Secure Two-Party Computation in Sublinear (Amortized) Time

S. Dov Gordon
Columbia University
gordon@cs.columbia.edu

Jonathan Katz
University of Maryland
jkatz@cs.umd.edu

Vladimir Kolesnikov
Alcatel-Lucent Bell Labs
kolesnikov@research.bell-labs.com

Fernando Krell
Columbia University
fernando@cs.columbia.edu

Tal Malkin
Columbia University
tal@cs.columbia.edu

Mariana Raykova
Columbia University
mariana@cs.columbia.edu

Yevgeniy Vahlis
AT&T Security Research Center
evahlis@att.com

ABSTRACT

Traditional approaches to generic secure computation begin by representing the function f being computed as a circuit. If f depends on each of its input bits, this implies a protocol with complexity at least linear in the input size. In fact, linear running time is *inherent* for non-trivial functions since each party must "touch" every bit of their input lest information about the other party's input be leaked. This seems to rule out many applications of secure computation (e.g., database search) in scenarios where inputs are huge.

Adapting and extending an idea of Ostrovsky and Shoup, we present an approach to secure two-party computation that yields protocols running in *sublinear* time, in an amortized sense, for functions that can be computed in sublinear time on a random-access machine (RAM). Moreover, each party is required to maintain state that is only (essentially) linear in its own input size. Our approach combines generic secure two-party computation with *oblivious RAM* (ORAM) protocols. We present an optimized version of our approach using Yao's garbled-circuit protocol and a recent ORAM construction of Shi et al.

We describe an implementation of our resulting protocol, and evaluate its performance for obliviously searching a database with over 1 million entries. Our implementation outperforms off-the-shelf secure-computation protocols for databases containing more than 2^{18} entries.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General–security and protection

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1651-4/12/10 ...$15.00.

Keywords

Theory, Security, Cryptography, Secure Computation

1. INTRODUCTION

Consider the task of searching over a sorted database of n items. Using binary search, this can be done in time $O(\log n)$. Now consider a secure version of this task where a client wishes to learn whether an item is in a database held by a server, with neither party learning anything more. Applying generic secure computation [23, 6] to this task, we would begin by expressing the computation as a (binary or arithmetic) circuit of size at least n, resulting in a protocol of complexity $\Omega(n)$. Moreover, (at least) linear complexity is *inherent*: in any secure protocol for this problem the server must "touch" every entry of the database; otherwise, the server learns information about the client's input by observing which entries of its database were never accessed.

This linear lower bound seems to rule out the possibility of ever performing practical secure computation over very large datasets. However, tracing the sources of the inefficiency, one may notice two opportunities for improvement:

- Many interesting functions (such as binary search) can be computed in *sublinear* time on a random-access machine (RAM). Thus, it would be nice to have protocols for generic secure computation that use RAMs — rather than circuits — as their starting point.
- The fact that linear work is inherent for secure computation of any non-trivial function f only applies when f is computed *once*. However, it does not rule out the possibility of doing better, in an amortized sense, when the parties compute the same function *multiple* times.

Inspired by the above, we explore scenarios where secure computation with *sublinear* amortized work is possible. We focus on a setting where a client and server repeatedly evaluate a function f, maintaining state across these executions, with the server's (huge) input D changing only a little between executions, and the client's (small) input x chosen anew each time f is evaluated. (It is useful to keep in mind the concrete application of a client making several read/write requests to a large database D, though our results are more general.) Our main result is:

Theorem 1. *Suppose f can be computed in time t and space s in the RAM model of computation. Then there is a secure two-party protocol for f in which the client and server run in amortized time $O(t) \cdot \mathsf{polylog}(s)$, the client uses space $O(\log(s))$, and the server uses space $s \cdot \mathsf{polylog}(s)$.*

The above holds in the semi-honest adversarial model.

We show a generic protocol achieving the above bounds by applying any protocol for secure two-party computation in a particular way to any *oblivious RAM* (ORAM) construction [7]. This demonstrates the feasibility of secure computation with sublinear amortized complexity. We then explore a concrete, optimized instantiation of our protocol based on the recent ORAM construction of Shi et al. [19], and using Yao's garbled-circuit approach [23] for the secure two-party computation. We chose the ORAM construction of Shi et al. since it is the simplest scheme we know of, it has poly-logarithmic *worst-case* complexity (as opposed to other schemes that only achieve this in an amortized sense), it requires small client state, and its time complexity in practice (i.e., taking constant factors into account) is among the best known. (In Section 6 we briefly discuss why we expect other schemes to yield worse overall performance for our application.) We chose Yao's garbled-circuit approach for secure computation since several recent results [10, 15] show that it is both quite efficient and can scale to handle circuits with tens of millions of gates. When combining these two schemes, we apply a number of optimizations to reduce the sizes of the circuits that need to be evaluated using generic secure computation.

We implemented the optimized protocol described above, and evaluated it for the task of database search. For small databases our protocol is slower than standard protocols for secure computation, but our protocol outperforms the latter for databases containing more than 2^{18} entries.

1.1 Technical Overview

Our starting point is the ORAM primitive, introduced in [7], which allows a client (with a small memory) to perform RAM computations using the (large) memory of a remote untrusted server. At a high level, the client stores encrypted entries on the server, and then emulates a RAM computation of some function by replacing each read/write access of the original RAM computation with a series of read/write accesses such that the actual access pattern of the client remains hidden from the server. Existing ORAM constructions have the following complexity for an array of length s: the server's storage is $s \cdot \mathsf{polylog}(s)$; the client's storage is $O(\log s)$; and the (amortized) work required to read/write one entry of the array is $\mathsf{polylog}(s)$.

The above suggests a method for computing $f(x, D)$ for any function f defined in the random-access model of computation, where the client holds (small) input x and the server holds (large) input D: store the memory array used during the computation on the server, and have the client access this array using an ORAM scheme. This requires an (expensive) pre-processing phase during which the client and server initialize the ORAM data structure with D; after this, however, the client and server can repeatedly evaluate $f(x_i, D)$ (on different inputs $x_1, \ldots$ of the client's choice) very efficiently. Specifically, if f can be evaluated in time t and space s on a RAM, then each evaluation of f in this client/server model now takes (amortized) time $t \cdot \mathsf{polylog}(s)$.

The above approach, however, only provides "one-sided security," in that it ensures privacy of the client's input against the server; it provides no security guarantees for the server against the client! We can address this by having the parties compute the next ORAM instruction "inside" a (standard) secure two-party computation protocol, with the intermediate state being shared between the client and server. The resulting ORAM instruction is output to the server, who can then read/write the appropriate entry in the ORAM data structure that it stores, and incorporate the result (in case of a read operation) in the shared state. The key observations here are that (1) it is ok to output the ORAM instructions to the server, since the ORAM itself ensures privacy for the client; thus, secure computation is needed only to determine the next instruction that should be executed. Moreover, (2) each computation of this "next-instruction function" is performed on *small* inputs whose lengths are *logarithmic* in s and independent of t: specifically, the inputs are just (shares of) the current state for the RAM computation of f (which we assume to have size $O(\log s)$, as is typically the case) and (shares of) the current state for the ORAM itself (which has size $O(\log s)$). Thus, the asymptotic work for the secure computation of f remains unchanged.

For our optimized construction, we rely on the specific ORAM construction of Shi et al. [19], and optimized versions of Yao's garbled-circuit protocol. We develop our concrete protocol with the aim of minimizing our reliance on garbled circuits for complex functionalities. Instead, we perform local computations whenever we can do so without losing security. For example, we carefully use encryption scheme where block-cipher computations can be done locally, with just an XOR computed via secure computation. For the parts of our protocol that do utilize generic secure computation, we rely on garbled-circuit optimization techniques such as the free-XOR approach [12, 2], oblivious-transfer extension [11], and pipelined circuit execution [10]. Wee also use precomputation (e.g., [1]) to push expensive computations to a preprocessing stage. Our resulting scheme only requires simple XOR operations for oblivious-transfer computations in an on-line stage, while exponentiations and even hashing can be done as part of preprocessing.

1.2 Related Work

Ostrovsky and Shoup [17] also observed that ORAM and secure computation can be combined, though in a different context and using a different approach. Specifically, they consider a (stateless) client storing data on *two* servers that are assumed not to collude. They focus on *private storage* of the data belonging to the client, rather than secure computation of a function over inputs held by a client and server as we do here. Finally, they do not evaluate the concrete efficiency of their approach.

Damgård et al. [3] also observe that ORAM can be used for secure computation. In their approach, which they only briefly sketch, players share the *entire* (super-linear) state of the ORAM, in contrast to our protocol where the client maintains only logarithmic state. They make no attempt to optimize the concrete efficiency of their protocol, nor do they offer any implementation or evaluation of their approach.

Though the above two works have a flavor similar to our own, our work is the first to explicitly point out that ORAM can be used to achieve secure two-party computation with sublinear complexity (for functions that can be computed in sublinear time on a RAM).

Oblivious RAM was introduced in [7], and in the past few years several improved constructions have been proposed (c.f. [21, 22, 18, 8, 9, 13, 19, 20]). Due to space limitations, we refer the reader to [19, 20] for further discussion and pointers to the sizeable literature on this topic.

2. DEFINITIONS

2.1 Random Access Machines

In this work, we focus on RAM programs for computing a function $f(x, D)$, where x is a "small" input that can be read in its entirety and D is a (potentially) large input that is viewed as being stored in a memory array that we also denote by D and that is accessed via a sequence of read/write instructions. Any such instruction $I \in (\{\mathsf{read}, \mathsf{write}\} \times \mathbb{N} \times \{0,1\}^\ell)$ takes the form (write, v, d) ("write data element d in location/address v") or $(\mathsf{read}, v, \perp)$ ("read the data element stored at location v"). We also assume a designated "stop" instruction of the form (stop, z) that indicates termination of the RAM protocol with output z.

Formally, a RAM program is defined by a "next instruction" function Π which, given its current state and a value d (that will always be equal to the last-read element), outputs the next instruction and an updated state. Thus if D is an array of n entries, each ℓ bits long, we can view execution of a RAM program as follows:

- Set $\mathsf{state}_\Pi = (1^{\log n}, 1^\ell, \mathsf{start}, x)$ and $d = 0^\ell$. Then until termination do:
    1. Compute $(I, \mathsf{state}'_\Pi) = \Pi(\mathsf{state}_\Pi, d)$. Set $\mathsf{state}_\Pi = \mathsf{state}'_\Pi$.
    2. If $I = (\mathsf{stop}, z)$ then terminate with output z.
    3. If $I = (\mathsf{write}, v, d')$ then set $D[v] = d'$.
    4. If $I = (\mathsf{read}, v, \perp)$ then set $d = D[v]$.

(We stress that the contents of D may change during the course of the execution.) To make things non-trivial, we require that the size of state_Π, and the space required to compute Π, is polynomial in $\log n, \ell$, and $|x|$. (Thus, if we view a client running Π and issuing instructions to a server storing D, the space used by the client is small.)

We allow the possibility for D to grow beyond n entries, so the RAM program may issue write (and then read) instructions for indices greater than n. The space complexity of a RAM program on initial inputs x, D is the maximum number of entries used by the memory array D during the course of the execution. The time complexity is the number of instructions issued in the execution as described above. For our application, we do not want the running time of a RAM program to reveal anything about the inputs. Thus, we will assume that any RAM program has associated with it a polynomial t such that the running time on x, D is exactly $t(\log n, \ell, |x|)$.

2.2 Oblivious RAM

We view an oblivious-RAM (ORAM) construction as a mechanism that simulates read/write access to an underlying (virtual) array D via accesses to some (real) array $\tilde{D}$; "obliviousness" means that no information about the virtual accesses to D is leaked by observation of the real accesses to $\tilde{D}$. An ORAM construction can be used to compile any RAM program into an oblivious version of that program.

An ORAM construction consists of two algorithms OI and OE for initialization and execution, respectively. OI initializes some state $\mathsf{state}_{\mathsf{oram}}$ that is used (and updated by) OE. The second algorithm, OE, is used to compile a single read/write instruction I (on the virtual array D) into a sequence of read/write instructions $\tilde{I}_1, \tilde{I}_2, \ldots$ to be executed on (the real array) $\tilde{D}$. The compilation of an instruction I into $\tilde{I}_1, \tilde{I}_2, \ldots$, can be adaptive; i.e., instruction $\tilde{I}_j$ may depend on the values read in some prior instructions. To capture this, we define an iterative procedure called $\mathsf{doInstruction}$ that makes repeated use of OE. Given a read/write instruction I, we define $\mathsf{doInstruction}(\mathsf{state}_{\mathsf{oram}}, I)$ as follows:

- Set $d = 0^\ell$. Then until termination do:
    1. Compute $(\tilde{I}, \mathsf{state}'_{\mathsf{oram}}) \leftarrow \mathsf{OE}(\mathsf{state}_{\mathsf{oram}}, I, d)$, and set $\mathsf{state}_{\mathsf{oram}} = \mathsf{state}'_{\mathsf{oram}}$.
    2. If $\tilde{I} = (\mathsf{done}, z)$ then terminate with output z.
    3. If $\tilde{I} = (\mathsf{write}, v, d')$ then set $\tilde{D}[v] = d'$.
    4. If $\tilde{I} = (\mathsf{read}, v, \perp)$ then set $d = \tilde{D}[v]$.

If I was a read instruction with $I = (\mathsf{read}, v, \perp)$, then the final output z should be the value "written" at $D[v]$. (See below, when we define correctness.)

Correctness. We define correctness of an ORAM construction in the natural way. Let $I_1, \ldots, I_k$ be any sequence of instructions with $I_k = (\mathsf{read}, v, \perp)$, and $I_j = (\mathsf{write}, v, d)$ the last instruction that writes to address v. If we start with $\tilde{D}$ initialized to empty and then run $\mathsf{state}_{\mathsf{oram}} \leftarrow \mathsf{OI}(1^\kappa)$ followed by $\mathsf{doInstruction}(I_1), \ldots, \mathsf{doInstruction}(I_1)$, then the final output is d with all but negligible probability.

Security. The security requirement is that for any two equal-length sequences of RAM instructions, the (real) access patterns generated by those instructions are indistinguishable. We will use the standard definition from the literature, which assumes the two instruction sequences are chosen in advance.[1] Formally, let $\mathcal{ORAM} = \langle \mathsf{OI}, \mathsf{OE} \rangle$ be an ORAM construction and consider the following experiment:

Experiment $\mathsf{ExpAPH}_{\mathcal{ORAM},\mathsf{Adv}}(\kappa, b)$:

1. The adversary Adv outputs two sequences of queries $(\mathbf{I}^0, \mathbf{I}^1)$, where $\mathbf{I}^0 = \{I_1^0, \ldots, I_k^0\}$ and $\mathbf{I}^1 = \{I_1^1, \ldots, I_k^1\}$ for arbitrary k.
2. Run $\mathsf{state}_{\mathsf{oram}} \leftarrow \mathsf{OI}(1^\kappa)$; initialize $\tilde{D}$ to empty; and then execute $\mathsf{doInstruction}(\mathsf{state}_{\mathsf{oram}}, I_1^b)$, ..., $\mathsf{doInstruction}(\mathsf{state}_{\mathsf{oram}}, I_k^b)$ (note that $\mathsf{state}_{\mathsf{oram}}$ is updated each time $\mathsf{doInstruction}$ is run). The adversary is allowed to observe $\tilde{D}$ the entire time.
3. Finally, the adversary outputs a guess $b' \in \{0, 1\}$. The experiment evaluates to 1 iff $b' = b$.

[1] It appears that existing ORAM constructions are secure even if the adversary is allowed to adaptively choose the next instruction after observing the access pattern on $\tilde{D}$ caused by the previous instruction, but this has not been claimed by any ORAM construction in the literature.

Definition 1. *An ORAM construction* $\mathcal{ORAM} = \langle \mathsf{OI}, \mathsf{OE} \rangle$ *is* access-pattern hiding *if for every* PPT *adversary* Adv *the following is negligible:*

$$\left| \Pr\left[\mathsf{ExpAPH}_{\mathcal{ORAM},\mathsf{Adv}}(1^\kappa, b) = 1 \right] - \frac{1}{2} \right|$$

2.3 Secure Computation

We focus on the setting where a server holds a (large) database D and a client wants to repeatedly compute $f(x, D)$ for different inputs x; moreover, f may also change the contents of D itself. We allow the client to keep (short) state between executions, and the server will keep state that reflects the (updated) contents of D.

For simplicity, we focus only on the two-party (client/server) setting in the semi-honest model but it is clear that our definitions can be extended to the multi-party case with malicious adversaries.

Definition of security. We use a standard simulation-based definition of secure computation [5], comparing a real execution to that of an ideal (reactive) functionality F. In the ideal execution, the functionality maintains the updated state of D on behalf of the server. We also allow F to take a description of f as input (which allows us to consider a single ideal functionality).

The real-world execution proceeds as follows. An environment $\mathcal{Z}$ initially gives the server a database $D = D^{(1)}$, and the client and server then run protocol Π_f (with the client using input init and the server using input D) that ends with the client and server each storing some state that they will maintain (and update) throughout the subsequent execution. In the ith iteration ($i = 1, \ldots$), the environment gives x_i to the client; the client and server then run protocol Π_f (with the client using its state and input x_i, and the server using its state) with the client receiving output out_i. The client sends out_i to $\mathcal{Z}$, thus allowing adaptivity in $\mathcal{Z}$'s next input selection x_{i+1}. At some point, $\mathcal{Z}$ terminates execution by sending a special end message to the players. At this time, an honest player simply terminates execution; a corrupted player sends its entire view to $\mathcal{Z}$.

For a given environment $\mathcal{Z}$ and some fixed value κ for the security parameter, we let $\textsc{real}_{\Pi_f,\mathcal{Z}}(\kappa)$ be the random variable denoting the output of $\mathcal{Z}$ following the specified execution in the real world.

In the ideal world, we let F be a trusted functionality that maintains state throughout the execution. An environment $\mathcal{Z}$ initially gives the server a database $D = D^{(1)}$, which the server in turn sends to F. In the ith iteration ($i = 1, \ldots$), the environment gives x_i to the client who sends this value to F. The trusted functionality then computes

$$(\mathsf{out}_i, D^{(i+1)}) \leftarrow f(x_i, D^{(i)}),$$

and sends out_i to the client. (Note the server does not learn anything from the execution, neither about out_i nor about the updated contents of D.) The client ends out_i to $\mathcal{Z}$. At some point, $\mathcal{Z}$ terminates execution by sending a special end message to the players. The honest player simply terminates execution; the corrupted player may send an arbitrary function of its entire view to $\mathcal{Z}$.

For a given environment $\mathcal{Z}$, some fixed value κ for the security parameter, and some algorithm $\mathcal{S}$ being run by the corrupted party, we let $\textsc{ideal}_{F,\mathcal{S},\mathcal{Z}}(\kappa)$ be the random variable denoting the output of $\mathcal{Z}$ following the specified execution.

Secure initialization protocol

Input: The server has an array D of length n.

Protocol:

1. The participants run a secure computation of $\mathsf{OI}(1^\kappa, 1^s, 1^\ell)$, which results in each party receiving a secret share of the initial ORAM state. We denote this by $[\mathsf{state}_{\mathsf{oram}}]$.
2. For $i = 1, \ldots, n$ do
   (a) The server sets $I = (\mathsf{write}, v, D[v])$) and secret-shares I with the client. Denote the sharing by $[I]$.
   (b) The parties run $([\mathsf{state}'_{\mathsf{oram}}], [\perp]) \leftarrow \mathsf{doInstruction}([\mathsf{state}_{\mathsf{oram}}], [I])$ (see Figure 3), and set $[\mathsf{state}_{\mathsf{oram}}] = [\mathsf{state}'_{\mathsf{oram}}]$.

Figure 1: Secure initialization protocol π_{Init}.

Definition 2. *We say that* protocol Π_f securely computes f *if there exists a probabilistic polynomial-time ideal-world adversary* $\mathcal{S}$ *(run by the corrupted player) such that for all non-uniform, polynomial-time environments* $\mathcal{Z}$ *there exists a negligible function* negl *such that*

$$\left| \Pr\left[\textsc{real}_{\Pi_f,\mathcal{Z}}(\kappa) = 1 \right] - \Pr\left[\textsc{ideal}_{F,\mathcal{S},\mathcal{Z}}(\kappa) = 1 \right] \right| \leq \mathsf{negl}(\kappa).$$

3. GENERIC CONSTRUCTION

In this section we present our generic solution for achieving secure computation with sublinear amortized work, based on any ORAM scheme and any secure two-party computation (2PC) protocol. While our optimized protocol (in Section 4) is more efficient, this generic protocol demonstrates theoretical feasibility and provides a conceptually clean illustration of our overall approach. A high-level overview of our protocol was given in Section 1.1.

We provide our security definition in Appendix 2.3. We briefly describe here the definition of ORAM; formal definitions of the RAM and ORAM models of computation are given in Appendix 2.1 and Appendix 2.2 respectively.

An ORAM provides read/write access to a (virtual) array of length s using a data structure of length $s \cdot \mathsf{polylog}(s)$, where each "virtual" read/write instruction I (in the virtual array of length s) is emulated using $\mathsf{polylog}(s)$ read/write instructions $\hat{I}_1, \ldots$ on the actual ORAM array (of length $s \cdot \mathsf{polylog}(s)$). The underlying ORAM is defined by two algorithms OI and OE. The first represents the initialization algorithm (i.e., key generation), which establishes the client's initial state and can be viewed as also initializing an empty array that will be used as the main ORAM data structure. This algorithm takes as input κ (a security parameter), s (the length of the virtual array being emulated), and ℓ (the length of each entry in both the virtual and actual arrays). The second algorithm OE defines the actual ORAM functionality, namely the process of mapping a virtual instruction I to a sequence of real instructions $\hat{I}_1, \ldots$. Algorithm OE takes as input (1) the current ORAM state $\mathsf{state}_{\mathsf{oram}}$, (2) the virtual instruction I being emulated, and (3) the last value d read from the ORAM array, and outputs (1) an updated ORAM state $\mathsf{state}'_{\mathsf{oram}}$ and (2) the next instruction $\hat{I}$ to run.

Secure evaluation protocol π_f

Inputs: The server has array $\tilde{D}$ and the client has input $n, 1^\ell$, and x. They also have shares of an ORAM state, denoted $[\mathsf{state}_{\mathsf{oram}}]$.

Protocol:

1. The client sets $\mathsf{state}_\Pi = (n, 1^\ell, \mathsf{start}, x)$ and $d = 0^\ell$ and secret-shares both values with the server; we denote the shared values by $[\mathsf{state}_\Pi]$ and $[d]$, respectively.
2. Do:
   - (a) The parties securely compute $([\mathsf{state}'_\Pi], [I]) \leftarrow \Pi([\mathsf{state}_\Pi], [d])$, and set $[\mathsf{state}_\Pi] = [\mathsf{state}'_\Pi]$.
   - (b) The parties run a secure computation to see if $\mathsf{state}_\Pi = (\mathsf{stop}, z)$. If so, break.
   - (c) The parties execute $([\mathsf{state}'_{\mathsf{oram}}], [d']) \leftarrow \mathsf{doInstruction}([\mathsf{state}_{\mathsf{oram}}], [I])$. They set $[\mathsf{state}_{\mathsf{oram}}] = [\mathsf{state}'_{\mathsf{oram}}]$ and $[d] = [d']$.
3. The server sends (the appropriate portion of) its share of $[\mathsf{state}_\Pi]$ to the client, who recovers the output z.

Output: The client outputs z.

Figure 2: Secure evaluation of a RAM program defined by next-instruction function Π.

With the above in place, we can now define our protocol for secure computation of a function f over an input x held by the client (and assumed to be small) and an array $D \in (\{0,1\}^\ell)^n$ held by the server (and assumed to be large). We assume f is defined in the RAM model of computation in terms of a next-instruction function Π which, given the current state and value d (that will always be equal to the last-read element), outputs the next instruction and an updated state. We let s denote a bound on the number of memory cells of length ℓ required by this computation (including storage of D in the first n positions of memory). Our protocol proceeds as follows:

1. The parties run a secure computation of OI. The resulting ORAM state $\mathsf{state}_{\mathsf{oram}}$ is shared between the client and server.
2. The parties run a secure computation of a sequence of (virtual) write instructions that insert each of the n elements of D into memory. The way this is done is described below.
3. The parties compute f by using secure computation to evaluate the next-instruction function Π. This generates a sequence of (virtual) instructions, shared between the parties, each of which is computed as described below.
4. When computation of f is done, the state associated with this computation (state_Π) encodes the output z. The server sends the appropriate portion of its share of state_Π to the client, who can then recover z.

The $\mathsf{doInstruction}$ subroutine

Inputs: The server has array $\tilde{D}$, and the server and client have shares of an ORAM state (denoted $[\mathsf{state}_{\mathsf{oram}}]$) and a RAM instruction (denoted $[I]$).

1. The server sets $d = 0^\ell$ and secret shares this value with the client; we denote the shared value by $[d]$.
2. Do:
   - (a) The parties securely compute $([\mathsf{state}'_{\mathsf{oram}}], [\hat{I}]) \leftarrow \mathsf{OE}([\mathsf{state}_{\mathsf{oram}}], [I], [d])$, and set $[\mathsf{state}_{\mathsf{oram}}] = [\mathsf{state}'_{\mathsf{oram}}]$.
   - (b) The parties run a secure computation to see if $\hat{I} = (\mathsf{done}, z)$. If so, set $[d] = [z]$ and break.
   - (c) The client sends its share of $[\hat{I}]$ to the server, who reconstructs $[\hat{I}]$. Then:
     - i. If $\hat{I} = (\mathsf{write}, v, d')$, the server sets $\tilde{D}[v] = d'$ and sets $d = d'$.
     - ii. If $\hat{I} = (\mathsf{read}, v, \perp)$, the server sets $d = \tilde{D}[v]$.
   - (d) The server secret-shares d with the client.

Output: Each player outputs its shares of $\mathsf{state}_{\mathsf{oram}}$ and d.

Figure 3: Subroutine for executing one RAM instruction.

See Figures 1 and 2 for the secure initialization and secure computation of the RAM next-instruction. In the figures, we let $[v]$ denote a bitwise secret-sharing of a value v between the two parties. It remains to describe how a single virtual instruction I (shared between the two parties) is evaluated. This is done as follows (also see Figure 3):

1. The parties use repeated secure computation of OE to obtain a sequence of real instructions $\hat{I}_1, \ldots$. Each such instruction $\hat{I}$ is revealed to the server, who executes the instruction on the ORAM data structure that it stores. If $\hat{I}$ was a read instruction, then the value d that was read is secret-shared with the client.
2. After all the real instructions have been executed, emulation of instruction I is complete. If I was a read instruction, then the (virtual) value d' that was read is secret-shared between the client and server.

The key point to notice is that *each secure computation that is invoked is run only over* **small** *inputs.* This is what allows the amortized cost of the protocol to be sublinear.

The following summarizes our main theoretical result. The proof is tedious but relatively straightforward; due to space limitations, it is omitted from the present version but is available from the authors upon request.

THEOREM 2. *If an ORAM construction and a 2PC protocol secure against semi-honest adversaries are used, then our protocol securely computes f against semi-honest adversaries. Furthermore, if f can be computed in time t and space s on a RAM, then our protocol runs in amortized time $O(t) \cdot \mathsf{polylog}(s)$, the client uses space $O(\log(s))$, and the server uses space $s \cdot \mathsf{polylog}(s)$.*

We comment that if the underlying secure-computation is secure against *malicious* parties, then a simple change to our protocol will suffice for obtaining security against malicious parties as well. We simply change the outputs of all secure computations to include a signature on the outputs described above (using a signing key held by the other party), and we modify the functions used in the secure-computation such that they verify the signature on each input before continuing. We leave the proof of this informal claim to future work. We note that we cannot make such a claim for our more efficient, concrete solution presented in Section 4.1.

4. OUR OPTIMIZED PROTOCOL

In Section 3 we showed that any ORAM protocol can be combined with any secure two-party computation scheme to obtain a secure computation scheme with sublinear amortized complexity. In this section we present a far more efficient and practical scheme, based on instantiating our generic protocol with Yao's garbled circuits and the ORAM construction of Shi et. al [19]. However, rather than applying the secure computation primitive on the entire ORAM instruction, we deviate from the generic protocol by performing parts of the computation locally, whenever we could do so without violating security. This section describes our scheme, including concrete algorithmic and engineering decisions we made when instantiating our protocol, as well as implementation choices and complexity analysis. In Section 5 we present experimental performance results, demonstrating considerable improvement over using traditional secure computation over the entire input (i.e. without ORAM). We do not describe Yao's garbled circuit technique here, as this has been described in many prior works (see [14] for a very clear exposition). We do, however, attempt to present the discussion in a way that requires minimal knowledge of this particular technique.

The ORAM Construction of Shi et. al. [19].

We begin with an overview of the ORAM construction of [19], which is the starting point of our protocol. The main data storage structure used in this scheme is a binary tree with the following properties. To store N data items in the ORAM, we construct a binary tree of height $\log N$, where each node has the capacity to hold $\log N$ data items. Every item stored in the binary tree is assigned to a randomly chosen leaf node. The identifier of this leaf node is appended to the item, and the item, along with its assignment, is encrypted and stored somewhere on the path between the root and its assigned leaf node. To find a data item, the client begins by retrieving the leaf node associated with that item; we will explain how this is done below. He sends the identifier of the leaf node to the server, who then fetches and sends all items along the appropriate path, one node at a time. The client decrypts the content of each node and searches for the item he is looking for. When he finds it, he removes it from its current node, assigns it a new leaf identifier chosen uniformly at random and inserts the item at the root node of the tree. He then continues searching all nodes along the path in order to prevent the server from learning where he found the item of interest.

Since the above lookup process will work only while there is room in the root node for new incoming items, the authors of [19] devise the following load balancing mechanism to prevent nodes from overflowing. After each ORAM access instruction, two nodes are chosen at random from each level of the tree. One arbitrary item is evicted from each of these nodes, and is inserted in the child node that lies on the path towards its assigned leaf node. While the server will learn which nodes were chosen for this eviction, it should not learn which children receive the evicted items. To hide this information, the client insert encrypted data in *both* of the two child nodes, performing a "dummy" insertion in one node, and a real insertion in the other.

All that remains to describe is how the client recovers the leaf identifier associated with the item of interest. The number of such identifiers is linear in the size of the database, so storing the identifiers on the client side is not an option. The solution is to store these assignments on the server, recursively using the same type of binary trees. A crucial property which makes this solution possible is that an item can store more than a single mapping. If an item stores r mappings, then the total number of recursively built trees is $\log_r N$. The smallest tree will have very few items, and can thus be stored by the client. As an example, let the largest tree contain items with virtual addresses $v_1^{(1)}, \ldots, v_N^{(1)}$ that are assigned leaf identifiers $L_1^{(1)}, \ldots, L_N^{(1)}$. Then the tree at level 2 has $\frac{N}{r}$ items with virtual addresses $v_1^{(2)}, \ldots, v_{\frac{N}{r}}^{(2)}$, where the item with virtual address $v_j^{(2)}$ contains mappings $(v_i^{(1)}, L_i^{(1)})$ for $(j-1)r < i \leq jr$. With this modification, an ORAM lookup consists of a series of lookups, one in each of these trees, beginning with the smallest tree. In particular, given a virtual address v for a database query, the client derives the lookup values that he needs to use in tree i by computing $v^{(i)} = \lfloor \frac{v}{r^i} \rfloor$ for $0 \leq i \leq \log_r N$. Having these values the client starts with a lookup in the smallest tree for value $v^{(\log_r N)}$. He retrieves $L^{(\log_r N)}$ from his local memory and finds in it the mapping $(v^{(\log_r N-1)}, L^{(\log_r N-1)})$. Now he looks for $v^{(\log_r N-1)}$ in the next smallest tree using leaf identifier $L^{(\log_r N-1)}$. This process continues until the client retrieves the real database item at address v from the largest tree at the top level of the recursion. In each tree, the accessed item is assigned a new leaf node at random, and the item is inserted back in the tree's root node. In addition, its mapping is updated in the tree below to record its new leaf node.

The intuition for the security of this scheme can be summarized as follows. Every time the client looks up item v_i, he assigns it a new leaf node and re-inserts it at the root. It follows that the paths taken to find v_i in two different lookups are independent of one another, and cannot be distinguished from the lookup of any other two nodes. During the eviction process, a node is just as likely to accept a new item as it is to lose an item. Shi et al. prove in their work that with buckets of size $O(\log(MN/\delta))$ the probability that a bucket will overflow after M ORAM instructions is less than δ. It follows that with a bucket size of $O(\log N)$, the probability of overflow is negligible in the security parameter. However, as we shall see below, the precise constant makes a big difference, both in the resulting security and in the efficiency of the scheme.

4.1 High Level Protocol

As above, we assume a database of N items, and we allow each item in each recursive level to hold r mappings between virtual addresses and leaf identifiers from the level above. The client and a server perform the following steps to access an item at an address v:

1. The parties have shares v_C and v_S of the virtual address for the query in the database $v = v_C \oplus v_S$.

2. The client and the server run a two party computation protocol to produce shares $v_C^{(1)}, \ldots, v_C^{(\log_r N)}$ and

$v_S^{(1)}, \ldots, v_S^{(\log_r N)}$ of the virtual addresses that they will lookup in each tree of the ORAM storage: $\lfloor \frac{v}{r^i} \rfloor = v_C^{(i)} \oplus v_S^{(i)}$ for $0 \leq i \leq \log_r N$

3. The server generates random leaf identifiers $\tilde{L}^{(1)}, \ldots, \tilde{L}^{(\log_r N)}$ that will be assigned to items as they are re-inserted at the root.

4. The last tree in the ORAM storage has only a constant number of nodes, each containing a constant number of items. The client and server store shares of the leaf identifiers for these items. They execute a two party protocol that takes these shares as inputs, as well as the shares $v_C^{(\log_r N)}$ and $v_S^{(\log_r N)}$. The server's output of the secure computation is the leaf value $L^{(\log_r N)}$. The client has no output.

5. For each i such that $\log_r N \geq i \geq 2$:

   (a) The server retrieves the nodes on the path between the root and the leaf $L^{(i)}$ in the i-th tree.

   (b) The parties execute a secure two party protocol. The server's inputs are the nodes recovered above, and the secret share $v_S^{(i-1)}$. The client's input is $v_C^{(i-1)}$. The server receives value $L^{(i-1)}$ as output, which is the value stored at address $v_C^{(i-1)} \oplus v_S^{(i-1)}$, and which lies somewhere along the path to $L^{(i)}$.

   (c) The parties execute a secure two party protocol to update the content of item $v^{(i)}$ with the value of the new leaf identifier $\tilde{L}^{(i-1)}$ that will be assigned to $v_C^{(i-1)} \oplus v_S^{(i-1)}$ in the $i-1$-th tree.

   (d) The parties execute a secure two party protocol to tag item $v^{(i)}$ with it's new leaf node assignment $\tilde{L}^{(i)}$, and to insert $v^{(i)}$ in the first empty position of the root node.

6. For the first level tree that contains the actual items for the database, the server retrieves the nodes on the path between the root and the leaf $L^{(1)}$. The parties execute a secure two party protocol to find item $v = v_C^{(1)} \oplus v_S^{(1)}$. The outputs of the protocol are secret shares of the data $d = d_C \oplus d_S$ found at virtual address v. The server tags v with $\tilde{L}^{(1)}$, and the parties perform another secure protocol to insert v at the first empty spot in the root node.

4.2 Optimizations and Implementation Choices

Encryption and Decryption.

In our protocol description so far, we have left implicit the fact all data stored in the database at the server must be encrypted. Every time a data item is used in the RAM computation, it must first be decrypted, and it must be re-encrypted before it is re-inserted at the root node. In a naive application of our generic solution, the parties would have to decrypt, and later re-encrypt the data item completely inside a Yao circuit, which can be very time consuming. We choose the following encryption scheme, with an eye towards minimizing the computation done inside the garbled circuit: $\mathsf{Enc}(m; r) = (F_K(r) \oplus m, r)$, where F can be any pseudo-random function. The key K is stored by the client, and kept secret from the server. To ensure that encryption and decryption can be efficiently computed inside a garbled circuit, the server sends r to the client in the clear, along with a random r' that will be used for re-encryption. The client computes $F_K(r)$ and $F_K(r')$ *outside* the secure computation. Now the only part of decryption or re-encryption that has to be done inside the garbled circuit is boolean XOR, which is very cheap.

While this greatly improves the efficiency of our scheme, we note that it has to be done with care: sending the encryption randomness to the client could reveal information about the access pattern of the RAM, and, consequently, about the server's data. The issue arises during the eviction procedure, when a data item is moved from a parent to one of its children. During this process, it is important that neither player learn which child received the evicted data; the construction of Shi et al. [19] has the client touch both children, writing a dummy item to one of the two nodes, and the evicted item to the other node, thereby hiding from the server which node received the real item. In our case, this must be hidden from the client as well, which is ensured by performing the operation inside a secure computation. However, the exact way in which randomness is assigned to ciphertext has a crucial effect on security. For example, suppose the server sends randomness r_1 and r_2 to be used in the re-encryption, and our operation is designed so that r_1 is always used for encrypting the dummy item and r_2 is always used for the real item. The client can then keep track of the real item by waiting to receive r_2 for decryption in the future! We must therefore design the re-encryption operation so that randomness is associated with a node in the tree rather than the content of the ciphertext. Then, r_1 is always used in the left child, and r_2 in the right, independent of which node receives the real item and which receives the dummy item.

Although this issue is easily handled[2], it demonstrates the subtlety that arises when we depart from the generic protocol in order to improve the efficiency of the scheme.

Choosing a Bucket Size.

At each node of the ORAM structure we have a bucket of items, and choosing the size of each bucket can have a big impact on the efficiency of the scheme: we have to perform searches over $B \log N$ items for buckets of size B. However, if the buckets are too small, there is a high probability that some element will "overflow" its bucket during the eviction process. This overflow event can leak information about the access pattern, so it is important to choose large enough buckets. Shi et al. [19] prove that in an ORAM containing N elements, if the buckets are of size $(log(MN/\delta))$, then after M memory accesses, the probability of overflow is less than δ. It follows that to get, say, security 2^{-20}, it suffices to have buckets of size $O(\log N)$, but the constant in the notation is important.

In Figure 4 we provide our results from simulating over-

[2] We defer the proof of security to the full version. However, to give some intuition, note that as long as the assignment of the encryption randomness is independent of the access pattern, nothing can be learned by the client during decryption. To make this formal, we show that we can simulate his view by choosing random values for each bucket, storing them between lookups, and sending those same values the next time that bucket is scanned. This simulation would fail only if the assignment of the random values to buckets were somehow dependent on the particular content of the RAM.

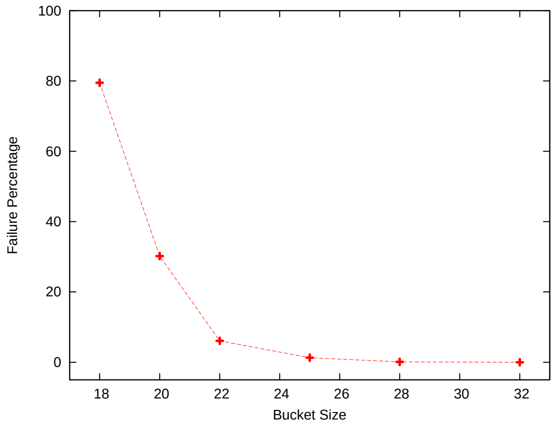

Figure 4: Overflow probability as a function of bucket size, for 65536 virtual instructions on a database of 65536 items.

flow for various bucket sizes. Notice that the value approaches 0 only as we approach $2 \log N$, and in fact the probability of failure is very high for values closer to $\log N$. Based on these simulations, we have chosen to use buckets of size $2 \log N$. We ran our experiment with $N = 2^{16}$ and estimated the probability of overflowing any bucket when we insert all N items, and then perform an additional N operations on the resulting database. We used 10,000 trials in the experiment. Note that for the specific example of binary search, we only need to perform $\log N$ operations on the database; for 2^{16} elements and a bucket size of 32, we determined with confidence of 98% that the probability of overflow is less than .0001. The runtime of our protocol (roughly) doubles when our bucket size doubles, so although we might prefer still stronger security guarantees, increasing the bucket size to $3 \log N$ will have a considerable impact on performance.

Computing Addresses Recursively.

Recall that the leaf node assigned to item $v^{(i)}$ in the ith tree is stored in item $v^{(i+1)} = \lfloor \frac{v^{(i)}}{r} \rfloor$ of the $i+1$th tree. In Step 2, where the two parties compute shares of $v^{(i)}$ for each i in $1, \ldots, \log_r N$, we observe that if r is a power of 2, each party can compute its own shares locally from its share of v. If $r = 2^j$ and $v = v_C \oplus v_S$, then we can obtain $v^{(i)} = \lfloor \frac{v}{r^i} \rfloor$ by deleting the last $i \cdot j$ bits of v. Similarly $v_C^{(i)}$ and $v_S^{(i)}$ can be obtained by deleting the last $i \cdot j$ bits from the values v_C and v_S. This allows us to avoid performing another secure computation when recovering shares of the recursive addresses.

Node Storage Instantiation.

Shi et al. [19] point out that the data stored in each node of the tree could itself be stored in another ORAM scheme, either using the same tree-based scheme described above, or using any of the other existing schemes. We have chosen to simply store the items in an array, performing a linear scan on the entire node. For the data sets we consider, $N = 10^6$ or 10^7, and $20 \leq \log N \leq 25$. Replacing a linear scan with an ORAM scheme costing $O(\log^3 N)$, or even $O(\log^2 N)$, simply does not pay off. We could consider the simpler ORAM of Goldreich and Ostrovsky [7] that has overhead $O(\sqrt{N})$, but the cost of shuffling around data items and computing pseudorandom functions inside garbled circuits would certainly erase any savings.

Using Client Storage.

When the client and server search for an item v, after they recover the leaf node assigned to v, the server fetches the $\log N$ buckets along the path to the leaf, each bucket containing up to $\log N$ items. The parties then perform a secure computation over the items, looking for a match on v. We have a choice in how to carry out this secure computation: we could compare one item at a time with v, or search as many as $\log^2 N$ items in one secure computation. The advantage to searching fewer items is that the garbled circuit will be smaller, requiring less client-side storage for the computation. The disadvantage is that each computation will have to output secret shares of the state of the search, indicating whether v has already been found, and, if so, what the value of its payload is; each computation will also have to take the shares of this state as input, and reconstruct the state before continuing with the search. The extra state information will require additional wires and gates in the garbled circuits, as well as additional encryptions and decryptions for preparing and evaluating the circuit. We have chosen to perform just a single secure computation over $\log^2 N$ items, using the maximal client storage, and the minimal computation. However, we note that the additional computation would have little impact,[3] and we could instead reduce the client storage at relatively little cost. To compute the circuit that searches $\log^2 N$ items, the client needs to store approximately $400,000$ encryption keys, each 80 bits long.

Garbled Circuit Optimizations.

The most computationally expensive part of Yao's garbled circuit protocol is often thought to be the oblivious transfer (OT) sub-protocol [4]. The parties must employ OT once for every input wire of the party that evaluates the circuit, and each such application (naively performed) requires expensive operations such as exponentiations. We use the following known optimizations to reduce OT costs and to further push its computation almost entirely to the preprocessing stage, before the parties begin the computation (even before they have their inputs), reducing the on-line OT computations to just simple XOR operations.

The most important technique we use is the OT extension protocol of Ishai et al. [11], which allows to compute an arbitrary number of OT instances, given a small (security parameter) number of "base" OT instances. We implement the base instances using the protocol of Naor and Pinkas [16], which requires six exponentiations in a prime order group, three of which can be computed during pre-processing. Following [11], the remaining OT instances will only cost us a couple of hash evaluations per instance. We then push these hash function evaluations to the preprocessing stage, in a way that requires only XOR during the on-line stage. Finally, Beaver's technique [1] allows us to start computing the OT's in the preprocessing stage as well, by running OT

[3] This is because sharing the state and reconstructing the state are both done using XOR gates, which are particularly cheap for garbled circuits, as we discuss below.

random inputs for both parties; the output is then corrected by appropriately sending real input XORed with the used random inputs in the online stage.

We rely on several other known garbled circuit optimizations. First, we use the *free XOR gates* technique of Kolesnikov and Schneider [12], which results in more than 60% improvement in the evaluation time for an XOR gate, compared to other gates. Accordingly, we aim to construct our circuits using as few non-XOR gates as possible.

Second, we utilize a wider variety of gates (as opposed to the traditional Boolean AND, OR, XOR, NAND gates). This pays off since in the garbled circuit construction every non-XOR gate requires performing encryption and decryption, and all gates of the same size are equally costly in this regard. In our implementation we construct and use 10 of the 16 possible gates that have 2 input bits and one output bit. We also rely heavily on the multiplexer gate on 3 input bits; this gate uses the first input bit to select the output from the other two input bits. In one circuit, we use a 16-bit multiplexer, which uses 4 input bits to select from 16 other inputs.

Finally, we utilize *pipelined circuit execution*, which avoids the naive traditional approach where one party sends the garbled circuit in its entirety to the second one. This naive approach is often impractical, as for large inputs the garbled circuits can be several gigabytes in size, and the receiving party cannot start the evaluation until the entire garbled circuit has been generated and transmitted and stored in his memory. To mitigate that, we follow the technique introduced by Huang et al. [10], allowing the generation and evaluation of the garbled circuit to be executed in parallel, where the sender can transmit each garbled gate as soon as he generates it, and continue to garble the next gates while the receiver is evaluating the received gates, thus improving the total evaluation time. This also alleviates the memory requirements for both parties since the garbler can discards the gates he has sent, and the receiver can discard a gate that he has evaluated.

5. IMPLEMENTATION

The goal of our experiments was to evaluate and compare execution times for two protocols implementing binary search: one using standard optimized Yao, and the other using our ORAM-based approach described in the previous sections. In our experiments, each of the two parties was run on a different server, each with a Intel Xeon 2.5GHz CPU, 16 GB of RAM, two 500 GB hard disk drives, and running a 64-bit Ubuntu operating system. They each had a 1 Gbit ethernet interface, and were connected through a 1Gbit switch.

Before running our experiments, we first populated the database structure on the server side: in our ORAM protocol, we randomly placed the encrypted data throughout the ORAM structure, and in the Yao protocol performing a linear scan, we simply stored the data in a large array. We then generated and stored the necessary circuit descriptions on each machine. Finally, the two parties interacted to pre-process the expensive part of the OT operations, in a manner that is independent of their inputs. We did not create the garbled gates for the circuits during pre-processing; the server begins generating these once contacted by the client. However, the server sent garbled gates to the client as they were ready, so as to minimize the impact on the total computation time. When we measured time in our experiments, we included: 1) the online phase of the OT protocol, 2) the time it takes to create the garbled gates and transfer the resulting ciphertexts, and 3) the processing time of the garbled circuits.

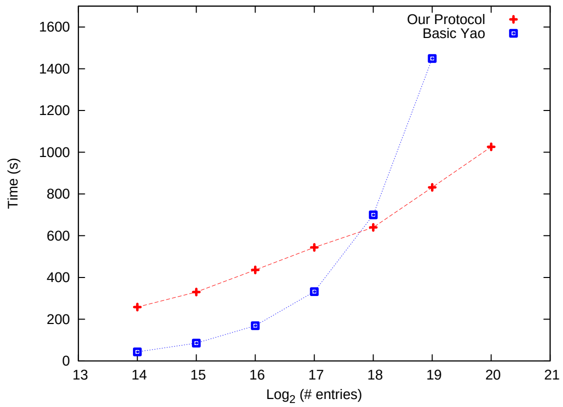

Figure 5: Time for performing binary search using our protocol vs. time for performing binary search using a standard garbled-circuit protocol as a function of the number of database entries. Each entry is 512 bits long.

5.1 Performance

In Figure 5, we compare the performance of our construction when computing a ORAM-based binary search to the performance of a Yao-based linear scan. We have plotted the x-axis on a logarithmic scale to improve readability. From the plot it can be seen that we outperform the Yao linear scan by a factor of 3 when dealing with input of size 2^{19}, completing the $\log N$ operations in less than 7 minutes, compared to 24 minutes for Yao. For input of size 2^{20}, we complete our computation in 8.3 minutes, while the Yao implementation failed to complete (we were unable to finish the linear scan because the OS began swapping memory). While we had no trouble running our ORAM-based protocol on input of size 2^{20}, for $N = 2^{21}$, we ran out of memory when populating the server's ORAM during pre-processing.

In Figure 6 we demonstrate how our protocol performs when evaluating a single read operation over N data elements of size 512 bits, for $N \in \{2^{16}, 2^{17}, 2^{18}, 2^{19}, 2^{20}\}$. We note that runtime for binary search using the ORAM is almost exactly the time it takes to run $\log N$ single lookups; this is expected, since the circuit for computing the next RAM instruction is very small. For 2^{16} items and a bucket size of 32, a single operation takes 27 seconds, while for 2^{20} items and buckets of size 40, it takes about 50 seconds. Recall that when relying only on secure computation, computing *any* function, even those making a constant number of lookups, requires a full linear scan; in this scenario, the performance gain is more than 30-fold. One example of such a function allows the client to search a large social network, in order to fetch the neighborhood of a particular node.

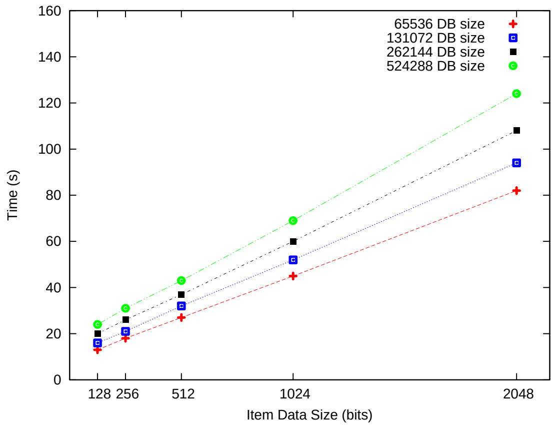

Figure 6: Single ORAM lookup times for different database sizes and item data lengths.

5.2 Discussion

Memory Constraints.

Memory is the primary limitation on scaling the computation to larger values of N. For the linear scan, the problem stems from the size of the circuit description, which is more than 23 gigabytes and 850 million wires, if $N = 2^{19}$ and the data elements are 512 bits. The pipe-lining technique of Huang et al. [10] prevents the parties from storing all 23 gigabytes in RAM, but the client still stores an 80 bit secret key for every wire in the circuit, and the server stores two; this requires 8.5 gigabytes of memory for the client and 17 gigabytes for server. This ends up requiring far more space than the data itself, which is only $512N = 33$ megabytes.

In contrast, when $N = 2^{19}$ and the data size is 512, the largest circuit in our protocol is less than 50 megabytes, and contains about 1million wires. On the other hand, each level of the data storage has a factor of $4 \log N$ overhead (when our bucket size is $2 \log N$), so server storage for the top level alone is more than $40000N = 2.5$ gigabytes. This explains why we eventually ran into trouble when pre-processing the data; to broaden the scale of what we can handle, we will need to improve the way we handle memory while inserting elements into the ORAM structure.

Pre-processing.

We have not done any calculations regarding the time required for secure pre-processing. As explained above, when running our experiments, we populated the ORAM structure by randomly placing items in the trees. This is of course insecure, since the server will know where all the items are in the ORAM: to ensure security, the insertion of the data would have to be interactive. One naive way to ensure security is to insert each item, one at a time, by performing the "write" protocol inside a secure computation, precisely as we have described an ORAM lookup. If we start this process with a data structure large enough to hold all items, we can estimate the time it will take to insert 2^{16} elements of 512 bits each, by multiplying the 13 seconds we require for a write operation by 2^{16}. It seems this would take almost 20 days to compute! We leave the problem of finding a more efficient method for data insertion to future work.

One natural approach would be to start with smaller structures, repeatedly doubling their size in some secure manner as insertion progresses. We stress that the pre-processing we do in our work is *fully secure* in a three-party model, where the database owner pre-processes his data, and then transfers the encrypted data to a semi-honest third party, who performs the secure computation on his behalf.

The Recursion Parameter.

In all of our experiments, we have chosen $r = 16$; that is, every item in tree $i > 1$ stores the leaf nodes of 16 items from tree $i-1$. This is a parameter that we could change, and it may have an impact on performance. However, one parameter we did investigate is the choice of how far to recurse. As can be seen in Table 1, the best performance occurs when the bottom level, which requires a linear scan, holds fewer than 2^{12} items. Interestingly, beyond that, further recursion does not seem to make a difference. The ith tree

DB size	2 trees	3 trees	4 trees	5 trees
2^{20}	35	14	**12.5**	13
2^{19}	20	**11.5**	12.5	-
2^{18}	12.5	**9.5**	**9.5**	-

Table 1: Time in seconds of a single ORAM access, with various numbers of recursion levels in the ORAM structure. The number of items in the bottom level is 2^{N-4i+4} when there are i trees.

Counting Gates.

Let N be the number of elements, let d be the length of each element, and let B denote the bucket size of each node. We calculate the number of non-XOR gates in the garbled circuits of our ORAM operation, and provide some relevant observations. We first consider the top level tree that contains the database items. During a lookup we need to check $\log N$ nodes along the path to the leaf associated with the searched item. Each of these nodes contains B elements of size $\log N + d$: a virtual address of size $\log N$ and a data element of size d. We use approximately 1 non-XOR gates for each of these. Therefore, a single lookup consists of $B \log N(\log N + d)$ non-free gates. In the eviction process that follows, we scan $2 \log N$ nodes for eviction, and write to both children of each node (one write is dummy). Thus, the eviction circuits require $6B \log N(\log N + d)$ non-free gates, which gives us a total of $7B \log N(\log N + d)$ non-free gates for each ORAM operation in the top level tree. The analysis at the lower level trees is similar, but asymptotically, this dominates the computation, since the lower level trees have only $N/16^i$ elements. We provide concrete numbers in Table 2, taken directly from our circuits. We considere $B = 2 \log N$ and d=512. We note that our circuits grow linearly in the size of each bucket. Also interesting is that it grows linearly in d. Since the Yao linear scan is also linear in the data size, with dN gates, we see that varying the length of the data element will have little impact on our comparison.

6. USING OTHER ORAM SCHEMES

In our concrete protocol we instantiated (and then optimized) our generic construction using the tree-based ORAM scheme of [19]. However, there are several other oblivious

DB size	XOR gates	Non-free gates	Wires
2^{20}	19,159,883	3,730,546	44,039,222
2^{19}	16,519,818	3,166,420	37,656,448
2^{18}	14,219,281	2,700,966	30,941,947
2^{17}	12,185,264	2,302,208	27,366,108
2^{16}	10,377,527	1,954,042	23,655,368

Table 2: Gate and wires counts for different size databases with item data of length 512

RAM schemes which we considered as possible instantiations for our ORAM component. We discovered that these schemes would entail higher complexity in the context of a two party computation protocol[4] since they involve more complicated building blocks such as pseudorandom shuffling protocol and Cuckoo hashing.

For example, the ORAM protocol of Goldreich and Ostrovsky [7] introduced the basic hierarchical structure that underlies many subsequent ORAM protocols. This approach crucially relies on two components that turn out to be quite inefficient when evaluated with a secure two-party computation: (1) the use of a pseudorandom function (PRF) in order to consistently generate a random mapping from virtual addresses to physical addresses; and (2) a joint shuffling procedure for mixing the different levels in the ORAM data structure. We direct the reader to [7] for the full details of the scheme.

Several more-recent ORAM solutions [18, 8, 9, 13] rely on cuckoo hashing (in addition to also using PRF computations). For their security, a new construction for a cuckoo hash table is needed [8], which involves building the corresponding cuckoo graph and conducting breadth-first search on the graph in order to allocate each new item inserted into the cuckoo table. Compiling this step into a secure two-party computation protocol seems likely to introduce a prohibitive performance hit.

7. CONCLUSION

In this work we showed efficient protocols for secure two party computation achieving only a small polylogarithmic overhead over the running time of the insecure version of the same functionality. This is a significant asymptotic improvement over traditional generic secure computation techniques, which inherently impose computation overhead at least linear in the input size. Our protocols rely on any (arbitrary) underlying oblivious RAM and generic two party computation protocols. We further investigate the most efficient instantiation of the protocol and demonstrated, empirically, the expected theoretical improvement. In particular, we implemented a protocol that performs a single access to a databases of size 2^{18} elements, outperforming an implementation of basic secure computation by a factor of 60. This translates also to a three-fold improvement in the running time of binary search. In addition to these concrete improvements, our work sheds light on many of the details faced when implementing ORAM and secure computation.

Acknowledgments

This work was supported in part by NSF and DARPA.

[4] Note that a better performing ORAM protocol does not necessarily translate to a better performing protocol when put through a generic secure computation.

8. REFERENCES

[1] D. Beaver. Precomputing oblivious transfer. In *Advances in Cryptology — Crypto '95*, volume 963 of *LNCS*, pages 97–109. Springer, 1995.

[2] S.G. Choi, J. Katz, R. Kumaresan, and H.-S. Zhou. On the security of the free-XOR technique. In *9th Theorey of Cryptography Conference — TCC 2012*, volume 7194 of *LNCS*, pages 39–53. Springer, 2012.

[3] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *8th Theory of Cryptography Conference — TCC 2011*, volume 6597 of *LNCS*, pages 144–163. Springer, 2011.

[4] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Comm. ACM*, 28(6):637–647, 1985.

[5] O. Goldreich. *Foundations of Cryptography. Volume I: Basic Tools.* Cambridge University Press, 2001.

[6] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229. ACM Press, 1987.

[7] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[8] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *38th Intl. Colloquium on Automata, Languages, and Programming (ICALP), Part II*, volume 6756 of *LNCS*, pages 576–587. Springer, 2011.

[9] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 157–167. ACM-SIAM, 2011.

[10] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *20th USENIX Security Symposium*, 2011.

[11] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology — Crypto 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.

[12] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *35th Intl. Colloquium on Automata, Languages, and Programming (ICALP), Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.

[13] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 143–156. ACM-SIAM, 2012.

[14] Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[15] L. Malka and J. Katz. VMCrypt — modular software architecture for scalable secure computation. Available at `http://eprint.iacr.org/2010/584`.

[16] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 448–457. ACM-SIAM, 2001.

[17] R. Ostrovsky and V. Shoup. Private information storage. In *29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 294–303. ACM Press, May 1997.

[18] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *Advances in Cryptology — Crypto 2010*, volume 6223 of *LNCS*, pages 502–519. Springer, 2010.

[19] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $o((\log n)^3)$ worst-case cost. In *Advances in Cryptology — Asiacrypt 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, 2011.

[20] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*. The Internet Society, 2012.

[21] P. Williams and R. Sion. Usable PIR. In *NDSS*. The Internet Society, 2008.

[22] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *15th ACM Conf. on Computer and Communications Security (CCS)*, pages 139–148. ACM Press, 2008.

[23] A. C.-C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.