the bucket of v having size $2k$, the overflow probability for this node in that operation is at most $e^{-k/3}$.*

Proof. We think of the evolution of data items in internal nodes as a game, in which in every step we throw a ball in the root and associate to it a random leaf, and then perform our deterministic evacuation procedure. Assume that the node v in question is at level i in the tree, which means that there are exactly k^{i-1} nodes in that level (and exactly k^i nodes one level below).

Fix some step $t \geq k^i$, and denote by $S_t(v)$ the set of all balls in the system that satisfy (a) the path from the root to the leaves of these balls go through the internal node v, and (b) the balls are at level i or above at time t. We prove that $\Pr[|S_t(v)| > 2k] < e^{-k/3}$, which clearly implies the lemma (since $S_t(v)$ is a superset of the balls in v at time t).

The property of the digit-reversed lexicographic order that we need for the analysis, is that every consecutive interval of k^i steps covers each length-i prefix exactly once. This means in particular, that every node at the level $i+1$ was on exactly one eviction path during the sequence of k^i steps $[t - k^i + 1, \ldots, t-1, t]$. Hence every ball that entered the system *before that interval* must have been evicted to level $i+1$ or below, and therefore in not in $S_t(v)$.

It remains to consider only the last k^i balls. For each of these balls we assign an indicator random variable χ_j ($j \in [t-k^i+1,\ldots,t]$), which is 1 if the path of that ball goes though v and 0 otherwise. Clearly the χ_j's are i.i.d. and $\Pr[\chi_j = 1] = k^{-i+1}$ for all j. Moreover only balls with $\chi_j = 1$ can belong to $S_t(v)$, hence $|S_t(v)| < \sum_{j=t-k^i+1}^{t} \chi_j$. Since the expected value of the sum is $E[\sum_j \chi_j] = k^i \cdot k^{-i+1} = k$, we can use Chernoff bound to conclude that

$$\Pr[|S_t(v)| > 2k] \le \Pr\left[\left(\sum_{j=t-k^i+1}^{t} \chi_j\right) > 2k\right] < e^{-k/3}.$$

■

3.2.3 Complexity Analysis

It remains to analyze the complexity of our new variant. As usual, the time complexity is governed by the communication complexity of the protocol. To analyze the communication, we recall the different parameters in our system:

We have security parameter k, and an N-element database, each element of size D bits. The largest tree has $\lceil N/k \rceil$ leaves, and each subsequent tree is smaller than the previous one by a constant factor of τ, hence the i'th tree has $\lceil N/(k\tau^i) \rceil$ leaves and we have $\ell = \lceil \log_\tau(N/k) \rceil$ trees. The trees have branching degree k, thus the height of the i'th tree is $\log_k(\lceil N/(k\tau^i) \rceil) \approx (\log N - i \log \tau)/\log k$. Every node in the largest tree contains $2k$ entries, each of size $D + O(\log N)$, for a total size of $O(k(D + \log N))$. In the smaller trees, every node contains $2k$ entries, each of size $O(\tau \log N)$, for a total size of $O(k\tau \log N)$.

Recall that in our protocol, for each access request the parties send to each other two paths in every tree (one for the access itself and another one for eviction). Hence the total communication complexity per access in our protocol is

$$\begin{aligned}
&O(\log N/\log k) \cdot O(k(D+\log N)) + \sum_{i=1}^{\ell-1} O((\log N - i\log\tau)/\log k) \cdot O(k\tau \log N) \\
&= \frac{k\log N}{\log k} \cdot O\big(D + \log N + \tau\ell \log N\big) \;=\; \frac{k\log N}{\log k} \cdot O\big(D + \log N + (\tau/\log\tau)\log^2 N\big) \\
&= O\big(k \log N (D + \log^2 N)/\log k\big),
\end{aligned}$$

where in the first two equalities above we assumed that $N > k^2$ (say), so that $\log(N/k) = O(\log N)$, and in the last equality we used the fact that τ was a constant, hence $(\tau/\log\tau) = O(1)$. As noted above, in the word model where we can process a D-bit word in one operation (and $D \ge \log N$), the above bound can be replaced by $O(k \log^2 N/\log k)$.

4 Oblivious Binary Search

Oblivious RAM lets us retrieve a value from the database by specifying its logical address, but in some applications we may want to retrieve the data by its key (or tag) rather than address. In such applications, we may initially sort the elements by their tags, then use binary search to retrieve the tag that we want in $\log n$ steps. Implementing this procedure obliviously is similarly going to take $\log n$ ORAM accesses, which may be quite expensive. Here we present a modification of the tree

ORAM protocol from before that lets us to implement a binary search query with only a single ORAM access. For a logical address v (in the actual database) denote by (t_v, d_v) the tag and data that are associated with that address, respectively.

Consider the recursive structure of the tree ORAM construction from [20]. In this structure, an entry in the tree at level $i+1$ contains a list of τ leaf numbers in the next tree at level i, and a client that fetches that entry chooses one of these leaves (based on the logical address that it wants to access) and then fetches that leaf (and the path to it) from the next tree. To enable efficient binary search, we need to add some information to let the client decide what leaf to fetch next, based on its tag rather than on its virtual address.

For an interval I of logical addresses, denote by $t_\downarrow(I), t^\uparrow(I)$ the smallest and largest of the tags associated with I, namely $t_\downarrow(I) = \min\{t_v : v \in I\}$ and $t^\uparrow(I) = \max\{t_v : v \in I\}$. Since the database is sorted by tags, we know that an address $v \in [N]$ belongs to I if and only if the corresponding tag t_v satisfies $t_\downarrow(I) \le t_v \le t^\uparrow(I)$.

Consider an entry in the tree at level $i+1$, let $[v, v+\tau^{i+1})$ be the interval of logical addresses corresponding to that entry, and also let $I_1, \ldots, I_\tau$ be the corresponding level-i sub-intervals, namely $I_j = [v + (j-1)\tau^i, v + j\tau^i)$. We modify the ORAM protocol by storing the tags $t_\downarrow(I_j)$ together with the leaf numbers L_j that are currently assigned to these sub-intervals. Namely, an entry in the tree has the form

$$\begin{array}{lllll} \text{Level-0}: & (& L^*, & v, & (t_v, d_v) \quad) \\ \text{Level-}i>0: & (& L^*, & v, & \big(\{L_1^{(i)}, t_\downarrow(I_1)\},\ \{L_2^{(i)}, t_\downarrow(I_2)\},\ \ldots,\ \{L_\tau^{(i)}, t_\downarrow(I_\tau)\}\big) \quad) \end{array}$$

When accessing some tag t, the client fetches some level-$(i+1)$ entry, compares the tag t against all the τ tags $t_\downarrow(I_j)$ in that entry, and based on that comparison decides which of the leaves L_j to get from the next tree. Specifically, we replace the Access subroutine from Section 2.3 by the following:

Binary Search. To access the data for tag t (if exists), the client looks up in its level-ℓ "tree" for the list $(\{L_1^{(\ell-1)}, t_\downarrow(I_1)\}, \ldots, \{L_\tau^{(\ell-1)}, t_\downarrow(I_\tau)\})$, and let j be the largest index such that $t_\downarrow(I_j) \le t$. (Such an index must exist since $t_\downarrow(I_1)$ is the smallest tag in the database.) The client sets $v_{\ell-1} = (j-1)\tau^{\ell-1}$ and $L^{(\ell-1)} = L_j^{(\ell-1)}$, chooses at random a new leaf $\hat{L}^{(\ell-1)}$ and replaces $L_j^{(\ell-1)}$ by this new value in the list. Then the client proceeds iteratively for $i = \ell - 1$ down to 0:

1. Request from the server all the buckets on the path from the root of the level-i tree down to the leaf $L^{(i)}$. Decrypt them and find in them an entry of the form $(L^{(i)}, v_i, \mathsf{data})$.
2. If $i > 0$ do the following:
   (a) Parse $\mathsf{data} = (\{L_1^{(i-1)}, t_1\},\ \ldots,\ \{L_\tau^{(i-1)}, t_\tau\})$, choose a new random leaf $\hat{L}^{(i-1)}$.
   (b) Let j be the largest index such that $t_j \le t$. (Such an index must exists since t_1 is equal to the $t_\downarrow(\cdot)$ value from the previous iteration.) Set $v_{i-1} = v_i + (j-1)\tau^{i-1}$ and $L^{(i-1)} = L_j^{(i-1)}$.
   (c) Replace $L_j^{(i-1)}$ by $\hat{L}^{(i-1)}$ inside data, denoting the result by data'.

   Else ($i = 0$), parse $\mathsf{data} = (t_v, d_v)$. if $t_v = t$ and this is a write operation then set $\mathsf{data}' = (t_v, d'_v)$ where d'_v is the new value. Else set $\mathsf{data}' = \mathsf{data}$.

3. Remove the entry $(L^{(i)}, v_i, \mathsf{data})$ from the bucket where is was found, and instead place in the root bucket the entry $(\hat{L}^{(i)}, v_i, \mathsf{data}')$. Re-encrypt all the buckets and send to the server.

As before, the client and server then run the Eviction subroutine for each of the trees. If this was a read operation then the return value is either d_v from level-0 (if $t_v = t$) or $\perp$ (if $t_v \neq t$).

Complexity. The binary-search algorithm above is nearly identical to the ORAM access from Section 2.3, except in the way that the next label is chosen from the current entry (i.e., the choice of j in Step 2(b)). While in the original ORAM tree the client can compute j directly from its logical address, in the binary search algorithm the client needs to compare its tag against all the tags in the entry to find the index j. Still, the complexity of this algorithm is linear in the size of the messages that the server sends, and if the tags are short (of size $O(\log n)$ bits) then that size is asymptotically the same as for the ORAM access. We also note that the same ORAM modification can also be used for regular queries that access logical address, for example if we want to handle a sparse set of addresses out of a large universe.

4.1 Applications

Range queries. Another type of query we can perform on sorted data is a range query. Given a sorted databased stored in an ORAM and an interval $[t_l, t_r]$, we can retrieve all database items with search tag in the interval $[t_l, t_r]$ with computational complexity proportional to the size of the result set times the cost of a single ORAM access. To do this we just run the binary-search algorithm from above on the tag t_l, and this returns both the first tag $t \geq t_l$ in the database as well as its logical address v. We then just access logical addresses $v+1, v+2, \ldots$ until we get the first tag $t > t_r$. (It is easy to avoid that last extra query by paying attention to the values $t_{\downarrow}(I)$ throughout the queries.)

Assuming that the client has enough memory to store the result set, we can modify the ORAM binary search algorithm to implement a range query with fewer communication rounds (namely, have the client accesses each tree only once). This is done by the client requesting not just one leaf from each level, but also all the leaves $L_j^{(i)}$ with larger tag values, as long as their t_j values satisfy $t_j \leq t_r$. In addition to saving communication rounds, this approach also saves on the total number of leaves accessed (especially in the small trees), since we do not access the same leaf more than once.

Keyword search. The keyword search problem is to retrieve all the records from a database that contain a given keyword. One solution is to build an (inverted) index, namely a list of all the words in the database, and for each word w keep a list of addresses of all the documents $D_1^{(w)}, \ldots, D_i^{(w)}$ that contain w. Having this index reduces the problem of keyword search to finding the keyword in the list of the inverted index, which can be done in sublinear time using binary search. With this setup we can use our binary search algorithm for an efficient access hiding algorithm for keyword search.

5 Secure Computation using HE-Over-ORAM

An exciting application of ORAM is secure computation with sublinear online complexity [17, 14]. In this setting, we have two players who first engage in an "offline" setup protocol, at the end

of which one of them is holding the server state of the oblivious RAM, while the client state is shared between them so that no single player knows it. Then, in the "online" phase, the players just carry out secure computation for the evolving finite state of the control (i.e., the client state), and implement any memory access that the computation needs using the underlying ORAM. Using this configuration, the online complexity of the protocol is related to the amount of data which is actually accessed during this computation, rather that to the overall amount of data held by the server. This configuration was first mentioned by Ostrovsky and Shoup [17], and recently an optimized version was described and implemented by Gordon et al. [14], working over the ORAM protocol of Shi et al. [20]. A crucial difference between the standard ORAM setting and this configuration of MPC-over-ORAM is that in the latter, the client is not allowed to know in the clear the logical addresses that are accessed or the values that are returned. Instead, that information is shared between the two parties, and is processed by the secure-computation protocol in this shared format.

In this section we show how to leverage homomorphic encryption to reduce the communication and interaction between the client and the server in the during our ORAM protocol (which itself may be embedded inside a larger secure computation protocol). A practical concern is that the complexity of contemporary HE schemes degrades quickly with the complexity of the functions that are computed (especially the multiplicative depth). We show how the functions used inside ORAM can be expressed as low-depth functions that can be evaluated by *somewhat homomorphic encryption* [6] schemes, which are weaker but much faster than fully homomorphic encryption schemes.

5.1 Our High-Level Approach to Sublinear Secure Computation

In our system, we have a client and a server. The server roughly plays the role of the ORAM-server and the client roughly plays the role of the ORAM-client, but the mapping is not quite one-to-one. Below we say "client" and "server" when referring to the two parties in our secure computation protocol, and "ORAM-client" and "ORAM-server" when referring to the roles in the underlying oblivious RAM construction.

Just as in the underlying ORAM construction, for our protocol the content of the database is stored at the server. In our optimized version, the content is encrypted using a HE scheme *to which the client holds the secret key.* Since we are implementing secure computation rather than just ORAM, the client in our case cannot hold in the clear the address-to-leaf mapping for the smallest tree (since this will leak to the client information about the logical addresses being accessed). Instead, this table is shared between the two players in a simple two-out-of-two fashion: if in the underlying construction the ORAM-client has a table with the i'th entry being some value l_i, in our protocol the server holds a random mask value m_i and the client holds $l_i \oplus m_i$.

When implementing the access protocol, the client and server cooperate to jointly compute the ORAM-client messages in the clear. Note that this does not leak any information due to the privacy property of oblivious RAM. In our case, each message of the ORAM-client consists of a single leaf address in one of the trees. In the underlying construction the server would reply by returning the full path to that leaf, but in our system the server will return much less than that. In some cases the server will compute the entire ORAM-client logic homomorphically, returning to the client in our protocol just the encrypted result (namely the leaf address in the next tree). In some cases we need to hide even this final result from the client (e.g., the actual data in the largest tree), so the server will blind it using a random mask, and send to the client this blinded value to be decrypted.

This will again result in the server holding a random mask m and the client holding the value $v \oplus m$ (where v is the value that the ORAM-client was supposed to get).

Note that homomorphic encryption allows us to save considerably on bandwidth. Below, we describe in more detail the implementation of the various sub-protocols involved.

5.2 ORAM Access

In the ORAM scheme of [20], the data that is stored in each node in any of the trees consists of pairs (v_i, d_i), where d_i is either the data element corresponding to logical address v_i (if this is the largest tree), or leaf identifiers in the next tree (if this is an intermediate tree). In the underlying ORAM construction the ORAM-client holds an address v, it gets from the ORAM-server an entire path in the tree, and it decrypts all the nodes and search for an entry matching that logical address v that it holds.

In our case, the client holds a masked logical address $u = v \oplus m$ and the server holds the corresponding mask m. The client encrypts its values u and sends to the server, who uses (additive) homomorphism to add-in its mask, thus obtaining a HE ciphertext encrypting v.

The server next evaluates homomorphically the client logic that selects the one entry from all of the nodes that match the logical address v. Specifically, for every encrypted entry of the form (v_i, d_i), the server first compute a single "choice bit" c_i which is one if $v_i = v$ and zero otherwise. This can be done naively by comparing each bit of v_i to the corresponding bit of v, then taking an AND of all the resulting bits. The bit-wise comparison is just a NOT XOR operation (hence it is linear), and the big AND has depth logarithmic in the size of v. Note that current homomorphic encryption schemes [3, 7, 8, 2] can evaluate arithmetic circuits of *polynomial depth* even without an expensive procedure called "bootstrapping", and that a scheme capable of evaluating *logarithmic depth* is realizable using comparatively small parameters. Alternatively, if the size of v is too large and one wants to lower the depth of the circuit further, one can use a low-degree approximation of the AND function, such as the approximation due to Ben-Or described, e.g., in Example 2.4 of [15]. (The example is a low-degree approximation of OR, but it is straightforward to convert it to a low-degree approximation of AND.) For a soundness parameter k, this low-degree approximation introduces an error probability of 2^{-k}, but it can be computed in depth only $\log k$. Once the server computes all the choice bits c_i, it then evaluates homomorphically the MUX function

$$\mathsf{out} = \sum_i c_i \cdot d_i$$

thus obtaining only the value d that matches the logical address v. (Note that the c_i's are individual bits, but the d_i's are bit-string. The operation $c_i \cdot d_i$ thus refers to multiplying by c_i each of the bits of d_i.) This last computation only adds one to the depth of the multiplication depth of the circuit.

The harder part of the access procedure is removing the entry (v_i, d_i) from its current intermediate node putting the corresponding (v_i, d'_i) at the root of the tree. To enable this homomorphic procedure, we keep with each entry in every node also an encrypted "full-bit" which is 1 if that entry is full and zero if it is empty (and this full-bit is used also in the comparison calculation above). After computing the choice bits c_i, we just multiply the negation of each c_i back into the corresponding full-bit, thus "removing" the entry from its current node. Choosing a new leaf value for the next tree (as needed for the intermediate trees) is done by both the client and server choosing random values, and the new address being the XOR of the two strings (this can be computed using additive homomorphism).

To put it in the top node, we compute for each entry in the top node a bit signaling whether that entry is the first empty entry in the node. This can be done naively in depth equal to the logarithm of the bucket size, namely for each entry we AND the negation of its full-bit with all the full-bits of previous entries. Here again we can use the Ben-Or trick to compute these bits in only $\log k$ depth, with k the security parameter. Once we compute these "first available" bits (call them a_i), we compute for each entry of the form (v, d) in the top node another MUX function, setting the new value of that entry to

$$(v'', d'') = a_i \cdot (v_i, d'_i) + (1 - a_i) \cdot (v, d)$$

We note that repeated accesses gradually increase the noisiness of the encryptions of the data stored in the tree, hence they need to be periodically refreshed, which can be done using bootstrapping [6]. Alternatively, we can use an interactive protocol where the server blinds the encrypted data using additive homomorphism, then send it to the client to be decrypted and freshly re-encrypted, and finally remove the blinding factors.

ORAM Evictions. If we use HE for encryption of the data in the ORAM, then the evictions can be done completely noninteractively by the server. Given a node chosen for eviction, the server homomorphically evaluates a function that pushes each of the items stored in the node to an available slot in the least-common-ancestor of the "current eviction leaf" and the leaf associated with that item. The operations involved with eviction are simlar to those required on read, and in particular can be implemented using functions of depth $\log k + \log \log N$.

5.3 Binary Search

In our extension of the ORAM scheme for more efficient binary search, we need to check not for exact match but whether the searchable address is within a given interval. For this purpose we can define a function:

$$g_{\overline{v_1 \ldots v_k}}(x_1, \ldots, x_k) = (x_1 - v_1)x_1 + (x_1 - v_1 + 1)g_{\overline{v_2 \ldots v_k}}(x_2, \ldots, x_k), \tag{1}$$

where the bits of x and v are ordered most significant to least significant. The function $g_v(x)$ takes value 1, if $x > v$ and 0, otherwise. Now, in order to implement a selection condition that v is in the interval (v_l, v_r) we can set $f_v(x) = g_{v_l}(v) \cdot g_v(v_r)$ for the selection function in Equation 1.

Acknowledgments. This work was supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center (DoI/NBC) contract number D11PC20202. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

References

[1] Noga Alon and Joel Spencer. *The Probabilistic Method.* John Wiley, 1992.

[2] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In Safavi-Naini and Canetti [19], pages 868–886.

[3] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS*, pages 309–325. ACM, 2012.

[4] Zvika Brakerski and Vinod Vaikuntanathan. Efficient Fully Homomorphic Encryption from (Standard) LWE. In Rafail Ostrovsky, editor, *FOCS*, pages 97–106. IEEE, 2011.

[5] Zvika Brakerski and Vinod Vaikuntanathan. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[6] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[7] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012.

[8] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the aes circuit. In Safavi-Naini and Canetti [19], pages 850–867.

[9] Oded Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In Alfred V. Aho, editor, *STOC*, pages 182–194. ACM, 1987.

[10] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

[11] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[12] Michael T. Goodrich and Michael Mitzenmacher. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP (2)*, volume 6756 of *Lecture Notes in Computer Science*, pages 576–587. Springer, 2011.

[13] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In Yuval Rabani, editor, *SODA*, pages 157–167. SIAM, 2012.

[14] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[15] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *FOCS*, pages 294–304. IEEE Computer Society, 2000.

[16] Rafail Ostrovsky. Efficient Computation on Oblivious RAMs. In Harriet Ortiz, editor, *STOC*, pages 514–523. ACM, 1990.

[17] Rafail Ostrovsky and Victor Shoup. Private information storage. In *STOC*, 1997.

[18] Benny Pinkas and Tzachy Reinman. Oblivious RAM Revisited. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 502–519. Springer, 2010.

[19] Reihaneh Safavi-Naini and Ran Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.

[20] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *ASIACRYPT*, pages 197–214, 2011.

[21] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.