ObliVM: A Programming Framework for Secure Computation

Chang Liu*, Xiao Shaun Wang*, Kartik Nayak*, Yan Huang† and Elaine Shi*
*University of Maryland and †Indiana University
{liuchang,wangxiao,kartik,elaine}@cs.umd.edu, yh33@indiana.edu

***Abstract*—We design and develop ObliVM, a programming framework for secure computation. ObliVM offers a domain-specific language designed for compilation of programs into efficient oblivious representations suitable for secure computation. ObliVM offers a powerful, expressive programming language and user-friendly oblivious programming abstractions. We develop various showcase applications such as data mining, streaming algorithms, graph algorithms, genomic data analysis, and data structures, and demonstrate the scalability of ObliVM to bigger data sizes. We also show how ObliVM significantly reduces development effort while retaining competitive performance for a wide range of applications in comparison with hand-crafted solutions. We open-source ObliVM at `www.oblivm.com`, offering a reusable framework to implement oblivious algorithms.**

I. Introduction

Secure computation is a powerful cryptographic primitive that allows multiple parties to perform rich data analytics over their private data, while preserving each individual or organization's privacy. The past decade has witnessed enormous progress in the practical efficiency of secure computation protocols [1]–[6] Quite a few system prototypes [7]–[15] have been built, while several attempts were made to commercialize secure computation techniques [16], [17].

Architecting a system framework for secure computation presents numerous challenges. First, the system must allow *non-specialist programmers* without security expertise to develop applications. Second, *efficiency* is a first-class concern in the design space, and scalability to big data is essential in many interesting real-life applications. Third, the framework must be reusable: *expert programmers* should be able to easily extend the system with rich, optimized libraries or customized cryptographic protocols, and make them available to non-specialist application developers.

We design and build ObliVM, a system framework to automate secure multi-party computation. ObliVM is designed to allow non-specialist programmers to write programs much as they do today, and our ObliVM compiler compiles the program to an efficient secure computation protocol. To this end, ObliVM offers a domain-specific language that addresses the gap between *circuits* (as cryptographic protocol designers perceive computations) and *programs* (as real-life developers' perspective of computations). In architecting ObliVM, our main contribution is the design of programming support and compiler techniques that facilitate such program-to-circuit conversion while ensuring maximal efficiency. Presently, our framework assumes a semi-honest two-party protocol in the back end. Our ObliVM framework, including source code and demo applications, is available at `http://www.oblivm.com`.

A. Background: "Oblivious" Programs and Circuits

To aid understanding, it helps to first think about an intuitive but somewhat imprecise view: Each variable and each memory location is labeled either as *secret* or *public*. Any secret variable or memory contents are secret-shared among the two parties such that neither party sees the values. The two parties run a cryptographic protocol to securely evaluate each instruction, making accesses to memory (public or secret-shared) whenever necessary. All messages transmitted are naturally secured by the underlying cryptographic protocol. However, the parties can additionally observe the following execution traces during the protocol execution: 1) the program counter (also referred to as the instruction trace); 2) addresses of all memory accesses (also referred to as the memory trace); and 3) the value of every public or declassified variable (similar to the notion of a low or declassified variable in standard information flow terminology). It is imperative that the program's observable execution traces (not including the outcome) be "oblivious" to the secret inputs. A more formal security definition involves the use of a simulation paradigm that is standard in the cryptography literature [18], and is similar to the notion adopted in the SCVM work [13].

B. ObliVM Overview and Contributions

In designing and building ObliVM, we make the following contributions.

Programming abstractions for oblivious algorithms. The most challenging part about ensuring a program's obliviousness is memory-trace obliviousness – therefore our discussions below will focus on memory-trace obliviousness. A straightforward approach (henceforth referred to as the generic ORAM baseline) is to provide an Oblivious RAM (ORAM) abstraction, and require that all arrays (whose access patterns depend on secret inputs) be stored and accessed via ORAM. This approach, which was effectively taken by SCVM [13], is generic, but does not necessarily yield the most efficient oblivious implementation for each specific program.

At the other end of the spectrum, a line of research has focused on customized oblivious algorithms for special tasks (sometimes also referred to as circuit structure design). For example, efficient oblivious algorithms have been demonstrated for graph algorithms [19], [20], machine learning algorithms [21], [22], and data structures [23]–[25]. The customized approach can outperform generic ORAM, but is extremely costly in terms of the amount of cryptographic expertise and time consumed.

ObliVM aims to achieve the best of both worlds by offering oblivious programming abstractions that are both user and compiler friendly. These programming abstractions are high-level programming constructs that can be understood and employed by non-crypto-expert programmers. Behind the scenes, ObliVM translates programs written in these abstractions into efficient oblivious algorithms that outperform generic ORAM. When oblivious programming abstractions are not applicable, ObliVM falls back to employing ORAM to translate programs to efficient circuit representations. Presently, ObliVM offers the following oblivious programming abstractions: MapReduce abstractions, abstractions for oblivious data structures, and a new loop coalescing abstraction which enables novel oblivious graph algorithms. We remark that this is not an exhaustive list of possible programming abstractions that facilitate obliviousness. It would be exciting future research to uncover new oblivious programming abstractions and incorporate them into our ObliVM framework.

An expressive programming language. ObliVM offers an expressive and versatile programming language called ObliVM-lang. When designing ObliVM-lang, we have the following goals.

- Non-expert application developers find the language intuitive.
- Easy for expert programmers to extend our framework with new features, e.g., to introduce new oblivious programming abstractions as libraries in ObliVM-lang (Section IV-B).
- Expert programmers can implement even low-level circuit libraries directly atop ObliVM-lang. Recall that unlike a programming language in the traditional sense, here the underlying cryptography fundamentally speaks only of AND and XOR gates. Even basic instructions such as addition, multiplication, and ORAM accesses must be developed from scratch by an expert programmer. ObliVM allows the development of such circuit libraries in the source language, greatly reducing programming complexity. Section V-A demonstrates case studies for implementing basic arithmetic operations and Circuit ORAM atop our source language ObliVM.
- Expert programmers can implement customized protocols in the back end (e.g., faster protocols for performing big integer operations or matrix operations), and export these customized protocols to the source language as native types and native functions.

To simultaneously realize these aforementioned goals, we need a much more powerful and expressive programming language for secure computation than existing ones [8], [12]–[15]. Our ObliVM-lang extends the SCVM language by Liu et al. [13] and offers new features such as phantom functions, generic constants, random types, as well as native types and functions. We will show why these language features are critical for implementing oblivious programming abstractions and low-level circuit libraries.

Additional architectural choices. ObliVM also allows expert programmers to develop customized cryptographic protocols (not necessarily based on Garbled Circuit) in the back end. These customized back end protocols can be exposed to the source language through native types and native function calls, making them immediately reusable by others.

C. Applications and Evaluation

ObliVM's easy programmability allowed us to develop a suite of libraries and applications, including streaming algorithms, data structures, machine learning algorithms, and graph algorithms. These libraries and applications will be shipped with the ObliVM framework. Our application-driven evaluation suggests the following results:

Efficiency. We use ObliVM's user-facing programming abstractions to develop a suite of applications. We show that over a variety of benchmarking applications, the resulting circuits generated by ObliVM can be orders of magnitude smaller than the generic ORAM baseline (assuming that the state-of-the-art Circuit ORAM [26] is adopted for the baseline) under moderately large data sizes.

Development effort. We give case studies to show how ObliVM greatly reduces the development effort and expertise needed to create applications over secure computation.

New oblivious algorithms. We describe several oblivious algorithms that we discovered during this process of programming language and algorithms co-design. Specifically, we demonstrate new oblivious graph algorithms including oblivious Depth-First-Search for dense graphs, oblivious shortest path for sparse graphs, and an oblivious minimum spanning tree algorithm.

D. Threat Model, Deployment, and Scope

Deployment scenarios and threat model. As mentioned, ObliVM presently supports a two-party semi-honest protocol. We consider the following primary deployment scenarios:

1) Two parties, Alice and Bob, each comes with their own private data, and engage in a two-party protocol. For example, Goldman Sachs and Bridgewater would like to perform joint computation over their private market research data to learn market trends.
2) One or more users break their private data (e.g., genomic data) into secret shares, and split the shares among two non-colluding cloud providers. The shares at each cloud provider are completely random and reveal no information. To perform computation over the secret-shared data, the two cloud providers engage in a secure 2-party computation protocol.
3) Similar as the above, but the two servers are within the same cloud or under the same administration. This can serve to mitigate Advanced Persistent Threats or insider threats, since compromise of a single machine will no longer lead to the breach of private data. Similar architectures have been explored in commercial products such as RSA's distributed credential protection [27].

In the first scenario, Alice and Bob should not learn anything about each other's data besides the outcome of the computation. In the second and third scenarios, the two servers should learn nothing about the users' data other than the outcome of the computation – note that the outcome of the computation can also be hidden by XORing the outcome with a secret random mask (like a one-time pad). We assume that the program text (i.e., code) is public.

II. Related Work

Existing general-purpose secure computation systems can be classified in two mostly orthogonal dimensions: 1) the cryptographic protocol used; and 2) whether they offer programming and compiler support.

GC Back End	Features	Garbling Speed	Bandwidth to match compute
FastGC [28]	Java-based	96K gates/sec	2.8MBps
ObliVM-GC (this paper)	Java-based	670K gates/sec, 1.8M gates/sec (online)	19.6MBps 54MBps (online)
GraphSC [21] (extends ObliVM-GC)	Java-based Parallelizable	580K gates/sec per pair of cores 1.4M gates/sec per pair of cores (online)	
JustGarble [2]	C-based Hardware AES-NI Garbling only, does not run end-to-end	11M gates/sec	315MBps
KSS [9]	Parallel execution in malicious mode Hardware AES-NI	320 gates/sec per pair of cores	2.4MBps per pair of cores

TABLE I: **Summary of known (2-party) Garbled Circuit Tools.** The gates/sec metric refer specifically to AND gates, since XOR gates are considered free [3]–[5]. Measurements for different papers are taken on computers when each paper was written.

A. *Garbled Circuits*

With (application layer) bandwidth of about 1.4MB/sec, garbled circuit protocol is presently among the fastest general-purpose secure computation techniques. It was first proposed in 1986 [29]. Numerous recent works improved the original protocol, such as free-XOR [3]–[5] and garbled row reduction [30], [31]. Oblivious Transfer (OT) [1], [6] is needed to bootstrap garbled circuit execution. Table I describes several existing secure computation prototypes using garbled circuits.

B. *Programming and Compiler Support*

Circuit generation. One key question is whether the circuits are *fully materialized* or *generated on the fly* during secure computation. Many first-generation secure computation compilers such as Fairplay [10], TASTY [11], Sharemind [7], CBMC-GC [14], PICCO [12], KSS12 [9], PCF [8] generate target code containing the fully materialized circuits. Since the circuit file size and compile time are proportional to the circuit size, they require siginificant compile time (e.g., 8.2 seconds for a circuit of size 700K in KSS12 [9]). In addition, the circuit must be recompiled for every input data size. Other secure computation compilers (e.g., Wysteria, and SCVM [8], [13], [15]) use *program-style target code*, which is a more compact intermediate representation of circuits. The program-style target code will be securely evaluated using a cryptographic protocol such as garbled circuit or GMW. Typically these protocols perform per-gate computation – therefore, circuits are generated on-the-fly at runtime. ObliVM also adopts program-style target code and on-the-fly circuit generation. Specifically, the circuit generation is pipelined [28] such that the it never needs to be materialized entirely.

ORAM support. Almost all existing secure computation compilers, including most recent ones such as Wysteria [15], PCF [8], and TinyGarble [32], compile dynamic memory accesses (whose addresses depend on secret inputs) to a linear scan of memory in the circuit representation. This is completely unscalable for computation over large secret data. SCVM leverages the idea of ORAM [33], [34] to make more efficient random accesses to secret data. SCVM employs the binary-tree ORAM [35] to implement dynamic memory accesses.

III. PROGRAMMING LANGUAGE AND COMPILER

We wish to design a powerful source language ObliVM-lang such that an expert programmer can *i)* develop oblivious programming abstractions as libraries ; and *ii)* implement low-level circuit gadgets atop ObliVM-lang.

ObliVM-lang builds on top of the recent SCVM source language [13]. In this section, we will describe new features that ObliVM-lang offers and explain intuitions behind our security type system. In Section IV, we give concrete case studies and show how to implement oblivious programming abstractions and low-level circuit libraries atop ObliVM-lang.

A. *Language features for expressiveness and efficiency*

Security labels. Except for the new random type introduced in Section III-B, all other variables and arrays are either of a `public` or `secure` type. `secure` variables are secret-shared between the two parties such that neither party sees the value. `public` variables are observable by both parties. Arrays can be publicly or secretly indexable. For example,

- `secure int10[public 1000] keys` defines an array whose contents is secret while the indices used to access the array will always be public. Thus, this array will be secret-shared but need not be placed in ORAMs.
- `secure int10[secure 1000] keys`: defines an array that will be indexed with secret value at least once, thus will be placed in a secret-shared ORAM.

Standard features. ObliVM-lang allows programmers to use C-style keyword `struct` to define *record types*. It also supports *generic types* similar to templates in C++. For example, a binary tree with public topological structure but secret per-node data can be defined without using pointers (assuming its capacity is 1000 nodes):

```
struct KeyValueTable<T> {
  secure int10[public 1000] keys;
  T[public 1000] values;
};
```

where int10 indicates the values are 10-bit signed integers. Each element in the array values has a generic type T similar to C++ templates. Note that ObliVM-lang currently requires data of type T is always secret-shared.

Generic constants. Besides general types, ObliVM-lang also supports *generic constants* to further improve the reusability. Let us consider the following tree example:

```
struct TreeNode@m<T> {
  public int@m key;
  T value;
  public int@m left, right;
};
struct Tree@m<T> {
  TreeNode<T>[public (1<<m)-1] nodes;
  public int@m root;
};
```

This code defines a binary search tree of key-value store nodes, where keys are m-bit integers. The *generic constant* @m is a variable whose value will be instantiated to a constant. "int@m left, right" indicates that m bits are enough to represent all the position references to the array. The type int@m refers to an integer type with m bits. Further, the capacity of array nodes can be determined by m as well (i.e. (1<<m)-1). Note that Zhang et al. [12] also allow specifying the length of an integer, but require this length to be a hard-coded constant – this necessitates modification and recompilation of the program for different inputs. ObliVM-lang's generic constant approach eliminates this constraint, and thus improves reusability.

Functions. ObliVM-lang allows programmers to define functions. For example, following the Tree defined as above, programmers can write a function to search the value associated with a given key in the tree as follows:

```
1   T Tree@m<T>.search(public int@m key) {
2     public int@m now = this.root, tk;
3     T ret;
4     while (now != -1) {
5       tk = this.nodes[now].key;
6       if (tk == key)
7         ret = this.nodes[now].value;
8       if (tk <= key)
9         now = this.nodes[now].right;
10      else
11        now = this.nodes[now].left;
12    }
13    return ret;
14  };
```

This function is a method of a Tree object, and takes a key as input, and returns a value of type T. The function body defines three local variables now and tk of type public int@m, and ret of type T. The definition of a local variable (e.g. now) can be accompanied with an optional initialization expression (e.g. this.root). When a variable (e.g. ret or tk) is not initialized explicitly, it is initialized to be a default value depending on its type.

The rest of the function is standard, C-like code, except that ObliVM-lang requires exactly one return statement at the bottom of a function whose return type is not void. Unlike previous loop-elimination-based work [7], [9]–[12], [14], ObliVM-lang allows arbitrary looping on a public guard (e.g. line 4) without loop unrolling.

Function types. Programmers can define a variable to have function type, similar to function pointers in C. However, our language is limited in that (*a*) the input and return types of functions cannot be function types; (*b*) generic types cannot be instantiated to function types.

Native primitives. ObliVM-lang supports native types and native functions. For example, ObliVM-lang's default back end implementation is ObliVM-GC, which is implemented in Java. Suppose an alternative BigInteger implementation in ObliVM-GC (e.g., using additively homomorphic encryption) is available in a Java class called BigInteger, programmers can define

```
typedef BigInt@m = native BigInteger;
```

Suppose this class supports four operations: add, multiply, fromInt and toInt, where the first two operations are arithmetic operations and last two operations are used to convert between Garbled Circuit-based integers and HE-based integers. We can expose these to the source language by declaring:

```
BigInt@m BigInt@m.add(BigInt@m x,
    BigInt@m y)= native BigInteger.add;
BigInt@m BigInt@m.multiply(BigInt@m x,
    = BigInt@m y) native BigInteger.multiply;
BigInt@m BigInt@m.fromInt(int@m y)
    = native BigInteger.fromInt;
int@m BigInt@m.toInt(BigInt@m y)
    = native BigInteger.toInt;
```

B. *Language features for security*

The key requirement of ObliVM-lang is that a program's execution traces will not leak information. These execution traces include a memory trace, an instruction trace, a function stack trace, and a declassification trace. The trace definitions are similar to Liu et al. [13]. We develop a security type system for ObliVM-lang. Liu et al. [13] has discussed how to prevent memory traces and instruction traces from leaking information. Here we explain the basic ideas of ObliVM-lang's type system concerning functions and declassifications.

Random numbers and implicit declassifications. Many oblivious programs such as ORAM and oblivious data structures crucially rely on randomness. In particular, the security of the programs requires that the joint distribution of memory traces is independent of the secret inputs (these algorithms typically have a cryptographically negligible probability of correctness failure). ObliVM-lang facilitates reasoning about this trace-obliviousness with a *random* type, which is governed by an affine type system. A random number will always be secret-shared between the two parties. We use rnd32 to denote the type of a 32-bit random integer.

We provide a built-in function RND that bears a signature:

```
rnd@m RND(public int32 m)
```

to generate random numbers. This function takes a public 32-bit integer `m` as input, and returns `m` random bits. Note that `rnd@m` is a *dependent type*, whose type depends on the value of `m`. ObliVM-lang limits the use of dependent types to only this `RND` function.

ObliVM provides special syntax to explicitly declassify outputs of a computation. However, it allows random numbers to be *implicit declassified* – by assigning them to public variables. By "implicitness", we mean that the declassification occurs without using ObliVM's explicit syntax of declassification.

For security reasons, we ensure that each random number is implicitly declassified *at most once*. Consider the following example where `s` is a secret variable.

```
 1   rnd32 r1 = RND(32), r2= RND(32);
 2   public int32 z;
 3   if (s) z = r1; // implicit declass
 4   else z = r2; // implicit declass
     ......
XX   public int32 y = r2; // NOT OK
```

random variables `r1` and `r2` are initialized in Line 1 – these variables are assigned a fresh, random value upon initialization. Up to Line 4, random variables `r1` and `r2` are each implicitly declassified. Line XX, however, could potentially cause `r2` to be declassified more than once. Line XX may not be secure because the observable public variable `y` and `z` could be correlated – depending on which secret branch was taken earlier.

Thus, we use an *affine type* system to ensure that each random variable is implicitly declassified at most once, so that each time a random variable is implicitly declassified, it only introduces an independent, uniform distributed variable to the observable trace. In the proof, a simulator can just sample a random number to produce an indistinguishable trace.

This trick is helpful in the implementation of oblivious RAM and oblivious data structures. We refer the readers to Sections IV and V-B for details.

Function calls and phantom functions. Function calls significantly complicate the analysis of information leakage. The basic idea of ObliVM-lang is to assume the native functions satisfy memory- and instruction-trace obliviousness. Beyond this basic idea, ObliVM-lang makes a step forward to enabling function calls within a secret if-statement by introducing the notion of *phantom function*. The idea is that each function can be executed in two modes, either a *real* mode or a *phantom* mode. In the real mode, all statements are executed normal with real computation and real side effects. In the phantom mode, the function execution merely simulates the memory traces of the real world; no side effects take place; and the phantom function call returns a secret-shared default value of the specified return type. This is similar to the padding trick used in [36], [37].

We illustrate phantom function with the `prefixSum` example below. The function `prefixSum(n)` accesses a global integer array `a`, and computes the prefix sum of the first $n + 1$ elements in `a`. After accessing each element (Line 3), the element in array `a` will be set to 0 (Line 4).

```
1  phantom secure int32 prefixSum
2    (secure int32[public int32] a, public int32 n) {
3    secure int32 ret=a[n];
4    a[n]=0;
5    if (n != 0) ret = ret+prefixSum(a, n-1);
6    return ret;
7  }
```

The keyword `phantom` indicates that the function `prefixSum` is a phantom function.

Consider the following code to call the phantom functions:

```
if (s) then x = prefixSum(a, n);
```

To ensure security, `prefixSum` will always be called no matter `s` is true or false. When `s` is false, however, `prefixSum` will be executed in a special way such that (1) elements in array `a` are not assigned to 0; and (2) the function results in traces distributed the same as when `s` is true. To this end, the compiler will generate target code with the following signature:

```
prefixSum(a, idx, indicator);
```

where `indicator` suggests whether the function will be called in the real or phantom mode. Since the global variable should be modified only if `indicator` is false. , the compiler will compile the code in line 4 into:

```
a[idx] = mux(0, a[idx], indicator);
```

thus leaving traces that are independent of `s` and `indicator`.

IV. Supporting Oblivious Programming Abstractions

We support oblivious programming abstractions that are potentially better understood by programmers, with a few specific language features.

A. *Programming Abstractions for Oblivious MapReduce*

A program efficiently expressed in parallel programming paradigms such as MapReduce and GraphLab [38], [39] (with a few additional constraints), can be easily compiled into its oblivious version. Note our focus here is to explain the language features, though the performance evaluation of this paper is completely restricted to using only single-cores.

Oblivious algorithms for streaming MapReduce. A *streaming* MapReduce program consists of two basic operations, `map` and `reduce`.

- The `map` operation takes an array $\{\alpha_i\}_{i\in[n]}$ where each $\alpha_i \in \mathcal{D}$ for some domain $\mathcal{D}$, and a function `mapper` : $\mathcal{D} \to \mathcal{K} \times \mathcal{V}$. `map` would apply `mapper`$(\alpha_{\mathtt{i}})$ to each α_i, and output an array of key-value pairs $\{(k_i, v_i)\}_{i\in[n]}$.
- The `reduce` operation takes in an initial value *init*, an array of key-value pairs denoted $\{(k_i, v_i)\}_{i\in[n]}$ and a function `reducer` : $\mathcal{K} \times \mathcal{V}^2 \to \mathcal{V}$. For every unique key k in this array, let $(k, v_{i_1}), (k, v_{i_2}), \dots (k, v_{i_m})$ denote all occurrences with the key k. `reduce` applies the following operation in a streaming fashion:

$$R_k := \mathtt{reducer}(k, \dots \mathtt{reducer}(k, \mathtt{reducer}(k, \mathit{init}, v_{i_1}), v_{i_2}), \dots, v_{i_m})$$

The result of `reduce` is an array of pairs (k, R_k), one pair for each unique k value in the input array.

```
1  Pair<K,V>[public n] MapReduce@m@n<I,K,V>
2      (I[public m] data,  Pair<K, V> map(I),
3       V reduce(K, V, V), V initialVal,
4       int2 cmp(K, K)) {
5    public int32 i;
6    Pair<K, V>[public m] d2;
7    for (i=0; i<m; i=i+1)
8      d2[i] = map(data[i]);
9    sort@m<K, V>(d2, 1, cmp);
10   K key = d2[0].k;
11   V val = initialVal;
12   Pair<int1, Pair<K, V>>[public m] res;
13   for (i=0; i+1<m; i=i+1) {
14     res[i].v.k = key;
15     res[i].v.v = val;
16     if (cmp(key, d2[i+1].k)==0) {
17       res[i].k.val = 1;
18     } else {
19       res[i].k.val = 0;
20       key = d2[i+1].k;
21       val = initialVal;
22     }
23     val = reduce(key, val, d2[i+1].v);
24   }
25   res[m-1].k.val = 0;
26   res[m-1].v.k = key;
27   res[m-1].v.v = val;
28   sort@m<int1, Pair<K, V>>
29     (res, 1, zeroOneCmp);
30   Pair<K, V>[public n] z;
31   for (i=0; i < n; i = i + 1)
32     z[i] = res[i].v;
33   return z;
34 }
```

Fig. 1: **Implementing MapReduce paradigm in ObliVM-lang.** `zeroOneCmp` is a built-in comparator for (k, v) pairs (based on k).

Goodrich and Mitzenmacher [40] observe that any program written in a streaming MapReduce abstraction can be efficiently executed obliviously. They leveraged this observation to construct an ORAM scheme.

- The `map` operation is inherently oblivious and can be done by a linear scan of the input array.
- The `reduce` operation can be done obliviously through an oblivious sorting (denoted o-sort) primitive.
  - First, o-sort the input array in ascending order by the key k.
  - Next, in a single linear scan, apply the `reducer` function: *i)* Output (k, R_k) whenever the last key-value pair for certain key k is encountered; and *ii)* a dummy entry $\perp$ for all other pairs.
  - Finally, o-sort all the resulting entries to move $\perp$ to the end.

The streaming MapReduce abstraction in ObliVM. It is not hard to implement the streaming MapReduce abstraction as a library with ObliVM-lang (Figure 1).

Our implementation of MapReduce introduces two generic constants `m` and `n`, representing the sizes of the input and output respectively. It also introduces three generic types, `I` for inputs' type, `K` for output keys' type, and `V` for output values' type. All the three types are assumed (restricted) to be secret. The MapReduce abstraction has five inputs: `data` denotes the input array, `map` denotes the mapper function, `reduce` denotes the reducer function, `initialVal` for the initial value of the reducer, and `cmp` denotes the key comparison function.

Lines 7-8 are the mapper phase of the algorithm, then line 9 uses the function `sort` to sort the intermediate results based on their keys (such that intermediate resulting pairs are grouped by their keys). Lines 10-27 compute the reduce phase, producing some dummy outputs. Finally, lines 28-29 eliminate these dummy items with another oblivious sort (using `zeroOneCmp` comparator). Finally, line 33 gives the output array. Note that all accesses to the arrays `data`, `d2`, `res`, and `z` do not depend on any secret value, thus can be efficiently processed without ORAM.

Using MapReduce. We illustrate how to use the MapReduce abstraction to implement an oblivious histogram. The purpose of histogram is to count the frequency of each value in a predefined range $[0..\mathtt{n}-1]$ in an array `a` of size `m`. it can be literally implemented by the following two loops:

```
for (public int i=0; i<n; ++i) c[i] = 0;
for (public int i=0; i<m; ++i) c[a[i]] ++;
```

Note because it makes dynamic memory accesses, a compiler (such as SCVM [13]) would put the array `c` inside an ORAM.

Nevertheless, an oblivious histogram computation can also be described using the MapReduce abstraction.

```
int2 cmp(int32 x, int32 y) {
  int2 r = 0;
  if (x < y) r = -1;
  else if (x > y) r = 1;
  return r;
}
Pair<int32, int32> mapper(int32 x) {
  return Pair<int32, int32>(x, 1);
}
int32 reducer(int32 k, int32 v1, int32 v2) {
  return v1 + v2;
}
```

Then the following code launches the histogram computation

```
c = MapReduce@m@n<int32, int32, int32>
        (a, mapper, reducer, cmp, 0);
```

In contrast, ObliVM-lang generates target code that relies on only oblivious sorting rather than a generic ORAM, improving the performance by a poly-logarithmic factor comparing to the SCVM implementation.

B. *Programming Abstractions for Oblivious Data Structures*

ObliVM provides programming abstractions for a class of pointer-based oblivious data structures proposed by Wang et al. [23]. The basic idea is that once an expert programmer provides library code for a class of pointer-based data structures, a non-specialist programmer can easily implement oblivious data structures such as oblivious stack, AVL tree, heap, and queue.

```
rnd@m RND(public int32 m) = native lib.rand;
struct Pointer@m {
  int32 index;
  rnd@m pos;
};
struct SecStore@m<T> {
  CircuitORAM@m<T> oram;
  int32 cnt;
};
phantom void SecStore@m<T>.add(int32 index,
  int@m pos, T data) {
  oram.add(index, pos, data);
}
phantom T SecStore@m<T>
  .readAndRemove(int32 index, rnd@m pos) {
  return oram.readAndRemove(index, pos);
}
phantom Pointer@m SecStore@m<T>.allocate() {
  cnt = cnt + 1;
  return Pointer@m(cnt, RND(m));
}
```

(a) Library code for supporting oblivious data structures.

```
struct StackNode@m<T> {
  Pointer@m next;
  T data;
};
struct Stack@m<T> {
  Pointer@m top;
  SecStore@m store;
};
phantom void Stack@m<T>.push(T data) {
  StackNode@m<T> node = StackNode@m<T> (
    top, data);
  this.top = this.store.allocate();
  store.add(top.(index, pos), node);
}
phantom T Stack@m<T>.pop() {
  StackNode@m<T> res = store
    .readAndRemove(top.(index, pos));
  top = res.next;
  return res.data;
}
```

(b) Oblivious stack code by non-expert programmers.

Fig. 2: **Programming abstractions for oblivious data structures.**

To support efficient data structure implementations, an expert programmer implements two essential structures (Figure 2a):

- `Pointer`, which keeps track of an `index` variable that stores the logical identifier of the memory block it points to, and a `pos` variable that stores the random leaf label of the memory block it points to.
- `SecStore`, which implements an ORAM, and provides the following member functions to an end-user: `SecStore.remove` (a syntactic sugar for the ORAM's `readAndRemove` interface [26], [35]), `SecStore.add` (a syntactic sugar for the ORAM's `Add` interface [26], [35]), and `SecStore.allocate` (which returns a new `Pointer` object with a fresh logical identifier and a fresh random leaf label `RND(m)`)

Note that the `rnd` type is governed by the affine type system, so that each `rnd` can be declassified (i.e., made public) at most once. Thus, `oram.readAndRemove` is guaranteed to declassify its argument `pos` only once.

Given the abstraction provided by the expert programmer, a non-expert programmer can write a class of data structures such as stack, queue, heap, AVL Tree, etc. Figure 2b gives an example for implementing oblivious stack.

C. Loop Coalescing and New Oblivious Graph Algorithms

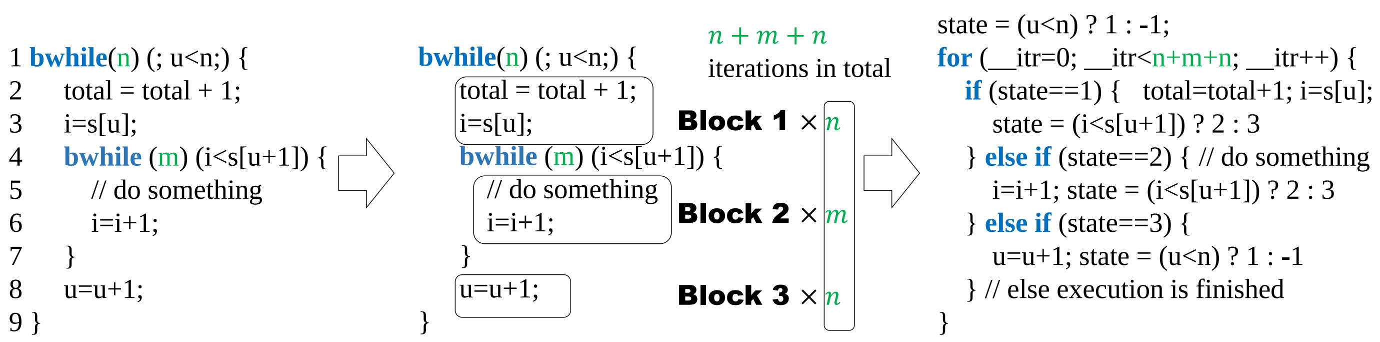
Many efficient graph algorithms involve nested loops while the total number of iterations has a smaller upper bound (than the product of the numbers of iterations of the inner and outer loops). One example of such algorithm is oblivious Dijkstra shortest path over sparse graphs (detailed explanation is given below). Note that intuitive translation of the program will result in an oblivious algorithm of complexity $O(mn)$ where m and n are the bounds of the two nested loops. In contrast, loop coalescing allows to preserve the overall efficiency of the nested loops to $O(v)$ where v is a bound smaller than mn. We note the same idea (termed "loop flattening") was used to parallelize irregular, nested loops [41] and remove data dependency when compiling RAM programs into efficiently verifiable circuits [42].

ObliVM supports loop coalescing by introducing a special syntax, called *bounded-for* loop (Figure 3). In Figure 3, the `bwhile(n)` and `bwhile(m)` syntax at Lines 1 and 4 indicate that the outer loop will be executed for a total of n times, whereas the inner loop will be executed for a total of m times, respectively.

To support loop coalescing, ObliVM partitions the code within the outer-loop into basic blocks. Then it transforms the outer bounded-loop into a normal loop with its body encoded by a state machine. Each state of the state machine corresponds to a basic block, while the control flow at the end of each basic block is carried out by state assignments. It is easy to verify that the total number of iterations is the sum of iterations for every basic block.

Oblivious Dijkstra shortest path for sparse graphs. While Blanton et al. [43] considered oblivious shortest path problem on *dense* graphs, our focus is more efficient oblivious algorithms over *sparse* graphs. Our starting point is the priority-queue-based Dijkstra's algorithm, whose basic idea is to update weights whenever a shorter path to a vertex is found. However, an naive translation of this idea to its oblivious version makes the update operation the dominant cost, as it would use a generic ORAM. Our solution to this problem is more efficient thanks to avoiding weight updates and adopting loop coalescing.

Avoiding weight updates. This is accomplished by two changes to a standard priority-queue-based Dijkstra's algorithm, i.e.,



```
bwhile(n) (; u<n;) {
    total = total + 1;
    i=s[u];
    bwhile (m) (i<s[u+1]) {
        // do something
        i=i+1;
    }
    u=u+1;
}
```

```
bwhile(n) (; u<n;) {
    total = total + 1;
    i=s[u];
    bwhile (m) (i<s[u+1]) {
        // do something
        i=i+1;
    }
    u=u+1;
}
```

```
state = (u<n) ? 1 : -1;
for (__itr=0; __itr<n+m+n; __itr++) {
    if (state==1) {   total=total+1; i=s[u];
        state = (i<s[u+1]) ? 2 : 3
    } else if (state==2) { // do something
        i=i+1; state = (i<s[u+1]) ? 2 : 3
    } else if (state==3) {
        u=u+1; state = (u<n) ? 1 : -1
    } // else execution is finished
}
```

Fig. 3: **Loop coalescing.** The outer loop will be executed at most n times, the inner loop will be executed at most m times. A naive approach compiler would pad the outer and inner loop to n and m respectively, incurring $O(nm)$ cost. With loop coalescing, overall cost is reduced to $O(n + m)$.

Algorithms		**Complexity**		
		Our Complexity	Generic ORAM	Best Known
Sparse Graph	Dijkstra's Algorithm	$O((E+V)\log^2 V)$	$O((E+V)\log^3 V)$	$O((E+V)\log^3 V)$ (Generic ORAM baseline [26])
	Prim's Algorithm	$O((E+V)\log^2 V)$	$O((E+V)\log^3 V)$	$O(E\frac{\log^3 V}{\log\log V})$ for $E = O(V\log^\gamma V), \gamma \geq 0$ [19] $O(E\frac{\log^3 V}{\log^\delta V})$ for $E = O(V2^{\log^\delta V}), \delta \in (0,1)$ [19] $O(E\log^2 V)$ for $E = \Omega(V^{1+\epsilon}), \epsilon \in (0,1]$ [19]
Dense Graph	Depth First Search	$O(V^2 \log V)$	$O(V^2 \log^2 V)$	$O(V^2 \log^2 V)$ [43]

TABLE II: **Summary of algorithmic improvement.** All costs are calculated in terms of circuit size (but ignoring the impact of bit-length in each word). The improvement of oblivious Dijkstra's algorithm and Prim's algorithm is a result of loop coalescing and oblivious data structures. The oblivious DFS algorithm uses additional technical techniques, which are detailed in Appendix A.

Algorithm 1 **Dijkstra's algorithm using bounded loops**

Secret Input: `src`: the source node
Secret Input: `e`: concatenation of adjacency lists stored in an ORAM.
Secret Input: `s`[u]: sum of out-degree on vertices from 1 to u.
Output: `dis`: the shortest distances from `src` to other nodes

```
1:  dis := [$\infty, \infty, ..., \infty$]
2:  PQ.push(0, src)
3:  dis[$s$] := 0
4:  bwhile(V)(!PQ.empty())
5:      (d, u) := PQ.deleteMin()
6:      if(dis[$u$] == d) then
7:          dis[u] := $-$dis[u];
8:          bfor(E)(i := s[u]; i < s[u + 1]; i = i + 1)
9:              (u, v, w) := e[i];
10:             newDist := d + w
11:             if (newDist < dis[v]) then
12:                 dis[v] := newDist
13:                 PQ.insert(newDist, u)
```

lines 6-7 and line 12 of Algorithm 1. The key observation is that, whenever a shorter distance `newDist` from s to u is found, instead of updating the existing weight of u in the heap, we insert a new pair $(\texttt{newDis}, u)$ into the priority queue. This change could result in multiple entries for the same vertex being inserted in the queue, henceforth raising two concerns: (1) the same vertex might be popped out of the queue and processed multiple times; and (2) the cost of operating the priority queue may increase asymptotically as the size of the queue grows. The first concern is unnecessary due to the check and negation in line 6-7: every vertex will be processed at

Algorithm 2 **Oblivious Dijkstra' algorithm**

Secret Input: `e`, `s`: same as Algorithm 1
Output: `dis`: the shortest distance from s to each node

```
1:  dis := [$\infty, \infty, ..., \infty$];  dis[$source$] = 0
2:  PQ.push(0, s);  innerLoop := false
3:  for $i := 0 \to 2V + E$ do
4:      if not innerLoop then
5:          (dist, u) := PQ.deleteMin()
6:          if dis[u] == dist then
7:              dis[u] := $-$dis[u];  i := s[u]
8:              innerloop := true;
9:      else
10:         if i < s[u + 1] then
11:             (u, v, w):= e[i]
12:             newDist := dist + $w$
13:             if newDist < dis[u] then
14:                 dis[u] := newDist
15:                 PQ.insert(newDist, v')
16:             i = i + 1
17:         else
18:             innerloop := false;
```

most once because `dis`[u] will be set negative once vertex v is processed. Regarding the second concern, we note the size of the queue is actually bounded by $E = O(V^2)$ (as $E = o(V^2)$ for sparse graphs). Therefore, the cost of each `insert` and `deleteMin` operation over the oblivious priority queue remains to be $O(\log^2 V)$ [23].

Loop coalescing. As V is considered a secret value, a naive approach to execute the nested loop would pad each loop to its maximum possible iterations (i.e., $V + E$). In contrast, using

```
1  int@(2*n) karatsubaMult@n(
     int@n x, int@n y) {
2    int@2*n ret;
3    if (n < 18) {
4      ret = x*y;
5    } else {
6      int@(n-n/2) a = x$n/2~n$;
7      int@(n/2) b = x$0~n/2$;
8      int@(n-n/2) c = y$n/2~n$;
9      int@(n/2) d = y$0~n/2$;
10     int@(2*(n-n/2)) t1 =
         karatsubaMult@(n-n/2)(a, c);
11     int@(2*(n/2)) t2 =
         karatsubaMult@(n/2)(b, d);
12     int@(n-n/2+1) aPb = a + b;
13     int@(n-n/2+1) cPd = c + d;
14     int@(2*(n-n/2+1)) t3 =
         karatsubaMult@(n-n/2+1)(aPb, cPd);
15     int@(2*n) padt1 = t1;
16     int@(2*n) padt2 = t2;
17     int@(2*n) padt3 = t3;
18     ret = (padt1<<(n/2*2)) + padt2 +
       ((padt3 - padt1 - padt2)<<(n/2));
19   }
20   return ret;
21 }
```

Fig. 4: **Karatsuba multiplication in ObliVM-lang.** `x$i~j$` extracts the `i`-th to the `j`-th bits of `x`.

loop coalescing technique, because at most V vertices and E edges will be visited, each iteration of the single coalesced loop will handle either a vertex (lines 5-8) or an edge (lines 11-16). Note ObliVM-lang coimpiler will pad the `if`-branches in Algorithm 2 to ensure trace obliviousness.

Since each iteration of the loop (lines 3-18) makes a constant number of ORAM accesses and two priority queue primitives operations (each cost $O(\log^2 V)$), the total cost is $O((V+E)\log^2 V)$.

Additional algorithmic results. We also develop a new oblivious Minimum Spanning Tree (MST) algorithm, and an oblivious Depth First Search (DFS) algorithm for dense graphs that is asymptotically faster than a baseline using generic ORAM (Table II). The detailed description of these algorithms are in Appendix A.

V. Case Studies

We present two case studies of ObliVM-lang programming.

A. Basic Arithmetic Operations

Figure 4 shows the implementation of oblivious Karatsuba multiplication [44] in ObliVM-lang. Karatsuba proposed the following recursive algorithm to compute multiplication of two n bit numbers. First, express the n-bit integers `x` and `y` as the concatenation of $n/2$-bit integers: `x = a*2`$^{n/2}$`+b, y = c*2`$^{n/2}$`+d`. Then `x*y` can be calculated as follows:

```
t1 = a*c; t2 = b*d; t3 = (a+b)*(c+d);
x*y = t1<<n + t2 + (t3-t1-t2)<<(n/2);
```

where the multiplications `a*c` and `b*d` are implemented through a recursive call to the Karatsuba algorithm itself.

```
1  #define BUCSIZE 3
2  #define STASHSIZE 33
3  struct Block@n<T>{
4    int@n id, pos;
5    T  data;
6  };
7  struct CircuitOram@n<T>{
8    dummy Block@n<T>[public 1<<n+1]
       [public BUCSIZE] buckets;
9    dummy Block@n<T>[public STASHSIZE] stash;
10 };
11 phantom T  CircuitOram@n<T>
     .readAndRemove(int@n id, rnd@n pos) {
12   public int32 pubPos = pos;
13   public int32 i = (1 << n) + pubPos;
14   T  res;
15   for (public int32 k = n; k>=0; k=k-1) {
16     for (public int32 j=0;j<BUCSIZE;j=j+1)
17       if (buckets[i][j] != null &&
18             buckets[i][j].id == id){
19           res = buckets[i][j].data;
20           buckets[i][j] = null;
21       }
22     i=(i-1)/2;
23   }
24   for (public int32 i=0;i<STASHSIZE;i=i+1)
25     if (stash[i]!=null&&stash[i].id==id) {
26         res = stash[i].data;
27         stash[i] = null;
28     }
29   return res;
30 }
```

Fig. 5: **Part of Circuit ORAM code in ObliVM-lang.**

We introduce a syntactic sugar in ObliVM-lang to extract subset of bits in an integer. For example, in lines 6-9 of Figure 4, `num$i~j$` denotes the i-th to j-th bits of `num`.

B. Circuit ORAM

Figure 5 shows part of the Circuit ORAM implementation using ObliVM-lang. Line 3-6 defines an ORAM block containing two metadata fields, an index `id` and a position label `pos`, along with a data field of type `<T>`. Circuit ORAM (line 7-10) is organized to contain an array of buckets and a stash. The `dummy` keyword in front of `Block@n<T>` indicates the value of this type can be `null`. Line 11-30 demonstrates how `readAndRemove` can be implemented.

VI. Evaluation

ObliVM incorporates a standard garbling scheme with Garbled Row Reduction [30], free-XOR [3], and Half-Gates [45]. It uses an OT extension protocol proposed by Ishai et al. [1] and a basic OT protocol by Naor and Pinkas [46].

A. Metrics and Experiment Setup

Number of AND gates. In Garbled Circuit-based secure computation, functions are represented in boolean circuits consisting of XOR and AND gates. Thanks to the free-XOR technique [3]–[5], the primary performance metric becomes the number of AND gates. This metric is platform independent, i.e., independent of the artifacts of the underlying software implementation, or the hardware configurations where the benchmark numbers are measured. This metric facilitates a fair comparison with existing works based on boolean circuits.

Wall-clock runtime. Unless noted otherwise, all wall-clock numbers are measured by executing the protocols between two Amazon EC2 machines of types c4.8xlarge and c3.8xlarge. This metric is platform and implementation dependent, and therefore we will explain how to best interpret wallclock runtimes.

Compilation time. For all programs we ran, the compilation time is under 1 second. Therefore, we do not separately report the compilation time.

B. ObliVM vs. Hand-Crafted Solutions

Development effort. We give two concrete case studies to demonstrate the reduction in developer effort using ObliVM-lang: (1) *Ridge regression.* Ridge regression is an important building block in various machine-learning tasks [47]. Previously, Nikolaenko et al. [47]'s implementation, took roughly three weeks development time, while it takes two student·hours to accomplish exactly the same task using ObliVM. In addition to the speedup gain from ObliVM-GC, our optimized libraries result in $3\times$ smaller circuits. (2) *Oblivious data structures.* In an earlier work [23], we designed an oblivious AVL tree algorithm, but were unable to implement it due to its complexity. This work enables us to implement an AVL tree with 311 lines of code in ObliVM-lang, taking 10 student·hours (including debugging time).

Competitive performance. We compared implementations generated by ObliVM to those hand-crafted without using ObliVM-lang in a number of applications, including Heap, Map/Set, AMS Sketch, Count-Min Sketch, and K-Means, Here the human experts are authors of this paper, who employs best known algorithms to accomplish the computational tasks. For example, Histogram and K-Means algorithms are implemented with oblivious sorting protocols instead of generic ORAM. Among the selected applications, ObliVM programs are competitive to hand-crafted implementations – and the performance difference is less than 5%.

C. End-to-End Application Performance

Table IV summarizes three types of applications, basic instructions (e.g., addition, multiplication, and floating point operations); linear and super-linear algorithms (e.g., Dijkstra, K-Means, Minimum Spanning Tree, and Histogram); and sublinear algorithms (e.g., Heap, Map/Set, Binary Search, Count Min Sketch, AMS Sketch). For cases where inputs are a large dataset (e.g., Heap, Map/Set, etc), depending on the application, the client may sometimes need to place the inputs in an ORAM, and secret-share the resulting ORAM. We do not measure this setup cost in the evaluation.

Program	Input size	ObliVM	
		#AND gates	Total time
Basic instructions			
Integer addition	1024 bits	**1024**	**1.7ms**
Integer mult.	1024 bits	**572K**	**833ms**
Integer Comparison	16384 bits	**16384**	**26ms**
Floating point addition	64 bits	**3035**	**4.32ms**
Floating point mult.	64 bits	**4312**	**6.29ms**
Hamming distance	1600 bits	**3200**	**5.07ms**
Linear and super-linear algorithms			
K-Means	0.5MB	**2269M**	**62.1min**
Dijkstra's Algorithm	48KB	**10B**	
MST	48KB	**9.6B**	**12.4h**
Histogram	0.25MB	**866M**	**21.5min**
Sublinear algorithms			
Heap	1GB	**12.5M**	**59.3s**
Map/Set	1GB	**23.9M**	**117.2s**
Binary Search	1GB	**1562K**	**7.36s**
Count Min Sketch	0.31GB	**8088K**	**20.77s**
AMS Sketch	1.25GB	**9949K**	**36.76s**

TABLE IV: **Application performance.** Numbers for basic instructions and sublinear algorithms are means of 20 runs.

Acknowledgments

We are indebted to Michael Hicks and Jonathan Katz for their continual support of the project. We are especially thankful towards Andrew Myers for his thoughtful feedback during the revision of the paper. We also gratefully acknowledge Srini Devadas, Christopher Fletcher, Ling Ren, Albert Kwon, abhi shelat, Dov Gordon, Nina Taft, Udi Weinsberg, Stratis Ioannidis, and Kevin Sekniqi for their insightful inputs and various forms of support. We thank the anonymous reviewers for their insightful feedback. This research is partially supported by NSF grants CNS-1464113, CNS-1314857, a Sloan Fellowship, Google Research Awards, and a subcontract from the DARPA PROCEED program.

References

[1] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending Oblivious Transfers Efficiently," in *CRYPTO 2003*, 2003.

[2] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient Garbling from a Fixed-Key Blockcipher," in *S & P*, 2013.

[3] V. Kolesnikov and T. Schneider, "Improved Garbled Circuit: Free XOR Gates and Applications," in *ICALP*, 2008.

[4] S. G. Choi, J. Katz, R. Kumaresan, and H.-S. Zhou, "On the security of the "free-xor" technique," in *TCC*, 2012.

[5] B. Applebaum, "Garbling xor gates "for free" in the standard model," in *TCC*, 2013.

[6] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, "More Efficient Oblivious Transfer and Extensions for Faster Secure Computation," ser. CCS '13, 2013.

[7] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A Framework for Fast Privacy-Preserving Computations," in *ESORICS*, 2008.

[8] B. Kreuter, B. Mood, A. Shelat, and K. Butler, "PCF: A portable circuit format for scalable two-party secure computation," in *Usenix Security*, 2013.

[9] B. Kreuter, a. shelat, and C.-H. Shen, "Billion-gate secure computation with malicious adversaries," in *USENIX Security*, 2012.

[10] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay: a secure two-party computation system," in *USENIX Security*, 2004.

[11] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "Tasty: tool for automating secure two-party computations," in *CCS*, 2010.

	Oblivious programming abstractions and compiler optimizations demonstrated	**Parameters**
Dijkstra's Algorithm MST	Loop coalescing abstraction (see Section IV-C).	$V = 2^{10}, E = 3V$
Heap Map/Set Binary Search	Oblivious data structure abstraction (see Section IV-B).	$N = 2^{23}, K = 32, V = 992$
AMS Sketch	Compile-time optimizations: split data into separate ORAMs [13].	$\epsilon = 2.4 \times 10^{-4}, \delta = 2^{-20}$
Count Min Sketch		$\epsilon = 3 \times 10^{-6}, \delta = 2^{-20}$
K-Means	MapReduce abstraction (see Section IV-A).	$N = 2^{16}$

TABLE III: **List of applications used in Table IV.** For graph algorithms, V, E stand for number of vertices and edges; for data structures, N, K, V stand for capacity, bit-length of key and bit-length of value; for streaming algorithms, ϵ, δ stand for relative error and failure probability; for K-Means, N stands for number of points.

[12] Y. Zhang, A. Steele, and M. Blanton, "PICCO: a general-purpose compiler for private distributed computation," in *CCS*, 2013.

[13] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks, "Automating Efficient RAM-model Secure Computation," in *S & P*, May 2014.

[14] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure Two-party Computations in ANSI C," in *CCS*, 2012.

[15] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations," in *S & P*, 2014.

[16] "Partisia," http://www.partisia.dk/.

[17] "Dyadic security," http://www.dyadicsec.com/.

[18] R. Canetti, "Security and composition of multiparty cryptographic protocols," *Journal of Cryptology*, 2000.

[19] M. T. Goodrich and J. A. Simons, "Data-Oblivious Graph Algorithms in Outsourced External Memory," *CoRR*, vol. abs/1409.0597, 2014.

[20] J. Brickell and V. Shmatikov, "Privacy-preserving graph algorithms in the semi-honest model," in *ASIACRYPT*, 2005.

[21] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, "GraphSC: Parallel Secure Computation Made Easy," in *IEEE S & P*, 2015.

[22] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *CCS*, 2013.

[23] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious Data Structures," in *CCS*, 2014.

[24] M. Keller and P. Scholl, "Efficient, oblivious data structures for MPC," in *Asiacrypt*, 2014.

[25] J. C. Mitchell and J. Zimmerman, "Data-Oblivious Data Structures," in *STACS*, 2014, pp. 554–565.

[26] X. S. Wang, T.-H. H. Chan, and E. Shi, "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound," Cryptology ePrint Archive, Report 2014/672, 2014.

[27] "Rsa distributed credential protection," http://www.emc.com/security/rsa-distributed-credential-protection.htm.

[28] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits," in *Usenix Security Symposium*, 2011.

[29] A. C.-C. Yao, "How to generate and exchange secrets," in *FOCS*, 1986.

[30] M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design," ser. EC '99, 1999.

[31] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole: Reducing data transfer in garbled circuits using half gates," in *EUROCRYPT*, 2015.

[32] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits," in *IEEE S & P*, 2015.

[33] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, 1996.

[34] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *STOC*, 1987.

[35] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ worst-case cost," in *ASIACRYPT*, 2011.

[36] J. Agat, "Transforming out timing leaks," in *POPL*, 2000.

[37] A. Russo, J. Hughes, D. A. Naumann, and A. Sabelfeld, "Closing internal timing channels by transformation," in *ASIAN*, 2006.

[38] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010.

[39] "Graphlab," http://graphlab.org.

[40] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious RAM simulation," in *ICALP*, 2011.

[41] A. M. Ghuloum and A. L. Fisher, "Flattening and parallelizing irregular, recurrent loop nests," *PPoPP*, pp. 58–67, Aug. 1995.

[42] S. S. R. Z. B. A. Wahby, R.S. and M. Walfish, "Efficient ram and control flow in verifiable outsourced computation," in *Network and Distributed System Security Symposium (NDSS)*, 2015.

[43] M. Blanton, A. Steele, and M. Alisagari, "Data-oblivious graph algorithms for secure computation and outsourcing," in *ASIA CCS*, 2013.

[44] A. A. Karatsuba, "The Complexity of Computations," 1995.

[45] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole reducing data transfer in garbled circuits using half gates," in *EUROCRYPT*, 2015.

[46] M. Naor and B. Pinkas, "Efficient oblivious transfer protocols," in *SODA*, 2001.

[47] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-preserving ridge regression on hundreds of millions of records," in *S & P*, 2013.

[48] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in)security of hash-based oblivious RAM and a new balancing scheme," in *SODA*, 2012.

Appendix A
Additional Oblivious algorithms

A. *Oblivious Depth-First Search*

We consider oblivious depth first search over dense graphs and present a more efficient protocol than using a generic ORAM. Our protocol runs in $O((E+V)\log V)$ time whereas a generic ORAM based solution will take $O((E+V)\log^2 V)$ time (ignoring possible $\log\log$ factors) [26], [48].

The challenge is to verify whether a vertex has been visited every time we explore a new edge. Typically, this is done by storing a bit-array that supports dynamic access, whose naive implementation using ORAM incurs $O(\log^2 V)$ cost per access (hence $O(E\log^2 V)$ time over $O(E)$ accesses).

To reduce cost, instead of recording if a vertex has been visited, we maintain a `tovisit` list of vertexes, which preserves the same traversal order as DFS. When adding new

Algorithm 3 Oblivious DFS

Secret Input: s: starting vertex;
Secret Input: E: adjacency matrix, stored in an ORAM of V blocks, *each block being one row of the matrix.*
Output: `order`: DFS traversal order *// not in ORAM*

```
1:  tovisit:=[$(s, 0)$, $\perp$, ..., $\perp$];  // not in ORAM
2:  for $i = 1 \to |V|$ do
3:      $(u, \texttt{depth})$ := tovisit[1];
4:      tovisit[1] := $(u, \infty)$; // mark as visited
5:      order[$i$] := $u$;
6:      edge := $E[u]$;
7:      for $v := 1 \to |V|$ do
8:          if edge[$v$] == 1 then // $(u, v)$ is an edge
9:              add[$v$] := $(v, i)$; // add is not in ORAM
10:         else  // $(u, v)$ is not an edge
11:             add[$v$] := $\perp$;
12:     $\boxed{\texttt{tovisit.Merge(add)}}$;
13: return order
```

vertexes to `tovisit`, we ensure each vertex appears in the list at most once by oblivious sorting. Algorithm 3 presents our oblivious DFS algorithm.

Since DFS explores the latest visited vertex first, so we maintain a stack-like `tovisit` array, where the top of the stack is stored in position 1. Each cell of `tovisit` is a pair `(`u`, depth)`:

- `(`u`,` ∞`)` indicates that vertex u has been visited.
- `(`u`, depth)` with a finite `depth` indicates that vertex u was reached at depth `depth`. The bigger the `depth`, the sooner u should be expanded.

Each iteration of the main loop (Lines 2-12) reads the top of the stack-like `tovisit` array, and expands the vertex encountered. The most interesting part of the algorithm is Line 12, highlighted in red. In this step, the newly reached vertices in this iteration, stored in the `add` array, will be added to the `tovisit` array in a non-trivial manner as explained below. At the end of each iteration (i.e., after executing Line 12), the following invariants hold for the array `tovisit`:

- *Sorted by depth.* All entries in `tovisit` are sorted by their `depth` in decreasing order. This ensures an entry added last (with largest `depth`) will be "popped" first.
- *No duplicates.* Any two entries (v, d) and (v, d') where $(d > d')$ will be combined into (v, d).
- *Fixed length.* The length of `tovisit` is exactly V.
- *Visited vertexes will never be expanded.* All entries with ∞ depth come after those with a finite depth.

The merge operation (Line 12). The operation is performed with two oblivious sorts. See Figure 6 for an illustrated example.

1) *O-sort and deduplicate.* This sorting groups all entries for the same vertex together, with the `depth` field in descending order (∞ comes first). All $\perp$ entries are moved to the end. Then, for all entries with the same vertex number (which are adjacent), we keep only the first one while overwriting others with $\perp$.
2) *O-sort and trim.* This sorting will (*a*) push all $\perp$ entries to the end; (*b*) push all ∞ entries to the end; and (*c*) sort all remaining entries in descending order of `depth`. Discard everything but the first V entries.

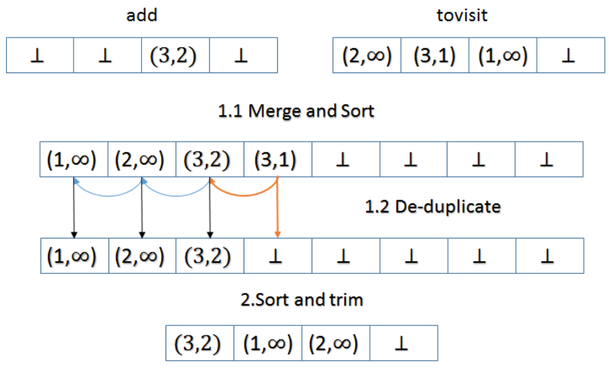


Fig. 6: Oblivious DFS Example: illustration of `tovisit.Merge(add)`.

Cost analysis. The inner loop (lines 8-11) runs in constant time, and will run V^2 times. Lines 3-5 also run in constant time, but will only run V times. Line 6 is an ORAM read, and it will run V times. Since the ORAM's block size is $V = \omega(\log^2 V)$, each ORAM read has an amortized cost of $O(V \log V)$. Finally, Line 12, which will run V times, consists of four oblivious sortings over an $O(V)$-size array, thus costs $O(V \log V)$. Hence, the overall cost of our algorithm is $O(V^2 \log V)$.

B. *Oblivious Minimum Spanning Tree*

In Algorithm 4, we show the pseudo-code for minimum spanning tree algorithm written using ObliVM-lang with the loop coalescing abstraction. The algorithm is very similar to the standard textbook implementation except for the annotations used for bounded-for loops in Lines 4 and 9. As described in Section IV-C, the inner loop (Line 9 to Line 11) will only execute $O(V+E)$ times in total. Further, each execution of the inner loop requires circuits of size $O(\log^2 V)$, using operations on oblivious priority queue [23] and Circuit ORAM [26]. So the overall complexity is $O((V+E)\log^2 V)$.

Algorithm 4 Minimum Spanning Tree with bounded for

Secret Input: `s`: the source node
Secret Input: `e`: concatenation of adjacency lists stored in a single ORAM array. Each vertex's neighbors are stored adjacent to each other.
Secret Input: `s[u]`: sum of out-degree over vertices from 1 to u.
Output: `dis`: the shortest distance from `source` to each node

```
1:  explored := [false, false, ..., false]
2:  PQ.push(0, s)
3:  res := 0
4:  bwhile(V)(!PQ.empty())
5:      (weight, u) := PQ.deleteMin()
6:      if(!explored[u]) then
7:          res:= res + weight
8:          explored[u] := true
9:          bfor(E)(i := s[u]; i < s[u + 1]; i = i + 1)
10:             (u, v, w) = e[i];
11:             PQ.insert(w, v)
```