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 are in the process of open-sourcing ObliVM and our rich libraries to the community (www.oblivm.com), offering a reusable framework to implement and distribute new cryptographic algorithms.

I. INTRODUCTION

Secure computation [1, 2] 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 [3–8]. As a result, secure computation has evolved from being just a nice theoretical concept to having real system prototypes [9–17]. Several attempts to commercialize secure computation techniques have also been made [18, 19].

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 for automated 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 is intended to address a fundamental representation gap, namely, secure computation protocols (and other branches of modern cryptography) rely on circuits as an abstraction of computation, whereas real-life developers write programs instead. 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. To demonstrate an end-to-end system, we chose to implement an improved Garbled Circuit protocol as the back end, since it is among the most practical protocols to date. Our ObliVM framework, including source code and demo applications, will be open-sourced on our project website http://www.oblixvm.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. For the time being, imagine that the cryptographic protocol used to execute each instruction securely realizes an ideal functionality without leaking any unintended information. Therefore, the parties can only observe the 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 secret (similar to the notion of a low or declassified variable in standard information flow terminology). Imprecisely speaking, for security, 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 [20], and is similar to the notion adopted in the SCVM work [15].

Relationship between oblivious programs and circuits. If a program is trace-oblivious by the aforementioned informal definition, it is then easy to convert the program into a sequence of circuits. These circuits are allowed to take memory accesses as inputs, however, these memory access must be oblivious to preserve security. By contrast, if a program is not memory-trace oblivious, then a dynamic memory access (whose address depends on secret inputs) cannot be efficiently made in the circuit representation—a straightforward approach (which is implicitly taken by almost all previous works except SCVM [13]) is to translate each dynamic memory access into a linear scan of memory in the resulting circuit, incurring prohibitive costs for large data sizes.

Moreover, instruction-trace obliviousness is effectively
guaranteed by executing both branches of a secret conditional in the resulting circuit where only one branch’s execution takes effect. Our type system (formally defined in a separate manuscript [21]) rejects programs that loop on secret variables – in these cases, a maximum public bound on the loop guard can be supplied instead.

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 ORAM 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 [15], 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 [22], [23], machine learning algorithms [24], [25], and data structures [26]-[28]. 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-specialist programmers without security expertise. 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 loop coalescing abstraction which enables novel oblivious graph algorithms. We remark that this is by no means 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-specialist application developers find the language intuitive.
- Expert programmers should be able to extend our framework with new features. For example, an expert programmer should be able to introduce new, user-facing oblivious programming abstractions by embedding them as libraries in ObliVM-lang (see Section V-B for an example).
- 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. In most previous frameworks, circuit libraries for these basic operations are developed in the back end. ObliVM, for the first time, 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 than any existing language for secure computation [10], [14]-[17]. Our ObliVM-lang extends the SCVM language by Liu et al. [15] 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. Section VI describes an example where an expert programmer designs a customized protocol for BigInteger operations using additively-homomorphic encryption. The resulting BigInteger types and operations can then be exported into our source language ObliVM-lang.

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 [29] is adopted for the baseline) under moderately large data sizes. We also compare our ObliVM-generated circuits with hand-crafted designs, and show that for a variety of applications, our auto-generated circuits are only 0.5% to 2% bigger in size than oblivious algorithms hand-crafted by human experts.
**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 a few new oblivious algorithms that we uncover 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 [30].

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 easily hidden simply 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.

**Scope.** A subset of ObliVM’s source language ObliVM-lang has a security type system which, roughly speaking, ensures that the program's execution traces are independent of secret inputs [15], [31]. However, a formal treatment of the language and the type system is outside the scope of this paper and deferred to a forthcoming manuscript [21].

By designing a new language, ObliVM does not directly retrofit legacy code. Such a design choice maximizes opportunities for compile-time optimizations. We note, however, that in subsequent work joint with our collaborators [32], we have implemented a MIPS CPU in ObliVM, which can securely evaluate standard MIPS instructions in a way that leaks only the termination channel (i.e., total runtime of the program) – this secure MIPS CPU essentially provides backward compatibility atop ObliVM whenever needed.

II. RELATED WORK

Existing general-purpose secure computation systems can be classified roughly based on two mostly orthogonal dimensions: 1) which “back end” secure computation protocol they adopt – this will also decide whether the system is secure against semi-honest or malicious adversaries, and whether the system supports two or multiple parties; and 2) whether they offer programming and compiler support – and if so, which language and compiler they adopt.

A. Back End: Secure Computation Implementations

Below we discuss choices of back end secure computation protocols and implementations. As discussed later, under realistic bandwidth provisioning about 1.4MB/sec, Garbled Circuit is presently among the fastest general-purpose protocol for secure computation. Currently, ObliVM primarily supports a semi-honest Garbled Circuit based back end, but developers can introduce customized gadgets for special-purpose types and functions (e.g., operations on sets, matrices, and big integers), and export them as native types and functions in the source language. It would not be too hard to extend ObliVM to support additional back end protocols such as GMW and FHE – in particular, almost all known protocols use a circuit abstraction (either boolean or arithmetic circuits). An interesting direction of the future research is to create new, compile-time optimizations that automatically selects the optimal mix of protocols for a given program, similar to what TASTY [15] proposed, but in a much broader sense.

**Garbled Circuit (GC) implementations.** The Garbled Circuit protocol was first proposed by Andrew Yao [35]. Numerous later works improved the original protocol: Free XOR shows that XOR gates can be computed almost “freely” [5]–[7]. Row reduction techniques show that only 2 or 3 garbled entries (rather than 4) need to be sent across the network per AND gate [36], [37]. A building block called Oblivious Transfer (OT) that is necessary for Garbled Circuit protocols was proposed and improved in a sequence of works as well [3], [8].

Several works have implemented the Garbled Circuit protocol – we give an overview of their features and performance characteristics in Table I.

**Non-GC protocols and implementations.** Besides Garbled Circuits, several other techniques have been proposed for general-purpose secure computation, including FHE [38], GMW [2], schemes based on linear secret-sharing [9], [14], etc. More discussions on non-GC protocols and implementations can be found in our online technical report [39].

B. Programming and Compiler Support

Secure computation compilers are in charge of compiling programs to circuit representations. One subtlety must be clarified: instead of a single circuit, here a program may be compiled to a sequence of circuits whose inputs are oblivious memory accesses. The number of these circuits will determine the number of interactions of the protocol.

**Circuit generation.** One key question is whether the circuits are fully materialized or generated on the fly during secure
Thus they show how to borrow hardware circuit synthesis and can have a somewhat more global view of the circuit. Songhori et al. show that by partially materializing a circuit, total runtime is diminished. Further, we stress that on-the-fly circuit generation is pipelined using a well-known technique by Huang et al. [33] such that the circuit is never materialized entirely, and thus only a constant amount of working memory is needed. Program-style target code is essentially adopted a much better architected and engineered version of FastGC [33]. The focus of this paper is our language, programming abstractions, and compiler. It is our future work to extend JustGarble (C-based, hardware AES-NI) to a fully working back end and integrate it with our language and compiler.

<table>
<thead>
<tr>
<th>GC Back End</th>
<th>Features</th>
<th>Garbling Speed</th>
<th>Bandwidth to match compute</th>
<th>Adopted by</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastGC [33]</td>
<td>Java-based</td>
<td>96K gates/sec</td>
<td>2.8MBps</td>
<td>PCF [10]</td>
</tr>
<tr>
<td>ObliVM-GC (this paper)</td>
<td>Java-based</td>
<td>670K gates/sec, 1.8M gates/sec (online)</td>
<td>19.6MBps, 54MBps (online)</td>
<td>ObliVM</td>
</tr>
<tr>
<td>GraphSC [24] (extends ObliVM-GC)</td>
<td>Java-based, Parallelizable</td>
<td>580K gates/sec per pair of cores, 1.4M gates/sec per pair of cores (online)</td>
<td>16MBps per pair of cores, 41MBps per pair of cores (online)</td>
<td>GraphSC [24]</td>
</tr>
<tr>
<td>JustGarble [4]</td>
<td>C-based, Hardware AES-NI Garbling only, does not run end-to-end</td>
<td>11M gates/sec</td>
<td>315MBps</td>
<td>TinyGarble [34]</td>
</tr>
<tr>
<td>KSS [11]</td>
<td>Parallel execution in malicious mode Hardware AES-NI</td>
<td>320 gates/sec per pair of cores</td>
<td>2.4MBps per pair of cores</td>
<td>PCF [10]</td>
</tr>
</tbody>
</table>

TABLE I: Summary of known (2-party) Garbled Circuit back ends. The gates/sec metric refer specifically to AND gates, since XOR gates are considered free [5]–[7]. Measurements for different papers are taken on off-the-shelf computers representative of when each paper is written. ObliVM essentially adopts a much better architected and engineered version of FastGC [33]. The focus of this paper is our language, programming abstractions, and compiler. It is our future work to extend JustGarble (C-based, hardware AES-NI) to a fully working back end and integrate it with our language and compiler.

computation. Many first-generation secure computation compilers such as Fairplay [12], TASTY [13], Sharemind [9], CBMC-GC [16], PICCO [14], KSS12 [11] generate target code containing the fully materialized circuits. This approach has the following drawbacks. First, the target code size and compile time are proportional to the circuit size. That is why some works report large compile times (e.g., 8.2 seconds for a circuit of size 700K in KSS12 [11]). Second, the program must be recompiled for every input data size – possibly taking a long time again!

Newer generations of secure computation compilers (e.g., PCF, Wysteria, and SCVM [10], [15], [17]) employ program-style target code instead. Program-style target code is fundamentally a more compact intermediate representation of circuits – fundamentally, the succinctness comes from introducing looping instructions in the intermediate representation, such that the circuit need not be fully unrolled in this intermediate representation.

The resulting program-style target code can then be securely evaluated using a cryptographic protocol such as Garbled Circuit or GMW. Typically these protocols perform per-gate computation – therefore, circuits are effectively 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 using a well-known technique by Huang et al. [33] such that the circuit is never materialized entirely, and thus only a constant amount of working memory is necessary. Further, we stress that on-the-fly circuit generation incurs unnoticeable cost in comparison with the time required to compute the cryptographic protocol. In ObliVM, on-the-fly circuit generation only contributes to less than 0.1% of the total runtime.

Finally, in a concurrent work called TinyGarble [34], Songhori et al. show that by partially materializing a circuit, they can have a somewhat more global view of the circuit. Thus they show how to borrow hardware circuit synthesis techniques to optimize the circuit size by roughly 50% to 80% in comparison with PCF [10]. TinyGarble’s techniques are orthogonal and complementary to this work.

**ORAM support.** Almost all existing secure computation compilers, including most recent ones such as Wysteria [17], PCF [10], and TinyGarble [34], 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 big data sizes. A solution to this problem lies in Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky [40], [41]. To the best of our knowledge, the only known compiler that provides ORAM support is our prior work SCVM which ObliVM builds on. SCVM employs the binary-tree ORAM [42] to implement dynamic memory accesses. Presently, Circuit ORAM is the most efficient ORAM scheme for secure computation – and ObliVM is the first to offer a Circuit ORAM implementation.

**Language expressiveness and formal security.** Most existing languages for secure computation are restrictive in nature. Existing languages [10], [13]–[17] lack essential features such as function calls and public loops inside secret-ifs. This prevents the implementation of a large class of interesting programs. We also offer several other new features such as native primitives, random types (with an affine type system), and generic constants that were lacking in previous languages [10], [13]–[17].

Earlier domain-specific languages [10], [13], [14], [16] for secure computation do not aim to offer formal security. More recent languages such as SCVM [15] and Wysteria [17] offer formal security through new type systems. In comparison, Wysteria’s type system is too restrictive – for example, Wysteria rejects programs with public loops and function calls inside secret-ifs. This prevents many interesting applications – for example, it is not feasible to implement ORAM and oblivious data structures efficiently in Wysteria. On the other hand, Wysteria supports multiple parties, and abstractions for writing
III. PROGRAMMING LANGUAGE AND COMPILER

As mentioned earlier, we wish to design a powerful source language ObliVM-lang such that an expert programmer can i) develop oblivious programming abstractions as libraries and offer them to non-specialist programmers; and ii) implement low-level circuit gadgets atop ObliVM-lang.

ObliVM-lang builds on top of the recent SCVM source language [15] – the only known language to date that supports ORAM abstractions, and therefore offers scalability to big data. In this section, we will describe new features that ObliVM-lang offers and explain intuitions behind our security type system which is formalized in a separate manuscript [21].

As compelling applications of ObliVM-lang, 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 II-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: secret array contents but indices to the array must be public. This array will be secret shared but not placed in ORAMs.
- secure int10[secure 1000] keys: This array will be placed in a secret-shared ORAM, and we allow secret indices into the array.

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):

```java
struct TreeNode<T> {
    public int m root;
    TreeNode<T>[public (1<<m)-1] nodes;
    public int m left, right;
};
struct Tree0m<T> {
    public int m now = this.root, tk;
    T ret;
    while (now != -1) {
        tk = this.nodes[now].key;
        if (tk == key)
            ret = this.nodes[now].value;
        if (tk <= key)
            now = this.nodes[now].left;
        else
            now = this.nodes[now].right;
    }
    return ret
};
```

This code defines a binary search tree implementation of a key-value store, where keys are m-bit integers. The generic constant @m is a variable whose value will be instantiated to a constant. It hints that m bits are enough to represent all the position references to the array. The type int0m 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. [14] 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:

```java
T Tree0m<T>.search(public int0m key) {
    public int0m now = this.root, tk;
    T ret;
    while (now != -1) {
        tk = this.nodes[now].key;
        if (tk == key)
            ret = this.nodes[now].value;
        if (tk <= key)
            now = this.nodes[now].left;
        else
            now = this.nodes[now].right;
    }
    return ret
}
```

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 int0m, 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. We highlight that ObliVM-lang allows arbitrary looping on a public guard (e.g. line 4) without loop unrolling, which cannot be compiled in previous loop-elimination-based work [9].

Function types. Programmers can define a variable to have function type, similar to function pointers in C. To avoid the complexity of handling arbitrary higher order functions, the input and return types of a function type must not be function types. Further, generic types cannot be instantiated with function types.
Native primitives. ObliVM-lang supports native types and native functions. For example, ObliVM-lang’s default back end implementation is ObliVM-GG, which is implemented in Java. Suppose an alternative BigInteger implementation in ObliVM-GG (e.g., using additively homomorphic encryption) is available in a Java class called BigInteger. Programmers can define

```plaintext
typedef BigIntOm = native BigInteger;
```

Suppose that 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:

```plaintext
BigInt Om BigInt Om.add(BigInt Om x, 
   BigInt Om y)= native BigInteger.add;
BigInt Om BigInt Om.multiply(BigInt Om x, 
   BigInt Om y)= native BigInteger.multiply;
BigInt Om BigInt Om.fromInt(int Om y) = native BigInteger.fromString;
int Om BigInt Om.toInt(BigInt Om 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. [15]. We develop a security type system for ObliVM-lang.

Liu et al. [15] has discussed how to prevent memory traces and instruction traces from leaking information. We explain the basic ideas of ObliVM-lang’s type system concerning functions and declassifications, but defer a formal discussion to a separate manuscript [21].

Random numbers and implicit declassifications. Many oblivious programs such as ORAM and oblivious data structures crucially rely on randomness. In particular, their obliviousness guarantee has the following nature: the joint distribution of memory traces is identical regardless of secret inputs (these algorithms typically have a cryptographically negligible probability of correctness failure). ObliVM-lang supports reasoning of such “distributional” trace-obliviousness by providing random types associated with an affine type system. For instance, rnd32 is the type of a 32-bit random integer. A random number will always be secret-shared between the two parties.

To generate a random number, there is a built-in function RND with the following signature:

```plaintext
rndOm RND(public int32 m)
```

This function takes a public 32-bit integer m as input, and returns m random bits. Note that rndOm is a dependent type, whose type depends on values, i.e. m. To avoid the complexity of handling general dependent types, the ObliVM-lang compiler restricts the usage of dependent types to only this built-in function, and handles it specially.

In our ObliVM framework, outputs of a computation can be explicitly declassified with special syntax. Random numbers are allowed implicit declassification – by assigning them to public variables. Here “implicitness” means that the declassification happens not because this is a specified outcome of the computation.

For security, we must ensure that each random number is implicitly declassified at most once for the following reason. When implicitly declassifying a random number, both parties observe the random number as part of the trace. Now consider the following example where s is a secret variable.

```plaintext
1  rnd32 r = RND(32), x;
2  if (s) x = r;
3  else x = RND(32);
......
4  public int32 p = x;
XX public int32 q = r; // NOT OK
```

This program makes an implicit declassification of random variables at Lines 4 and XX – by assigning the random variables to public ones. if the secret value s is 1, then correlated random numbers (r, x) will be publicly revealed in Lines 4 and XX. Otherwise, uncorrelated random numbers (r', x) will be publicly revealed. The problem here is that there is a possible control flow where the random number r will be publicly revealed (i.e., implicitly declassified) twice. Therefore, we use an affine type system to ensure that each random variable is publicly revealed at most once. This way, each time a random variable is publicly revealed, it will introduce a independently uniform variable to the observable trace. In our security proof, a simulator can just sample this random number to simulate the trace. Despite its simplicity, it turns out that the above example reflects the essence of what is needed to implement oblivious RAM and oblivious data structures in our source language. We refer the readers to Sections [IV] and [V-B] for details.

Function calls and phantom functions. A straightforward idea to prevent stack behavior from leaking information is to enforce function calls in a public context. Then the requirement is that each function’s body must satisfy memory- and instruction-trace obliviousness. Further, by defining native functions, ObliVM-lang implicitly assumes that their implementations 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 dual modes, a real mode and 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 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 padding ideas used in several previous works [43], [44].

We will illustrate the use of phantom function with the following prefixSum example. 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).
The keyword `phantom` indicates that the function `prefixSum` is a phantom function.

Consider the following code to call the phantom functions:

```java
if (s) then x = prefixSum(n);
```

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

```java
prefixSum(idx, indicator)
```

where `indicator` means whether the function will be called in the real or phantom mode. To achieve the first goal, the global variable will be modified only if `indicator` is false. The compiler will compile the code in line 4 into the following pseudo-code:

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

It is easy to see, that all instructions will be executed, and thus the generated traces are identical regardless of the value of `indicator`. Note, that such a function is not implementable in any prior loop-unrolling based compiler, since `n` is provided at runtime only.

It is worth noticing that phantom function relaxed the restriction posed by previous memory trace oblivious type systems [31], which do not allow looping in the secure context (i.e. within a secret conditional). The main difficulty in previous systems was to quantify the numbers of loop iterations in the two branches of an if-statement, and to enforce the two numbers to be the same. Phantom functions remove the need of this analysis by executing both branches, with one branched really executed, and the other executed phantomly. As long as an adversary is unable to distinguish between a real execution from a phantom one, the secret guard of the if-statement will not be leaked, even when loops are virtually present (i.e. in a phantom function).

### IV. USER-FACING OBLIVIOUS PROGRAMMING ABSTRACTIONS

Programming abstractions such as MapReduce and GraphLab have been popularized in the parallel computing domain. In particular, programs written for a traditional imperative-style programming language like C, compile-time optimizations would have been much more limited.

#### A. MapReduce Programming Abstractions

An interesting observation is that “parallelism facilitates obliviousness” [45], [46]. If a program (or part of a program) can be efficiently expressed in parallel programming paradigms such as MapReduce and GraphLab [47], [48] (with a few additional constraints), there is an efficient oblivious algorithm to compute this task. We stress that in this paper, we consider MapReduce merely as a programming abstraction that facilitates obliviousness – in reality we compile MapReduce programs to sequential implementations that runs on a single thread. Parallelizing the algorithms is outside the scope of this paper. However, in a subsequent work GraphSC [24] jointly with our collaborators, we do offer parallel oblivious implementations of programs written in a GraphLab abstraction – and doing so requires the design of new, non-trivial parallel oblivious algorithms detailed in the GraphSC paper [24].

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

- The `map` operation: takes an array denoted \( \{ \alpha_i \}_{i \in [n]} \) where each \( \alpha_i \in D \) for some domain \( D \), and a function \( \text{mapper} : D \rightarrow K \times V \). Now `map` would apply \( (k_i, v_i) := \text{mapper}(\alpha_i) \) to each \( \alpha_i \), and output an array of key-value pairs \( \{(k_i, v_i)\}_{i \in [n]} \).

- The `reduce` operation: takes in an array of key-value pairs denoted \( \{(k_i, v_i)\}_{i \in [n]} \) and a function \( \text{reducer} : K \times V^2 \rightarrow V \). For every unique key value \( k \) in this array, let \( (k, v_{i_1}), (k, v_{i_2}), \ldots, (k, v_{i_m}) \) denote all occurrences with the key \( k \). Now the `reduce` operation applies the following operation in a streaming fashion:

\[
R_k := \text{reducer}(k, \ldots, \text{reducer}(k, \text{reducer}(k, v_{i_1}, v_{i_2}), v_{i_3}), \ldots, v_{i_m})
\]

The result of the `reduce` operation is an array consisting of a pair \( (k, R_k) \) for every unique \( k \) value in the input array.

Goodrich and Mitzenmacher [45] observe that any program written in a streaming MapReduce abstraction can be converted to efficient oblivious algorithms, and they leverage this observation to aid the construction of an ORAM scheme.

- The `map` operation is inherently oblivious, and can be done by making a linear scan over the input array.

- The `reduce` operation can be made oblivious through an oblivious sorting (denoted o-sort) primitive.
  - First, o-sort the input array in ascending order of the key, such that all pairs with the same key are grouped together.
  - Next, in a single linear scan, apply the `reducer` function: i) If this is the last key-value pair for some key \( k \), write down the result of the aggregation \( (k, R_k) \). ii) Else, write down a dummy entry 1.
Providing the streaming MapReduce abstraction in ObliVM-lang. It is easy to implement the streaming MapReduce abstraction as a library in our source language ObliVM-lang. The ObliVM-lang implementation of streaming MapReduce paradigm is provided in Figure 1.

```java
1 Pair<K,V>[public n] MapReduce@m×n<K,V>
2 (I[public m] data, Pair<K, V> mapper(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) {
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<int32, Pair<K, V>>[public m] res;
13 for (i=0; i+1<m; i=i+1) {
14 res[i].k.val = key;
15 res[i].v.v = val;
16 if (cmp(key, d2[i+1].k)==0) {
17 res[i].k.val = 0;
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@[int32, Pair<K, V>> (res, 1, zeroOneCmp);
29 Pair<K, V>[public n] top;
30 for (i=0; i < n; i = i + 1)
31 top[i] = res[i].v;
32 return top;
33 }
34 }
```

Fig. 1: Streaming MapReduce in ObliVM-lang. See Section IV-A for oblivious algorithms for the streaming MapReduce paradigm. 

- Finally, o-sort all the resulting entries to move ⊥ to the end.

This program counts the frequency of each values in [0..n−1] in the array a of size m. Since the program makes dynamic memory accesses, the SCVM compiler would decide to put the array c inside an ORAM.

An end user can write the same program using a simple MapReduce abstraction as follows. Our ObliVM-lang compiler would generate target code that relies on oblivious sorting primitives rather than generic ORAM, improving the performance by a logarithmic factor in comparison with the SCVM implementation. In Section VII, we show that the practical performance gain ranges from 10× to 400×.

```java
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;
}
```

The top-level program can launch the computation using

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

B. Programming Abstractions for Data Structures

We now explain how to provide programming abstractions for a class of pointer-based oblivious data structures described by Wang et al. [26]. Figure 2b gives an example, where an expert programmer provides library support for implementing a class of pointer-based data structures such that a non-specialist programmer can implement data structures which will be compiled to efficient oblivious algorithms that outperform generic ORAM. We stress that while we give a stack example for simplicity, this paradigm is also applicable to other pointer-based data structures, such as AVL tree, heap, and queue.
Implementing oblivious data structure abstractions in ObliVM. We assume that the reader is familiar with the oblivious data structure algorithmic techniques described by Wang et al. [26]. To support efficient data structure implementations, an expert programmer implements two important objects (see Figure 2b):

- A Pointer object stores two important pieces of information: an index variable that stores the logical identifier of the memory block pointed to (each memory block has a globally unique index); and a pos variable that stores the random leaf label in the ORAM tree of the memory block.
- A SecStore object essentially implements an ORAM, and provides the following member functions to an end-user: The SecStore.remove function essentially is a syntactic sugar for the ORAM’s readAndRemove interface [29], [42], and the SecStore.add function is a syntactic sugar for the ORAM’s Add interface [29], [42]. Finally, the SecStore.allocate function returns a new Pointer object to the caller. This new Pointer object is assigned a globally unique logical identifier (using a counter cnt that is incremented each time), and a fresh random chosen leaf label RND(m).

Stack implementation by a non-specialist programmer. Given abstractions provided by the expert programmer, a non-specialist programmer can now implement a class of data structures such as stack, queue, heap, AVL Tree, etc. Figure 2a gives a stack example.

Role of affine type system. We use Figure 2b as an example to illustrate how our rnd types with their affine type system can ensure security. As mentioned earlier, rnd types have an affine type system. This means that each rnd can be declassified (i.e., made public) at most once. In Figure 2b the oram.readAndRemove call will declassify its argument rnd@m pos inside the implementation of the function body. From an algorithms perspective, this is because the leaf label pos will be revealed during the readAndRemove operation, incurring a memory trace where the value rnd@m pos will be observable by the adversary.

C. Loop Coalescing and New Oblivious Graph Algorithms

When a program has nested loops, we can convert the nested loops into a single loop using a loop coalescing technique. This technique appeared in the compiler optimization literature [49], and more recently, was independently adopted by Wahby et al. in designing automated compilers for verifiable computation [50].

ObliVM incorporates loop coalescing as a programming abstraction. In combination with oblivious data structures, it allowed us to design novel oblivious graph algorithms such as Dijkstra’s shortest path and minimum spanning tree for sparse graphs. Loop coalescing is non-trivial to embed as a library in ObliVM-lang. We therefore support this programming abstraction by introducing special syntax and modifications to our compiler. Specifically, we introduce a new syntax called bounded-for loop as shown in Figure 3. For succinctness, in this section, we will present pseudo-code.

In the program in Figure 3 the bwhile(n) and bwhile(m) syntax at Lines 1 and 3 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 – over all iterations of the outer loop.

To deal with loop coalescing, the compiler partitions the code within a bounded-loop into code blocks, each of which will branch at the end. The number of execution times for each code block will be computed as the bound number for the...
bwhile(n) (; u<n;) {
5        total = total + 1;
6        i=s[u];
7        // do something
8    } else if (state==3) {
9        u=u+1; state = (u<n) ? 1 : -1
10    } // else execution is finished
11} // block 3
12

TABLE II: Summary of algorithmic results. All costs reported are in terms of circuit size. The asymptotic notation omits
13

The outer loop will be executed at most \( n \) times in total, the inner loop will be executed at most \( m \) times in total – over all iterations of the outer loop. A naive approach compiler would pad the outer and inner loop to \( n \) and \( m \)
16

respectively, incurring \( O(nm) \) cost. Our loop coalescing technique achieves \( O(n+m) \) cost instead.
17

Sparse Graph
18

Oblivious Dijkstra shortest path for sparse graphs. It is an open problem how to compute single source shortest path
19

(SSSP) obliviously for sparse graphs more efficiently than generic ORAM approaches. Blanton et al. \[51\] designed a
20

solution for a dense graph, but their technique cannot be applied when the graph is sparse.
21

Recall that the priority-queue-based Dijkstra’s algorithm has to update the weight whenever a shorter path is found to
22

any vertex. In an oblivious version of Dijkstra’s, this operation dominates the overhead, as it is unclear how to realize it
23

more efficiently than using generic ORAMs. Our solution to oblivious SSSP is more efficient thanks to (1) avoiding this
24

weight update operation, and (2) a loop coalescing technique.
25

Avoiding weights updating. This is accomplished by two changes to a standard priority-queue-based Dijkstra’s algo-
26

rithm, i.e., lines 6-7 and line 12 in Algorithm\[7\]. The basic idea is, whenever a shorter distance \( \text{newDist} \) from \( s \) to a vertex
27

inner most bounded-loop that contains the code block. Then the compiler will transform a bounded loop into a normal loop,
28

whose body simulates a state machine that each state contains a code block, and the branching statement at the end of each
29

code block will be translated into an assignment statement that moves the state machine into a next state. The total number of
30

iterations of the enabled normal loop is the summation of the execution times for all code blocks. Figure 3 illustrates this
31

compilation process.
32

We now show how this loop coalescing technique leads to new novel oblivious graph algorithms.
33

Oblivious Dijkstra shortest path for sparse graphs. It is an open problem how to compute single source shortest path
34

(SSSP) obliviously for sparse graphs more efficiently than generic ORAM approaches. Blanton et al. \[51\] designed a
35

solution for a dense graph, but their technique cannot be applied when the graph is sparse.
36

Recall that the priority-queue-based Dijkstra’s algorithm has to update the weight whenever a shorter path is found to
37

any vertex. In an oblivious version of Dijkstra’s, this operation dominates the overhead, as it is unclear how to realize it
38

more efficiently than using generic ORAMs. Our solution to oblivious SSSP is more efficient thanks to (1) avoiding this
39

weight update operation, and (2) a loop coalescing technique.
40

Avoiding weights updating. This is accomplished by two changes to a standard priority-queue-based Dijkstra’s algo-
41

rithm, i.e., lines 6-7 and line 12 in Algorithm\[7\]. The basic idea is, whenever a shorter distance \( \text{newDist} \) from \( s \) to a vertex
42

u is found, instead of updating the existing weight of u in the heap, we insert a new pair (newDist, u) into the priority
43

queue. This change can result in multiple entries for the same vertex in the queue, leading to two concerns: (1) the size of
44

the priority queue cannot be bounded by \( V \); and (2) the same vertex might be popped and processed multiple times from
45

the queue. Regarding the first concern, we note the size of the queue can be bounded by \( E = O(V^2) \) (since \( E = o(V^2) \)
46

bwhile(n) (; u<n;) {
34        total = total + 1;
35        i=s[u];
36        // do something
37    } else if (state==3) {
38        u=u+1; state = (u<n) ? 1 : -1
39    } // else execution is finished
40} // block 3
41

TABLE II: Summary of algorithmic results. All costs reported are in terms of circuit size. The asymptotic notation omits
42

the bit-length of each word for simplicity. Our oblivious Dijkstra’s algorithm and oblivious Prim’s algorithm can be composed
43

using our novel loop coalescing programming abstraction and oblivious data structures. Our oblivious DFS algorithm requires
44

independent novel techniques. Due to space constraint, we only describe the oblivious Dijkstra’s algorithm as an example of
45

loop coalescing. We defer the full description of our oblivious MST and DFS algorithms to Appendix A.
46

Algorithm 1 Dijkstra’ algorithm with bounded for
47

Secret Input: \( s \): the source node
48

Secret Input: \( e \): concatenation of adjacency lists stored in a
49

single ORAM array. Each vertex’s neighbors are stored adjacent to each other.
50

Secret Input: \( s[u] \): sum of out-degree over vertices from 1 to
51

u.
52

Output: \( \text{dis} \): the shortest distance from source to each node
53

1: \( \text{dis} := [\infty, \infty, \ldots, \infty] \)
54
2: \( \text{PQ}.\text{push}(0, s) \)
55
3: \( \text{dis}[s] := 0 \)
56
4: \( \text{bwhile}(V)(\text{PQ}.\text{empty}()) \)
57
5: \( (\text{dist}, u) := \text{PQ}.\text{deleteMin}() \)
58
6: if(\( \text{dis}[u] == \text{dist} \)) then
59
7: \( \text{dis}[u] := -\text{dis}[u] \);
60
8: \( \text{bfor}(\text{E}[i := s[u]; i < s[u+1]; i = i + 1) \)
61
9: \( (u, v, w) := \text{E}[i] \);
62
10: \( \text{newDist} := \text{dist} + w \)
63
11: if(\( \text{newDist} < \text{dis}[v] \)) then
64
12: \( \text{dis}[v] := \text{newDist} \)
65
13: \( \text{PQ}.\text{insert}(<\text{newDist}, u) \)
66
7
8
9
10
11
12
13

Fig. 3: Loop coalescing. The outer loop will be executed at most \( n \) times in total, the inner loop will be executed at most \( m \)
times in total – over all iterations of the outer loop. A naive approach compiler would pad the outer and inner loop to \( n \) and \( m \)
respectively, incurring \( O(nm) \) cost. Our loop coalescing technique achieves \( O(n+m) \) cost instead.

inner most bounded-loop that contains the code block. Then the compiler will transform a bounded loop into a normal loop,
whose body simulates a state machine that each state contains a code block, and the branching statement at the end of each
code block will be translated into an assignment statement that moves the state machine into a next state. The total number of
iterations of the enabled normal loop is the summation of the execution times for all code blocks. Figure 3 illustrates this
compilation process.

We now show how this loop coalescing technique leads to new novel oblivious graph algorithms.

Oblivious Dijkstra shortest path for sparse graphs. It is an open problem how to compute single source shortest path
(SSSP) obliviously for sparse graphs more efficiently than generic ORAM approaches. Blanton et al. \[51\] designed a
solution for a dense graph, but their technique cannot be applied when the graph is sparse.

Recall that the priority-queue-based Dijkstra’s algorithm has to update the weight whenever a shorter path is found to
any vertex. In an oblivious version of Dijkstra’s, this operation dominates the overhead, as it is unclear how to realize it
more efficiently than using generic ORAMs. Our solution to oblivious SSSP is more efficient thanks to (1) avoiding this
weight update operation, and (2) a loop coalescing technique.

Avoiding weights updating. This is accomplished by two changes to a standard priority-queue-based Dijkstra’s algo-
rithm, i.e., lines 6-7 and line 12 in Algorithm\[7\]. The basic idea is, whenever a shorter distance \( \text{newDist} \) from \( s \) to a vertex
ObliVM-lang provides a rich language for implementing cryptographic circuits. We demonstrate this by deriving an oblivious Dijkstra's algorithm using the ObliVM-lang and showing how to implement Karatsuba multiplication in ObliVM-lang. The Karatsuba algorithm reduces the number of multiplications by using a divide-and-conquer strategy, which is particularly efficient for large numbers.

Algorithm 2: Oblivious Dijkstra's algorithm

**Secret Input:** \( s \) is the same as Algorithm 1

**Output:** \( \text{dis} \): the shortest distance from \( s \) to each node

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

for sparse graphs). Hence, each priority queue insert and deleteMin operation can still be implemented obliviously in \( O(\log^2 V) \) [26]. The second concern is resolved by the check in lines 6-7: every vertex will be processed at most once because \( \text{dist}[v] \) will be set negative once vertex \( v \) is processed.

Loop coalescing. In Algorithm 1 the two nested loops (line 4 and line 8) use secret data as guards. In order not to leak the secret loop guards, a naive approach is to iterate each loop a maximal number of times (i.e., \( V + E \), as \( V \) alone is considered secret).

Using our loop coalescing technique, we can derive an oblivious Dijkstra's algorithm that asymptotically outperforms a generic ORAM baseline for sparse graphs. The resulting oblivious algorithm is described in Algorithm 2. Note that at most \( V \) vertices and \( E \) edges will be visited, we coalesce the two loops into a single one. The code uses a state variable innerLoop to indicate whether a vertex or an edge is being processed. Each iteration deals with one of a vertex (lines 5-8), an edge (lines 15-18), or the end of a vertex's edges (line 13). So there are \( 2V + E \) iterations in total. Note the ObliVM-lang compiler will pad the if-branches in Algorithm 2 to ensure obliviousness. Further, an oblivious priority queue is employed for \( \text{PQ} \).

Cost analysis. In Algorithm 2 each iteration of the loop (lines 3-18) makes a constant number of ORAM accesses and two priority queue primitives (insert and deleteMin, both implemented in \( O(\log^2 V) \) time). So, the total runtime is \( O((V + E) \log^2 V) \).

Additional algorithmic results. Summarized in Table II, our loop coalescing also immediately gives a new oblivious Minimum Spanning Tree (MST) algorithm whose full description is deferred to Appendix A. Additionally, in the process of developing rich libraries for ObliVM, we also designed a novel oblivious Depth First Search (DFS) algorithm that asymptotically outperforms a generic ORAM baseline for dense graphs. The new DFS requires new algorithmic techniques, and we defer its full description to Appendix A.

Implementing Karatsuba multiplication. Figure 4 contains the implementation of Karatsuba multiplication [32] in ObliVM-lang. Karatsuba multiplication implements the following recursive algorithm to compute multiplication of two \( n \) bit integers, \( x \) and \( y \), taking \( O(n\log_2 3) \) amount of time. As a quick overview, the algorithm works as follows. First, express the \( n \)-bit integers \( x \) and \( y \) as the concatenation of \( \frac{n}{2} \)-bit integers: \( x = a*2^n + b \), \( y = c*2^n + d \). Now, \( x*y \) can be calculated as follows:

\[
t1 = a*c; t2 = b*d; t3 = (a+b)*(c+d);
\]

\[
x*y = t1<<n + t2 + (t3-t1-t2)<<\lfloor n/2 \rfloor;
\]

where the multiplications \( a*c \) and \( b*d \) are performed through a recursive call to the Karatsuba algorithm itself (until the bit-length is small enough).

To implement Karatsuba efficiently, we need to perform operations on a subset of bits. We hence introduce the following syntactic sugar in ObliVM-lang: In lines 6 to 9 of Figure 4, the syntax \( \text{num}@i \rightarrow j \) means extracting the part of integer \( \text{num} \) from \( i \)-th bit to \( j \)-th bit.

V. IMPLEMENTING RICH CIRCUIT LIBRARIES IN SOURCE LANGUAGE

A. Case Study: Basic Arithmetic Operations

The rich language features provided by ObliVM-lang make it possible to implement complex arithmetic operations easily and efficiently. We give a case study to demonstrate how to use ObliVM-lang to implement Karatsuba multiplication.
leads to a more efficient circuit generation.

(dummy indicates the value of this type can be null)

(dummy array of blocks). The implementation consists of buckets (i.e. arrays of ORAM blocks), and a stash (i.e. an integer. Here, an implicit declassification on an implicit declassified at most once.

return res;

Line 11 demonstrates how readAndRemove can be implemented. Taking the input of an secret integer index \( \text{id} \), and a random position label \( \text{pos} \), the label \( \text{pos} \) is first declassified into public. Then affine type system allows declassifying \( \text{pos} \) once, i.e. \( \text{pos} \) is never used for the rest of the program. Further in a function calling readAndRemove with inputs \( \text{arg1} \) and \( \text{arg2} \), \( \text{arg2} \) cannot be used either for the rest of the program. This is crucial to enforce that every position labels will use revealed only once after its generation, and, to our best knowledge, no prior work enables such an enforcement in a compiler.

VI. BACK END ARCHITECTURE

Our compiler emits code to a Java-based secure computation back end called ObliVM-GC. We defer details of ObliVM-GC to our online full version \[39\].

VII. EVALUATION

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. Due to well-known Free XOR techniques \[5\]–[7], the cost of evaluating XOR gates is insignificant in comparison with AND gates. Therefore, a primary performance metric is 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, and is one of the most popular metrics used in earlier works \[10\], \[11\], \[13\], \[16\], \[25\], \[26\], \[33\], \[53\], \[54\].

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, and how these runtimes will be affected by the underlying software and hardware configurations.

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

B. Comparison with Previous Automated Approaches

The first general-purpose secure computation system, Fairplay, was built in 2004 \[12\]. Since then, several improved systems were built \[9\], \[11\], \[13\], \[14\], \[16\], \[33\]–[35]. Except for our prior work SCVM \[15\], existing systems provide no support for ORAM – and therefore, each dynamic memory access would be compiled to a linear scan of memory.

We now evaluate the speedup ObliVM achieves relative to previous approaches. To illustrate the sources of the speedup, we consider the following sequence of progressive baselines. We start from Baseline 1 which is representative of a state-of-the-art automated secure computation system. We then add one feature at a time to the baseline, resulting in the next baseline, until we arrive at Baseline 5 which is essentially our ObliVM system.

Baseline 1: A state-of-the-art automated system with no ORAM support. Baseline 1 is intended to characterize a state-of-the-art automated secure computation system with no ORAM support. We assume a compiler that can detect public memory accesses (whose addresses are statically inferrable), and directly make such memory accesses. For each each dynamic memory access (whose address depends on secret inputs), a linear scan of memory is employed. Baseline 1 is effectively a lower-bound
Parameters for Figure 6

<table>
<thead>
<tr>
<th>Application</th>
<th>Abstraction/Implementation</th>
<th>Parameters for Table IV</th>
<th>Parameters for Table V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra's Algorithm</td>
<td>Loop coalescing abstraction (see Section IV-C)</td>
<td>$V = 2^{14}, E = 3V$</td>
<td>$V = 2^{10}, E = 3V$</td>
</tr>
<tr>
<td>MST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Map/Set Binary</td>
<td>Oblivious data structure abstraction (see Section IV-B)</td>
<td>$N = 2^{27}, K = 32, V = 480$</td>
<td>$N = 2^{23}, K = 32, V = 992$</td>
</tr>
<tr>
<td>AMS Sketch</td>
<td>Compile-time optimizations: split data into separate ORAMs [15]</td>
<td>$\epsilon = 6 \times 10^{-1}, \delta = 2^{-20}$</td>
<td>$\epsilon = 2.4 \times 10^{-4}, \delta = 2^{-20}$</td>
</tr>
<tr>
<td>K-Means Sketch</td>
<td>MapReduce abstraction (see Section IV-A)</td>
<td>$N = 2^{18}$</td>
<td>$N = 2^{16}$</td>
</tr>
<tr>
<td>Count Min Sketch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE III: List of applications used in Figures 6. 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, $\epsilon, \delta$ stand for relative error and failure probability; for K-Means, $N$ stands for number of points.

estimate of the cost incurred by CMBC-GC [16], a state-of-the-art system in 2012.

- **Baseline 2: With GO-ORAM [40].** In Baseline 2, we implement the GO-ORAM scheme on top of Baseline 1. Dynamic memory accesses made by a program will be compiled to GO-ORAM accesses. We make no additional compile-time optimizations.

- **Baseline 3: With Circuit ORAM [29].** Baseline 3 is essentially the same as Baseline 2 except that we now replace the ORAM scheme with a state-of-the-art Circuit ORAM scheme [29].

- **Baseline 4: Language and compiler.** Baseline 4 assumes that the ObliVM language and compiler are additionally employed (on top of Baseline 3), resulting in additional savings due to our compile-time optimizations as well as our oblivious programming abstractions.

- **Baseline 5: Back end optimizations.** In Baseline 5, we employ additional back end optimizations atop Baseline 4. Baseline 5 reflects the performance of the actual ObliVM system.

We consider a set of applications in our evaluation as described in Table III. We select several applications to showcase our oblivious programming abstractions, including MapReduce, loop coalescing, and oblivious data structure abstractions. For all applications, we choose moderately large data sizes ranging from 768KB to 10GB. For data structures (e.g., Heap, Map/Set) and binary search, for Baseline 1, we assume that each operation (e.g., search, add, delete) is done with a single linear scan. For Baseline 2 and 3, we assume that a typical sub-linear implementation is adopted. For all other applications, we assume that Baseline 1 to 3 adopt the most straightforward implementation of the algorithm.

Results. Figure 6 shows the speedup we achieve relative to a state-of-the-art automated system that does not employ ORAM [16]. This speedup comes from the following sources:

- **No ORAM to GO-ORAM:** For most of the cases, the data size considered was not big enough for GO-ORAM to be competitive to a linear-scan ORAM. The only exception was AMS sketch, where we chose a large sketch size. In this case, using GO-ORAM would result in a 300× speedup in comparison with no ORAM (i.e., linear-scan for each dynamic memory access). This part of the speedup is reflected in purple in Figure 6. Here the speedup stems from a reduction in circuit size (as measured by the number of AND gates).

- **Circuit ORAM:** The red parts in Figure 6 reflect the multiplicative speedup attained when we instead use Circuit ORAM (as opposed to no ORAM or GO-ORAM, whichever is faster). This way, we achieve an additional 51× to 530 performance gains – reflected by a reduction in the total circuit size.

Language and compiler: As reflected by the blue bars in Figure 6, our oblivious programming abstractions and compile-time optimizations bring an additional 2× to 2500× performance savings on top of a generic Circuit ORAM-based approach. This speedup is also measurable in terms of reduction in the circuit size.

Back end optimizations: Our ObliVM-GC is a better architected and more optimized version of its predecessor FastGC [33] which is employed by CMBC-GC [16]. FastGC [33] reported a garbling speed of 96K AND gates/sec, whereas ObliVM garbles at 670K AND gates/sec on a comparable machine. In total, we achieve an 7× overall speedup compared with FastGC [33].

We stress, however, that ObliVM’s main contribution is not the back end implementation. In fact, it would be faster to hook up ObliVM’s language and compiler with a JustGarble-like system that employs a C-based implementation and hardware AES-NI. However, presently JustGarble does not provide a fully working end-to-end protocol. Therefore, it is an important direction of future work to extend JustGarble to a fully working protocol, and integrate it into ObliVM.

Comparison with SCVM. In comparison with SCVM, ObliVM’s offers the following new features: 1) new oblivious programming abstractions; 2) Circuit ORAM implementation that is $20 \times$ to $30 \times$ times faster than SCVM’s binary-tree ORAM implementation for 4MB to 4GB data sizes; and 3) ability to implement low-level gadgets including the ORAM algorithm itself in the source language.

In the online full version [39], we give a detailed comparison with an SCVM-like system. Since the design of efficient ORAM algorithms is mainly the contribution of the Circuit ORAM paper [29], here we focus on evaluating the gains from programming abstractions. Therefore, instead of comparing with SCVM per se, we compare with SCVM + Circuit ORAM instead (i.e., SCVM with its ORAM implementation updated to the latest Circuit ORAM).
C. ObliVM vs. Hand-Crafted Solutions

We show that ObliVM achieves competitive performance relative to hand-crafted solutions for a wide class of common tasks. We also show that ObliVM significantly reduces development effort in comparison with previous secure computation frameworks.

**Competitive performance.** For a set of applications, including Heap, Map/Set, AMS Sketch, Count-Min Sketch, and K-Means, we compared implementations auto-generated by ObliVM with implementations hand-crafted by human experts. Here the human experts are authors of this paper. We assume that the human experts have wisdom of employing the most efficient, state-of-the-art oblivious algorithms when designing circuits for these algorithms. For example, Histogram and K-Means algorithms are implemented with oblivious sorting protocols instead of generic ORAM. Heap and Map/Set employ state-of-the-art oblivious data structure techniques [26]. The graph algorithms including Dijkstra and MST employ novel oblivious algorithms proposed in this paper. In comparison, our ObliVM programs for the same applications do not require special security expertise to create. The programmer simply has to express these tasks in the programming abstractions we offer whenever possible. Over the suite of application benchmarks we consider, our ObliVM programs are competitive to hand-crafted implementations – and the performance difference is only $0.5\% - 2\%$ throughout.

We remark that the hand-crafted circuits are not necessarily the optimal circuits for each computation task. However, they do represent asymptotically the best known algorithms (or new algorithms that are direct implications of this paper). It is conceivable that circuit optimization techniques such as those proposed in TinyGarble [34] can further reduce circuit sizes by a small constant factor (e.g., 50%). We leave this part as an interesting direction of future research.

**Developer effort.** We use two concrete case studies to demonstrate the significant reduction of developer effort enabled by ObliVM.

*Case study: ridge regression.* Ridge regression [55] takes as input a large number of data points and finds the best-fit linear curve for these points. This algorithm is an important building block in various machine-learning tasks [54]. Previously, Nikolaenko et al. [54] developed a system to securely evaluate ridge regression, using the FastGC framework [33], which took them roughly three weeks [56]. In contrast, we spent two student-hours to accomplish the same task using ObliVM. In addition to the speedup gain from ObliVM-GC back end, our optimized libraries result in 3x smaller circuits with aligned parameters. We defer the detailed comparison to the online technical report [39].

*Case study: oblivious data structures.* Oblivious AVL tree (i.e., the Map/Set data structure) is an example algorithm that was previously too complex to program as circuits, but now becomes very easy with ObliVM. In an earlier work [26], we designed an oblivious AVL tree algorithm, but were unable to implement it due to high programming complexity. Now, with ObliVM, we implement an AVL tree with 311 lines of code in ObliVM-lang, consuming under 10 student-hours (including the implementation as well as debugging).

We stress that it is not possible to implement oblivious AVL tree in previous languages for secure computation, including the state-of-the-art Wysteria [17].

D. End-to-End Application Performance

Currently in ObliVM-GC, we implemented a standard garbling scheme with Garbled Row Reduction [36] and FreeXOR [5]. We also implemented an OT extension protocol proposed by Ishai et al. [3] and a basic OT protocol by Naor and Pinkas [57].

**Setup.** For evaluation, here we consider a scenario where a client secret shares its data between two non-colluding cloud providers a priori. 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 among the two cloud providers. We do not measure this setup cost in our evaluation – this cost can depend highly on the available bandwidth.
TABLE IV: Application performance. Actual measured numbers are in bold. The remainder are estimated numbers and should be interpreted with care. ObliVM numbers for basic instructions and sublinear-time algorithms are the mean of 20 runs. Since for all these applications, our measurements have small spread (all runs are within 6% from the mean), we use a single run for linear-time and super-linear algorithms (the same for Table V).

<table>
<thead>
<tr>
<th>Program</th>
<th>Input size</th>
<th>CMBC-GC (estimate)</th>
<th>ObliVM Framework (estimate)</th>
<th>ObliVM + JustGarble (estimate)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#AND gates</td>
<td>Total time</td>
<td>#AND gates</td>
</tr>
<tr>
<td>Integer addition</td>
<td>1024 bits</td>
<td>2977</td>
<td>31ms</td>
<td>1024</td>
</tr>
<tr>
<td>Integer mult.</td>
<td>1024 bits</td>
<td>6.4M</td>
<td>66.4s</td>
<td>572K</td>
</tr>
<tr>
<td>Integer Comparison</td>
<td>16384 bits</td>
<td>32K</td>
<td>335.7ms</td>
<td>16384</td>
</tr>
<tr>
<td>Floating point addition</td>
<td>64 bits</td>
<td>10K</td>
<td>104ms</td>
<td>3035</td>
</tr>
<tr>
<td>Floating point mult.</td>
<td>64 bits</td>
<td>10K</td>
<td>104ms</td>
<td>4312</td>
</tr>
<tr>
<td>Hamming distance</td>
<td>1600 bits</td>
<td>30K</td>
<td>310ms</td>
<td>3200</td>
</tr>
</tbody>
</table>

Basic instructions

| K-Means          | 0.5MB      | 550B      | 66d        | 2260M      | 62.1min    | 23.6min     | 4.58min    | 1.9min      |
| Dijkstra’s Algorithm | 48KB      | 755B      | 91d        | 10B        | 12.6h      | 3.09h       | 20.4min    | 8.5min      |
| MST              | 48KB       | 755B      | 91d        | 9.6B       | 12.4h      | 3h          | 19.6min    | 8.2min      |
| Histogram        | 0.25MB     | 137B      | 16.5d      | 866M       | 21.5min    | 8.56min     | 1.7min     | 42.5s       |

Linear or super-linear algorithms

| Heap             | 1GB        | 32B       | 3.9d       | 12.5M      | 59.3s      | 10.42s      | 1.5s       | 625ms       |
| Map/Set          | 1GB        | 32B       | 3.9d       | 23.9M      | 117.2s     | 20.67s      | 2.9s       | 1.2s        |
| Binary Search    | 1GB        | 32B       | 3.9d       | 1562K      | 7.36s      | 1.34s       | 189ms      | 78.8ms      |
| Count Min Sketch | 0.31GB     | 9.9B      | 30.8h      | 8088K      | 20.77s     | 6.4s        | 0.98s      | 0.41s       |
| AMS Sketch       | 1.25GB     | 40B       | 5.18d      | 9949K      | 36.76s     | 9.95s       | 1.21s      | 504ms       |

Sublinear-time algorithms

between the client and the two cloud providers. Therefore, our evaluation begins assuming this one-time setup has completed.

End-to-end application performance. In Table IV we consider three types of applications, basic instructions (e.g., addition, multiplication, and floating point operations); linear or super-linear algorithms (e.g., Dijkstra, K-Means, Minimum Spanning Tree, and Histogram); and sublinear-time algorithms (e.g., Heap, Map/Set, Binary Search, Count Min Sketch, AMS Sketch). We report the circuit size, online and total costs for a variety of applications at typical data sizes.

In Table IV, we also compare ObliVM with a state-of-the-art automated secure computation system CMBC-GC [16]. We note that the authors of CMBC-GC did not run all of these application benchmarks, so we project the performance of CMBC-GC using the following estimate: we first change our compiler to adopt a linear scan of memory upon dynamic memory accesses – this allows us to obtain an estimate of the circuit size CMBC-GC would have obtained for the same applications. For the set of application benchmarks (e.g., K-Means, MST, etc) CMBC-GC did report in their paper, we confirmed that our circuit size estimates are always a lower bound of what CMBC-GC reported. We then estimate the runtime of CMBC-GC based on their reported 96K AND gates per sec – assuming that a network bandwidth of at least 2.8MBps is provisioned.

As mentioned earlier, the focus of this paper is our language and compiler, not the back end cryptographic implementation. It should be relatively easy to integrate our language and compiler with a JustGarble-like back end that employs hardware AES-NI. In Table IV we also give an estimate of the performance we anticipate if we ran our ObliVM-generated circuits over a JustGarble-like back end. This is calculated using our circuit sizes and the 11M AND gates/sec performance number reported by JustGarble [4].

- **Online cost.** To measure online cost, we assume that all work that is independent of input data is performed offline, including garbling and input-independent OT pre-processing. Our present ObliVM implementation achieves an online speed of 1.8M gates/sec consuming roughly 54MBps network bandwidth.

- **Offline cost.** When no work is deferred to an offline phase, ObliVM achieves a garbling speed of 670K gates/sec consuming 19MBps network bandwidth.

Slowdown relative to a non-secure baseline. For completeness, we now describe ObliVM’s slowdown in comparison with a non-secure baseline where computation is performed in cleartext. As shown in Table V our slowdown relative to a non-secure baseline is application dependent, and ranges from $45 \times 10^9$ to $9.3 \times 10^9 \times$. We also present the estimated slowdown if a JustGarble-like back end is used for ObliVM-generated circuits. These numbers are estimated based on our circuit sizes as well as the reported 11M AND gates/sec performance metric reported by JustGarble [4].

In particular, we elaborate on the following interesting cases. First, the distributed genome-wide association study (GWAS) application is Task 1 in the iDASH secure genomic analysis competition [58], with total data size 380KB. This task achieves a small slowdown, because part of the computation is done locally – specifically, Alice and Bob each performs some local pre-processing to obtain the allele frequen-
cies of their own data, before engaging in a secure computation protocol to compute $\chi^2$-statistics. For details, we refer the reader to our online short note on how we implemented the competition tasks. On the other hand, benchmarks with floating point operations such as K-Means incur a relatively larger slowdown because modern processors have special floating point instructions which makes it favorable to the insecure baseline.

VIII. CONCLUSION, SUBSEQUENT AND FUTURE WORK

We design ObliVM, a programming framework for automated secure computation. Additional examples can be found at our project website [http://www.oblivm.com], including popular streaming algorithms, graph algorithms, data structures, machine learning algorithms, secure genome analysis [58], etc.

A. Subsequent Works and Adoption of ObliVM

To the best of our knowledge, our framework is already being adopted in other projects. First, the GraphSC work [24] extends our ObliVM-GC framework to support parallel execution of gadgets on modern architectures with inherent parallelism, such as multi-core processor architectures, and compute clusters. Because of ObliVM-GC’s clean architecture, it was not too much work for GraphSC’s parallel extension, which required about 1200 more lines of code on top of ObliVM-GC. Second, our collaborators (and a subset of the authors of this paper) are implementing a MIPS processor over our ObliVM framework. Such a MIPS processor will allow maximum backward compatibility: code written in any language can be compiled to a MIPS processor using a stock compiler, and then evaluated securely. Third, a group of networking researchers have used our ObliVM-GC framework to develop privacy-preserving software-defined networking applications [59]. Fourth, we used our ObliVM framework to participate in the iDash Secure Genome Analysis Competition [58], [60]. Finally, Wagner et al. also use our ObliVM framework to develop privacy-preserving applications on human microbiomes [61].

B. Future Work

In future work, we will implement a C-based Garbled Circuit back end similar to JustGarble [4], such that we can exploit hardware AES-NI features of modern processors. We will also implement the state-of-the-art OT optimizations [8]. It will also be interesting to provide support for multiple parties and malicious security. Since ObliVM is designed to be good at compiling programs to compact circuits, it will be interesting to extend ObliVM to support other cryptographic back ends such as fully homomorphic encryption, program obfuscation and verifiable computation.

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 Srinivas 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


“Partisia,” http://www.partisia.dk/.


J. Brickell and V. Shmatikov, “Privacy-preserving graph algorithms in the semi-honest model,” in ASIACRYPT, 2005.


“Private communication,” 2014.


“human genomeneprivacy.org/2015


APPENDIX A
ADDITIONAL OBLIVIOUS ALGORITHM

A. Additional Oblivious Graph Algorithm

It is has been an open question how to construct an Oblivious Depth First Search (ODFS) algorithm that outperforms one built on generic ORAMs [22] Here we answer this question for dense graphs. We present $O((E + V) \log V)$ time ODFS algorithm. In comparison, a generic-ORAM based oblivious solution would take $O((E+V) \log^2 V)$ time (ignoring possible log-log factors) [27, 67].

The challenge is that in standard DFS, we need 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. To make it oblivious would require placing this bit-array inside an ORAM, thus incurring $O(\log^2 V)$ cost per access, and $O(E \log^2 V)$ time over all $O(E)$ accesses.

To solve this problem, instead of verifying if a vertex has been visited, we maintain a tosca list of vertexes, which preserves the same traversal order as DFS. When new vertexes
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 := \{$(s,0)$, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_\}; // not in ORAM
2. for $i = 1 \rightarrow |V|$ do
   3. $(u, \text{depth}) := \text{tovisit}[1]$;
   4. tovisit[1] := $(u, \infty)$; // mark as visited
   5. order[i] := $u$;
   6. edge := $E[ u ]$;
   7. for $v := 1 \rightarrow |V|$ do
      8. if $\text{edge}[v] \neq \bot$ 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 $v := \bot$;
     12. tovisit.Merge(add);
3. return order

are added to this list, we guarantee that each vertex appears in the list at most once using an oblivious sorting algorithm. Algorithm 3 presents our oblivious DFS algorithm, and defines the inputs, outputs, and how they are stored.

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 $(u, \text{depth})$:

- $(u, \text{depth} = \infty)$: indicates that vertex $u$ has been expanded, and will not be expanded again.
- $(u, \text{depth} \neq \infty)$: indicates that vertex $u$ was reached at depth $\text{depth}$. 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.
- **Visited vertexes will never be expanded.** All entries with a $\infty$ depth come after those with a finite depth.
- **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 $V$.

The merge operation (Line 12). The operation is performed with two oblivious sorts. See Figure 7 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 ($\infty$ comes first). All $\bot$ entries are moved to the end. Then, for all entries with the same vertex number (which are adjacent), we keep only the first entry (which has the largest depth value) while overwriting others with $\bot$.

2) **O-sort and trim.** This sorting will (a) push all $\bot$ entries to the end; (b) push all $\infty$ entries to the end; and (c) sort all remaining entries in descending order of depth. Discard everything but the first $V$ entries.

**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 our new 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 over all iterations of the outer loop. Further, each execution of the inner loop requires circuits of size $O(\log^2 V)$, using latest oblivious data structures [26] and Circuit ORAM [29]. So the overall complexity is $O((V + E) \log^2 V)$. We defer further discussions about minimum spanning tree algorithm to our online full version [39].
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:** $d[s]$: 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)