Obliv-C: A Language for Extensible Data-Oblivious Computation

Samee Zahur
samee@virginia.edu
University of Virginia

David Evans
evans@virginia.edu
University of Virginia

Abstract

Many techniques for secure or private execution depend on executing programs in a *data-oblivious* way, where the same instructions execute independent of the private inputs which are kept in encrypted form throughout the computation. Designers of such computations today must either put substantial effort into constructing a circuit representation of their algorithm, or use a high-level language and lose the opportunity to make important optimizations or experiment with protocol variations. We show how extensibility can be improved by judiciously exposing the nature of data-oblivious computation. We introduce a new language that allows application developers to program secure computations without being experts in cryptography, while enabling programmers to create abstractions such as oblivious RAM and width-limited integers, or even new protocols without needing to modify the compiler. This paper explains the key language features that safely enable such extensibility and describes the simple implementation approach we use to ensure security properties are preserved.

1. Introduction

A protocol for *secure computation* allows two or more parties to collaboratively perform some computation without revealing their own inputs. There are many generic protocols for secure computation, which can perform arbitrary computation on encrypted data [8, 18, 24, 34]. The way these generic protocols work is that the entire computation is first converted into a *data-oblivious* representation, where the control flow of the program does not depend on the secret program inputs in any way. Such a program can be executed on encrypted data without leaking any information about intermediate results, since the control flow is the same for all executions and does not depend on the data.

A common data-oblivious program representation is a Boolean logic circuit: every logic gate (e.g., AND, OR) is specified before the secret inputs are even known. Another popular representation uses addition or multiplication gates that operate directly on finite field elements (instead of just Boolean values). Given a circuit that describes the desired computation, the protocol specifies how to execute the circuit without revealing any inputs or intermediate results.

While many previous languages and frameworks for secure computation have been developed (see Section 7), none are sufficiently expressive to allow programmers to implement even simple library abstractions. The reason is that these languages have been designed to provide traditional programming abstractions that hide the data-oblivious nature of secure computation from the programmer. Our approach provides high-level programming abstractions while exposing the essential data-oblivious nature of such computations.

Motivating Example. Consider this simple C example of a dynamically resized array:

```
DynVec *vec = dynVecNew();
for (i = 0; i < n; i++) {
    if (cond) {
        dynVecAppend(vec,x);
    }
    ...
```

Implementing a library like this for standard computation is trivial. The DynVec object just needs to keep track of the current size of the vector, and resize an internal buffer when more space is needed to complete an operation.

Writing something similar for a data-oblivious computation, requires the compiler to implement an append under an unknown condition: the internal memory buffer must be resized regardless of the now unknown semantic value of cond, whereas the value of x should be appended into that buffer (which is now encrypted) using a conditional write that depends on the value of cond specified outside of the function.

This problem is exacerbated for more complex library abstractions. For example, an ORAM structure that allows random access to a memory bank without revealing anything about the access pattern. On every read or write operation it needs to do things like network transfers, pseudo-random shuffling, and cryptographic operations. Defining a simple oramWrite() function is problematic if we want to allow it to be called from inside a conditional block: the function needs to specify a whole series of operations, some of which need to be done conditionally while others are done unconditionally. Indeed, it is not clear how a traditional programming language could even be adapted to express the situations that commonly arise in data-oblivious computation.

Contributions. We show how a language can be designed to support extensible secure programming introducing control structures that expose the data-oblivious nature of secure computation. To make it easier for programmers to develop and reason about data-oblivious programs, we provide a type system that incorporates oblivious data.

Our Obliv-C language is a strict extension of C that supports all C features (including **struct**, **typedef**, pointers, recursive calls, and indirect function calls), along with new data types and control structures to support data-oblivious programs. Section 2 introduces our language and describes how its language constructs and type system support data-oblivious computation.

We describe the architecture of our Obliv-C compiler in Section 6, showing that our language can be implemented on top of a traditional language and in a way that provides high confidence that security properties of the underlying protocol are preserved.

Obliv-C is designed to enable practitioners to more easily develop scalable secure protocols, and to allow researchers to easily implement and test new features or techniques by simply writing a new libraries rather than having to modify or build a new compiler. To demonstrate how our approach supports exploration at many levels, Section 4 shows how Obliv-C could be used to easily implement various library-based features including range-tracked integers, ORAM, and multi-threading that could not be done with existing languages, and Section 5 shows how Obliv-C supports experimentation with protocols.

2. Obliv-C

Obliv-C is a strict extension of C that provides data-oblivious programming constructs. Next, we provide an overview of the design and philosophy behind the language. Section 2.2 presents a concrete example of an Obliv-C program. We provide details on the type system in Section 3. Our implementation compiles an Oliv-C program into standard C, as described in Section 6.

2.1. Overview

Obliv-C is designed to guarantee that all security properties provided by the underlying protocol are maintained, while exposing aspects of data-oblivious computation to the programmer. Our design emphases safety, guaranteeing that no information can be leaked by program executions (assuming the underlying protocol is secure) while giving programmers enough control (including the ability to circumvent type rules) to do things that would not be possible with other high-level languages.

The main construct we introduce is an oblivious conditional. For example, consider the following statement where x and y are secret data:

```
obliv if (x > y) x = y;
```

Since the truth value of the x > y condition will not be known even at runtime, this code cannot be executed normally. Instead, every assignment inside the if statement will have to use "multiplexer" circuits in much the same way Boolean logic circuits use multiplexers to choose between two different values. We could translate this code into something like:

```
cond = (x > y); // 0 or 1
x = x + cond * (y − x);
```

This removes any explicit control flow dependency on unknown values by using conditional assignments.

Obliv-C extends C in the following ways:

- Every basic data type (e.g., **int**, **char**, etc.) has an **obliv**-qualified counterpart (e.g., **obliv int**, **obliv char**, etc.) which is represented using an encrypted value.
- Every **if** statement with a condition that depends on **obliv**-qualified data is explicitly indicated as **obliv if**. An **obliv if** statement executes in a way that prevents control dependencies from leaking the condition value.
- Type rules related to **obliv if** are enforced across function boundaries at compile time by using two different function families: ones that can be invoked from inside **obliv if**, and ones that cannot.
- Special *unconditional segments* allow library writers to perform actions unconditionally, which allow them to write various library abstractions.

These segments escape the type system, but do not risk any information leak, just the possibility that a program does not mean what the programmer intended.

Next, we walk through a simple example illustrating the general structure of Obliv-C programs and how the programmer uses it.

2.2. Millionaires' Problem

Figure 1 shows an Obliv-C implementation of Yao's classic millionaires' problem [34]. It simply outputs which of two integers is greater (purportedly, to enable two millionaires to decide who should pay for dinner without disclosing their actual wealth).

When the program executes, both parties (in this protocol, although our design can support any number of parties) execute the same program. By convention, we will call them Alice (Party 1) and Bob (Party 2). The a, b, and res variables are declared using the **obliv** keyword to indicate that their values may depend on secret inputs.

The program obtains secret inputs using:

```
obliv int feedOblivInt (int value, int p)
```

This function is executed synchronously by both parties to introduce the input into **obliv int** variables of the shared computation. It converts a value from one of the parties (party p) into a new cryptographic **obliv int** value that can no longer be deciphered by either party on its own. The value provided by the other party is simply ignored. Since both parties have their own copy of each variable each party can use the myinput field to hold their own inputs. Thus, in Figure 1, the first invocation of feedOblivInt() only reads Party 1's copy of myinput into the shared variable a, while the second one reads only from Party 2. These variables can still be manipulated using ordinary C operators, and even mixed with ordinary **int**s in expressions, but the results are all **obliv**-qualified and only accessible as encrypted values.

The only way any values derived from secret data can be converted back to a semantic value is by using a reveal function, such as:

```
void revealOblivInt(int *dest, obliv int src, int p)
```

When this function is invoked by both parties on the same variable src, the value is decrypted and stored into the integer pointed to by dest. If p == 0, all parties receive the result; otherwise p specifies a single party who receives it. This ensures that only the values that both parties agree to reveal are actually revealed by the execution. The underlying protocol ensures that a reveal function only succeeds if both parties provide consistent parameters to the function (e.g., it will fail if they provide different values for src or p).

To run the program, both the files in Figure 1 are compiled with the oblivcc command provided by our tool. It is a simple wrapper that provides a familiar command-line interface. It preprocesses any input file with an ".oc" extension to a plain C file before passing it on to gcc and links with additional runtime libraries required for Obliv-C code. Once compiled, the two parties simply execute the program with appropriate inputs like any other program: the end user does not need to know about Obliv-C or even need to install it separately.

```
typedef struct {
  int myinput;
  bool result;
} ProtocolIO;

void millionaire (void *args);
```

(a) File "million.h"

```
#include <million.h>
#include <obliv.oh>

void millionaire (void *args) {
    ProtocolIO *io = args;
    obliv int a, b;
    obliv bool res = false;
    a = feedOblivInt (io->myinput, 1);
    b = feedOblivInt (io->myinput, 2);
    obliv if (a < b) res = true;
    revealOblivBool (&io->result, res, 0);
}
```

(b) File "million.oc"

```
#include <million.h>

int main (int argc, char *argv[]) {
    ProtocolDesc pd;
    ProtocolIO io;
    int p = (argv[1] == '1' ? 1 : 2);
    sscanf(argv[2], "%d", &io.myinput);
    // ... set up TCP connections

    setCurrentParty (&pd, p);
    execYaoProtocol (&pd, millionaire, &io);
    printf ("Result: %d\n", result);
    // ... cleanup
}
```

(c) File "million.c"

Figure 1: Code for the Millionaires' Problem.
Figure (a) shows the header file that defines the datatype, (b) describes the secure computation in a protocol-neutral manner in Obliv-C and (c) shows code in plain C that invokes the former with a specific protocol with appropriate inputs, outputs and options.

3. Type System

The Obliv-C type system builds from a traditional information-flow based type system [31] with two levels of security. Variables declared using **obliv** are considered sensitive, and the type system ensures that information from these variables never flows into the non-sensitive ones through either explicit data dependencies or implicit control dependencies.

We add several rules beyond standard information-flow to support data-oblivious computation. First, we want programmers to be able to easily estimate the relative computation cost of their code, and to help programmers avoid writing unscalable code. This is why, for instance, we do not allow pointers with **obliv** addresses, or loops directly using **obliv** conditions. Obliv-C provides other means for accomplishing the same goals which make the costs more explicit and controllable.

Second, we account for the fact that control flow is not actually sensitive in our system. Any apparent control dependency indicated by our **obliv if** structures is not really a control dependency since it is implemented by converting it into a data dependency. Statements inside an **obliv if** become conditionally-executed statements that will be executed regardless of whether the controlling condition is true or false, which which have no semantic effect when the condition is false. Control flow is always public information in our system. This is what ultimately allows us to define features such as unconditional segments, which are very useful in writing libraries.

The purpose of our type rules is different from the normal purpose of information-flow type systems. The security of the **obliv** values is enforced at runtime by cryptographic means: even inspecting memory dumps or network logs should not provide any useful information. Hence, our type system is not used for preventing information leaks, it only exists to help the programmers avoid mistakes by providing compile time errors for code that would cause runtime errors or meaningless results. For example, this is legal Obliv-C code:

```
obliv int x; ...; int y = *((int *) &x);
```

Although our compiler will allow casts like this, the resulting code will not leak any information. At runtime, y will just contain gibberish bits of ciphertext. Obviously we do not recommend writing code like this, but it will not leak any information about x. The only way to reveal values is through the proper use of reveal family of functions on mutually agreed upon values.

In true C fashion, we allow programmers to shoot themselves in the foot, but provide a type system to help programmers avoid doing this accidentally.

3.1. Oblivious Data

The first four type rules explain how oblivious data is declared and used in programs.

Rule 1: Only basic C types (such as **obliv int**, **obliv char**, etc.) can be **obliv**-qualified. An **obliv**-qualified type represents a variable whose value may be unknown at runtime.
This excludes types such as structures, and pointers, although we do support structures with **obliv** fields or pointers to **obliv** variables. (Functions may be qualified with **obliv**, although it has a somewhat different purpose that we will discuss in Section 3.3.)

The following two rules provide a flow-sensitive type system that prevents sensitive data flowing into non-**obliv** variables:

Rule 2: Any expression that combines **obliv** values and non-**obliv** values results in an **obliv** value.

Rule 3: Non-**obliv** variables cannot be assigned to **obliv** values. Non-**obliv** values can be implicitly converted to **obliv** values and assigned to **obliv** variables.
The next rule limits where **obliv** values can be used, primarily to encourage programmers to avoid surprisingly expensive operations:

Rule 4: An **obliv** value may not be used as an array index, offset in pointer arithmetic, or as a shift amount in a bitwise shift expression. All other operators can freely mix both types of operands.
Note that we do allow **int**s to index into arrays of **obliv int**s, but not vice versa. Although we could have avoided Rule 4 and added support for oblivious array indexes using circuits such as full multiplexers, but they are notoriously slow in practice. Instead, we want to encourage developers to explicitly weigh the trade-offs between various other mechanisms of indirect access, such as those using circuit structures [35] or oblivious RAM (Section 4.2), all of which can be implemented as library modules in Obliv-C. Similarly, it is a deliberate decision to not support pointers whose addresses can be unknown at runtime. Such pointers would make it very easy to write inefficient programs that would need to multiplex over the entire heap at every pointer dereference.

3.2. Conditional Constructs

Rule 5 ensures that control flow never depends on **obliv** values, except as used in the new **obliv if** construct:

Rule 5: A condition expression of a traditional control structure (e.g. **while**, **for**, **switch**, etc.) may not be **obliv**. An **if** statement using **obliv** values must be explicitly marked as **obliv if**.
The **obliv if** statement has the following syntax:

```
obliv if (cond) { ... } [else { ... }]
```

Marking **obliv if** explicitly helps the programmer (and code readers), since it has implications both in the type system and in the runtime. Since the condition may not be known at runtime, *both* the consequent and alternative branches will be executed (possibly using conditional instructions) no matter what the condition actually was. As a result, execution always incurs the runtime overhead of both branches.

An **obliv if** statement introduces an **obliv** *context*, where certain operations are restricted. Non-**obliv** variables declared outside an **obliv** context cannot be modified inside it. Locally declared non-**obliv** variables, however, can be modified since they are not visible outside the **obliv** context. This allows us to run loops inside **obliv if** constructs:

```
obliv if (cond) {
    for (int i = 0; i < n; ++i) {
        // ...
    }
}
```

Without this exception for locally declared variables, we would not be able to modify i for the loop counter. But here, this is not a problem since i will go out of scope once we exit the conditional branch. Thus, this exception for locally declared variables does not violate the requirements for data obliviousness.

As we explain in Section 3.3, this also allows us to safely invoke functions from inside an **obliv if** even if they modify some non-**obliv** variables. Our rules for preventing such control dependencies are slightly complex since we want them to work across function boundaries, without actually inlining functions.

The restriction on oblivious values in conditional expressions for other control structures appears draconian, but is consistent with our goals to provide programmers with a clear view of the costs of different programming constructs. The amount of computational resources used by a program, such as CPU time or memory usage, would leak information about the loop condition if the number of executions varies. Hence, loop conditions in secure programs must not depend on secret values. Instead, a data-oblivious program needs to impose a predetermined conservative upper limit to the number of iterations, and iterate that many times regardless of the condition. Within the loop body, we can use an **obliv if** statement to limit the effective number of iterations. For example, if n is an **obliv** variable, the loop:

```
for (i = 0; i < n; i++) { ... }
```

could be rewritten as:

```
for (i = 0; i < MAX_BOUND; i++) {
    obliv if (i < n) { ... }
}
```

In practice, the restriction on oblivious values in loop conditions is necessary, because whatever a loop condition is, the parties executing it will have to somehow know when to terminate the loop. Which means, it can always be written in a way such that the condition is a non-**obliv** value.

3.3. Functions

Not all functions can be allowed inside **obliv if**, since they may modify non-**obliv** global variables. To handle this, we introduce a second family of functions called **obliv** functions. These functions can be invoked from anywhere, but may not modify global non-**obliv** variables or invoke other non-**obliv** functions.

Here is an example of an **obliv** function:

```
void writeArray (obliv int* arr, int size,
                 obliv int index, obliv int value) obliv {
    for (int i = 0; i < size; ++i) {
        obliv if (i == index) {
            arr[i] = value;
        }
    }
}
```

The **obliv** suffix after the parameters denotes that writeArray is an can be called from inside a conditional context. The compiler checks the body of an **obliv** function indeed adheres to the restrictions on modifying global state.

As for writing to arrays at an **obliv** index, note that we cannot do much better than this in general. The standard practice is to create a linear-sized multiplexer circuit to perform the write, which is essentially what writeArray does. Each assignment inside the **obliv if** is a conditional assignment (i.e., a multiplexer between old and new values), which is controlled by a different condition for each value of i.

The type rules for **obliv** functions are:

Rule 6: Non-**obliv** functions may not be invoked from inside **obliv if** or other **obliv** functions.

Rule 7: Inside **obliv** functions, all non-**obliv** global variables are **frozen**. Moreover, they may not invoke other non-**obliv** functions.

3.4. Frozen State

The **frozen** qualifier allows us to safely pass variables by reference and store them in structures, as well as to reason about **obliv if** contexts more precisely.

A **frozen** variable is similar to a **const**-qualified variable. The **frozen** qualifier follows the same rules for type propagation and conversion as **const** in C. This includes the fact that a **frozen**-qualified L-value cannot be modified, as expected. In addition to the standard C rules for **const**, the meaning of **frozen** is defined by the following four rules:

Rule 8: All non-**obliv** variables defined outside an **obliv if** become **frozen**-qualified inside it (as well as in the body of the associated **else** clause). Freezing an already **frozen** variable has no effect.

Rule 9: Similarly, all non-**obliv** global variables defined outside an **obliv** function become **frozen** in the body of the function.

Rule 10: Dereferencing any pointer of type $T *$ **frozen**, for any type T, produces an L-value of type T **frozen**.

Rule 11: On **obliv** data, **frozen** qualifiers are ignored.

The reason we had to introduce a new qualifier (along with Rule 10) instead of just reusing **const** is that we frequently need to handle situations like this:

```
struct Value { int *p; } v;

obliv if (cond) { // v is frozen inside conditional context
    v−>p = 5; // error
}
```

Here, if we used **const** instead, f−>p would have been of type **int** $*$ **const**, which freezes only the pointer, not the referenced value. This is not what we want, since we need all variables reachable through pointers declared outside the conditional context to be **frozen**.

3.5. Unconditional Blocks

Obliv-C provides a way to escape the normal type rules by using an unconditional block:

∼**obliv**(*varname*) { ... }

This is only meaningful inside an **obliv if** or an **obliv** function, where code is running in a conditional context controlled by some oblivious condition. That condition is assigned to a new **obliv bool** variable named *varname*.

Code within an unconditional block may modify frozen variables:

Rule 12: All **frozen** qualifiers are ignored directly in the scope of an unconditional segment.

```
typedef struct {
    obliv int* arr;
    obliv int sz;
    int maxsz;
} Resizeable;

void writeArray (Resizeable *r, obliv int index,
                 obliv int val) obliv;

// obliv function, may be called from inside obliv if
void append (Resizable *r, obliv int val) obliv {
    ∼obliv(_c) { // condition unused here
        r→arr = reallocateMem (r→arr, r→maxsz + 1);
        r→maxsz++;
    }
    writeArray (r, r→sz, val);
    r→sz++;
}
```

Figure 2: Example use of an unconditional block.

Code inside an unconditional block is executed unconditionally. Note that this does not risk any information leak, however, since the code in the unconditional block *always executes*, regardless of the value of the oblivious condition that would normally control its execution.

An example of its use is shown in Figure 2, which shows part of the implementation of a simple resizable array. It is implemented as a **struct** as shown at the top of the figure. While the current length of the array is unknown (since we might append() while inside an **obliv if**), we can still use an unconditional block to track a conservative upper bound of the length. We use this variable to allocate memory space for an extra element when it might be needed.

4. Extensible Data-Oblivious Programming

This section presents several examples of how the Obliv-C system supports extensible programming for data-oblivious computation. They highlight how having access to the full C language and libraries allows an Obliv-C programmer to add features to Obliv-C that would not possible in any other framework.

The first two show ways data structures can be implemented in Obliv-C that enable performance improvements that could not be done without explosing data-oblivious computation to the programmer: range-tracked integers and oblivious RAM. The next shows how programmers can incorporate special techniques into Obliv-C programs, in this case taking advantage of secret random numbers. Finally, we show how POSIX

```
for (i = 1; i <= n1; ++i) {
  for (j = 1; j <= n2; ++j) {
      obliv int temp = omin(dp[i][j−1], dp[i−1][j]);
      obliv int d = 1;

      obliv if (temp >= dp[i−1][j−1]) {
          temp = dp[i−1][j−1];
          d = (s1[i−1] != s2[j−1]);
      }
      dp[i][j] = temp + d;
  }
}
```

```
for (i = 1; i <= n1; ++i) {
   for(j = 1; j <= n2; ++j) {
      Accum temp = acMin (dp[i][j−1], dp[i−1][j]);
      obliv bool d = true;

      obliv if (acLessEq(dp[i−1][j−1], temp)) {
         acCopy(&temp, &dp[i−1][j−1]);
         d = (s1[i−1] != s2[j−1]);
      }
      dp[i][j] = acAdd(temp, acFromBoundedOInt (0, 1, d));
   }
}
```

Figure 3: Computing edit distance with ordinary integers, vs. range-tracked integers

threads can be integrated into Obliv-C to produce protocols with multithreading support, demonstrating some of the advantages of seamless integration with standard C.

4.1. Range-Tracked Integers

Programs often do not need full 32-bit wide integers for all their variables, so it is possible to make arithmetic operations cheaper by using integers of limited bit-width. This can achieve significant speedups for applications that use lots of small integers, for example when counting or accumulating values. Here, we show how to write a library to support range-tracked integers that automatically maintain a conservative upper bound for a value, and resize their bit-widths accordingly.

Figure 3 shows an example of how it may be used. The example we use is that of computing edit distance between two strings. If the strings are of length n, we know that the results can never exceed n, and can then use appropriate widths for each integer. As shown in Table 1, range-tracking integers can lead to significant performance improvements.

To implement this abstraction, we define the type accum as a **struct** with fields for maintaining the actual value (which is oblivious, so not semantically known) and the conservatively-estimated maximum value:

```
typedef struct {
    obliv unsigned value;
    unsigned maxValue;
} accum;
```

Note that maxValue is not **obliv**-qualified — it is a publicly known upper bound that depends on the program the two parties are executing, not on their private values. It must be public so both parties may calculate the width of the circuits needed for each operation.[1]

Here is an example how a function operating on a range-tracked accumulator could be used:

```
accum x = ...;
obliv if (y > 0) { accumAddInt(&x, 1); ... }
```

Since we expect accumAddInt to be used inside **obliv** scopes, we need to make this function an **obliv** function. Moreover, we will not know, even at run-time, if the condition y > 0 was actually satisfied. To hide the condition, the protocol will require executing accumAddInt() regardless of the condition.

While the implementation can conditionally modify the oblivious value, x.value, the value of x.maxValue must be conservatively adjusted regardless of the (unknown) condition. In other words, it is publicly known that the value might have increased, and so the upper bound has to increase accordingly.

Here is the implementation of accumAddInt:

```
void accumAddInt (accum ∗dest, int x) obliv
{
    ∼obliv(en) { dest→maxValue += x; }
    int mask = (1<<width(dest→maxValue)) − 1;
    dest→value = (dest→value + x) & mask;
}
```

We use an unconditional segment to unconditionally modify the upper bound. When the actual addition is performed, we mask out the higher-order bits beyond the current maximum size to zero. This clears out any ciphertext produced for the higher order bits by the carry-out bits of addition, and allows simple bit-level constant propagation to emit fewer gates during the later arithmetic operations. Functions for min, max, addition, and copying are implemented similarly.

At this point the reader might wonder why we are implementing something so simple in a library rather

1. In our implementation we also have a similar field for tracking the lower bound, but we omit that here to simplify our discussion and assume that the lower bound is always zero.

than having it as built-in optimizations. Indeed, while we might add such optimizations to the compiler in the future, this example demonstrates that the programmer can go ahead and implement such optimizations as a high-level library without needing to modify the compiler. Further, even if range-tracking integers were provided by the compiler, there will always be special cases where the compiler will not be able to detect opportunities for optimization that are apparent to a programmer with understanding of deeper properties of the application. Compiler optimizations are not powerful enough to substitute for enhanced language expressiveness and control.

4.2. Oblivious RAM

While the previous section demonstrated an abstraction that works with any bit-level protocol for secure computation, this section presents a more complex, but protocol-specific abstraction. A programmer who is willing to write an application in a way that is not protocol-agnostic can use Obliv-C to take advantage of specific functionalities available in the protocol of his choice without needing to modify the compiler.

Specifically, we show how a library can add ORAM functionalities to Yao's garbled circuits designed for semi-honest adversaries. Implementations of such hybrid ORAM-based protocols were first described by Gordon et al. [9].

The purpose of ORAM is to avoid the linear-time cost associated with naïve array lookups when the index depends on unknown data, and is therefore **obliv**-qualified. There are many constructions of ORAM [29, 30], and Gordon et al. [9] describe how it can be integrated into secure computation, so we will not go into details here. The way it works is that a single access to one location gets converted into multiple accesses at pseudo-random locations. These pseudo-random locations are then revealed to one (or both) parties, so that the corresponding data blocks can be read from some encrypted store and fed into the secure protocol. Once inside the garbled circuit, the data gets decrypted, used, possibly shuffled, re-encrypted, and then written back. The encryption is randomized, so the same plaintext may have many different ciphertexts. The decrypted information never leaves the garbled circuit protocol, and the logical locations are always hidden. As a result, neither party is aware of which location is being accessed and when.

Library interface. We implemented the Path ORAM [30] protocol and a "naïve ORAM" that uses the linear-sized circuit. Very recently, SCORAM [32] was developed specifically to be efficient in secure computation. While we have not yet implemented that, they do not have any fundamental difference and should be implementable in Obliv-C just as easily. We need to be able to support read/write operations while in an **obliv** context, or else it will not be a drop-in replacement for the naïve array operations. So our API defines the read/write functions with the following types:

```
void oramRead (obliv bool∗ dest,
               Oram∗ oram, obliv int ind) obliv;

void oramWrite (Oram∗ oram, obliv int ind,
                const obliv bool∗ src) obliv;
```

Reads and writes always happen through blocks of pre-specified sizes, which also determines the size of **obliv bool** arrays dest and src. The **obliv** keyword at the end of the prototype indicates that they may be invoked inside **obliv if** blocks (either directly or through other functions). The way we have architected this library is that both kinds of ORAM implement the exact same interface, so we can perform reads and writes using the same functions. As far as user code is concerned, the only difference is in their initialization:

```
Oram∗ naiveOramNew(int eltsize, int eltcount) obliv;
Oram∗ pathOramNew(int eltsize, int eltcount) obliv;
```

Both return the same type, so there is no need to change any other code to switch between ORAM implementations. The way we implemented this takes advantage of indirect function calls:

```
struct Oram {
  int eltSize, eltCount;
```

	100 x 100 characters			200 x 200 characters		
	Normal int	Range-tracked	Improvement	Normal int	Range-tracked	Improvement
Total time	7.28 s	4.28 s	41.2%	23.19 s	12.04 s	48.08%
OT time	1.95 s	1.88 s	—	1.98 s	1.94 s	—
Gate execution time	5.33 s	2.40 s	55.0%	21.21 s	10.10 s	52.4%
Number of gates	1,669,010	668,429	60.0%	6,678,412	2,835,763	57.5%

Table 1: Improvements obtained from integer range-tracking in edit distance calculation

```
  void (*read)(obliv bool *,Oram *, obliv int) obliv;
  void (*write)(Oram *, obliv int, const obliv bool *) obliv;
  void (*cleanup)(Oram *) obliv;
  void *extra;
};
```

Each type of ORAM sets these runtime hooks during initialization, keeping any construction-specific data in a structure pointed to by extra. Notice how the pointer types reflect the fact that they point to **obliv**-functions. This way, the compiler knows that calling them requires passing a hidden condition variable, and that it can be safely invoked from a conditional scope.

Naïve ORAM. While the implementation of a naïve ORAM is straightforward and inefficient, we use this opportunity to demonstrate how Obliv-C makes it easy to write a first prototype, while at the same time provides enough flexibility for the programmer to optimize heavily used functions.

Recall that Obliv-C does not allow **obliv** types to be directly used for indexing into an array, since the best way to do so depends on the application. So, to perform a write, we do the same thing circuit-based logic does: linearly scan every single element and update just the specified element. Here is the code for naiveOramWrite:

```
void naiveOramWrite(Oram *oram, obliv int ind,
                    const obliv bool *src) obliv
{
  obliv bool *extra = oram->extra->storage;

  for (int i = 0; i < oram->eltCount; ++i) {
    obliv if (i == ind) {
      copyBools(storage + i * oram, src,
                oram->eltSize);
    }
  }
}
```

We can improve this a little by using a decoder logic instead of doing full comparison at every index:

```
~obliv(en) {
  obliv bool *flags = calloc (oram->eltCount,
                              sizeof (obliv bool));
  decoder (flags, en, ind, oram->eltCount);
  for (int i = 0; i < oram->eltCount; ++i) {
    obliv if (flags[i]) {
      copyBools (storage + i * oram, src,
                 oram->eltSize);
    }
  }
  free(flags);
}
```

What the function decoder() does is that it fills flags with all false values, except possibly a single true value at position ind. Even that value is set to false if it is not enabled with the input en set to true. This reduces the number of gates used in comparison from $n*width$ to just $n-1$.

Finally, even though the index ind may be unknown, it does not need to be. Often, through simple constant propagation, the program ends up invoking a write on a publicly known index at runtime, even though its compile-time type is **obliv int**. In such cases, the full loop is unnecessary, and we can just branch into a faster path:

```
if (isKnownInt(&i, index)) {
  copyBools(storage + i * oram, src, oram->eltSize);
} else {
  ~obliv(en) {
    obliv bool *flags = ...
```

The function isKnownInt() is provided by Obliv-C. The way it works is that, if the value of the second parameter is known publicly, isKnownInt() returns true and copies the known value into the first parameter as an ordinary integer. But if index is unknown (i.e., depends on any secret values), isKnownInt() returns false, and i is left unchanged.

This provides users with a robust, simple, and high-level interface for performing writes, while the library writer can use Obliv-C to still perform low-level circuit optimizations.

Path ORAM. Path ORAM is an efficient oblivious RAM design introduced by Stefanov et al. [30]. We also use the techniques from Dov Gordon et al. [9] to integrate it with Yao's protocol. At this point, it should be clear how the design Obliv-C supports implementing an ORAM library by allowing functions to execute in conditional contexts. For example, we can write programs such as:

```
obliv if (cond) {
  oramWrite(oram,index,value);
}
```

This executes correctly even though the value of cond will be unknown at runtime. Internally, the function performs network transfers, pseudo-random shuffling, and extra cryptographic operations unconditionally, while the actual write is performed conditionally.

In implementing Path ORAM, it was particularly useful to be able to use any existing C library functions for networking and cryptography, something not possible in other languages for secure computation. The best part is, the user of the function is still completely oblivious to all of this: all the user needs to know is that it allows random access in polylogarithmic time. A programmer can use such functions without any cryptographic background.

Since there is no need to modify the compiler to change the ORAM design, this system will be useful

```
obliv unsigned ocRandomOblivInt(void)
{
  obliv unsigned res = 0;
  int p, pc = ocCurrentProto()−>partyCount;
  unsigned x;

  gcry_randomize(&x, sizeof(x),
                 GCRY_STRONG_RANDOM);
  for (p = 1; p <= pc; ++p) {
     res ^= feedOblivInt(x,p);
  }
  return res;
}
```

Figure 4: Generating secret random integers.

for researchers experimenting with their own custom ORAM constructions or other special-purpose sub-protocols.

4.3. Generating Secret Randomness

Generating randomness is very common operation in cryptographic protocols. There are well known examples [3] of how being able to generate secret random numbers (unknown to any party) can lead to significantly faster computation. In this section we describe how we can generate such randomness in Obliv-C and can be used as an optimization strategy.

Figure 4 shows a possible implementation for generating random integers. It just XORs random inputs from all parties, but does not reveal the result.

One example of its usefulness is the computation of modular inverses modulo a publicly known prime number, common in cryptography. Ordinarily, computing modular inverses require the extended Euclid's algorithm, which involves $\Theta(n)$ divisions and multiplications do be done securely in a circuit for n-bit numbers.

A faster approach would use secret randomness (similar to the techniques by Damgård et al. [3]). To compute $a^{-1} \bmod p$, we first generate a secret random number r. We then securely compute $ar \bmod p$ and reveal it to everyone. Masking by a secret randomness prevents any semantic information leak.

The parties can then locally compute $x = (ar)^{-1} \bmod p$, and use another secure multiplication obtain $rx = r(ar)^{-1} = a^{-1}$. Thus, we obtain the modular inverse by using just two secure multiplications and inexpensive local computation. Similar techniques can also be used to find inverses of matrices and group elements.

We ran some experiments with 32-bit integers, and found that this technique reduces runtime for inverse computation in semi-honest Yao protocols for 100 integers from 24.7 s to just 9.1 s.

This provides another demonstration of how simple Obliv-C library functions can allow users to easily write their own primitives that work seamlessly work with the rest of the language. No existing framework that provides a high-level language allows programmers to invent such primitives and perform optimizations.

Compatibility. This function would work in any protocol any protocol that supports input/output in the middle of a running protocol (e.g., semi-honest Yao as done here). However, other protocols such as the dual-execution version of Yao will not support this because it requires all outputs to be revealed at the very end (or else it risks leaking one bit of private inputs for each round of output).

4.4. Multithreading

Despite the prevalence of multicore processors today, no existing secure computation frameworks provide full multithreading support.[2] The reason is simply that full support requires a fairly extensive library for managing threads and providing synchronization primitives. Instead, our Obliv-C design enables users to take advantage of existing C libraries. Compared to ordinary computation, however, for threading to provide useful parallelism, two-party protocols need coordination between threads of both parties.

We implemented some threading support library to help us write the dual-execution protocol (Section 5.2), but we did not implement a full thread-enabled Yao yet (i.e., we have not yet implemented a user-exposed `thread_create()` function that can be launched during a protocol).

Implementing a protocol using multiple threads requires paying attention to three important properties, discussed below.

Network Channels. We need to set up separate TCP connections to avoid interference between data transfers for gates executing in different threads. We implemented a simple `newsock=sockSplit(oldsock)` function that creates a new TCP socket between parties that are already connected by an old socket. In particular, the server starts listening to a new unused port, sends the port number to the client using the old socket, after which the client connects. At this point, we can

2. There are many implementations of multiparty computation protocols that do use multithreading for executing various protocol stages [7, 12], but none of these allow application programmers to take advantage of multiple threads at the application level.

```
void obliv_mutex_lock(pthread_mutex_t* m) {
  if (ocCurrentParty() != 1) {
     recvDummy(1);
  } else {
     pthread_mutex_lock(m);
     for (int i = 2; i <= partyCount; ++i) sendDummy(i);
  }
}
```

Figure 5: Mutex implementation

use POSIX functions to create new threads and have each thread use a different socket so that they do not interfere.

Nonces. Any gate-specific nonce value must be carefully chosen to avoid duplicates across threads. In case of Yao's protocol, this is just the gate-specific "tweak" value, or serial number used in garbling. So, for instance, if we have two threads, we should make sure that one thread is only using even numbers while the other is using odd numbers, so that they do not accidentally use the same tweak and compromise security.

Synchronization. The final point is just a general concern for all multi-threaded programs, although we should take care to use synchronization that works in a distributed fashion. While there are many synchronization primitives that are useful in programs, we just discuss mutexes as an example of how they can be wrapped for our protocols. The challenge here is to make sure that the same thread wins the lock on all relevant parties (there could be more than two in some protocols).

Figure 5 shows one way to implement the mutex locking function. The idea here is that only one party keeps an actual mutex, while others wait on a network signal to know that it is safe to proceed. This way, only the thread that wins the lock for party 1 will actually proceed. The unlock function simply calls pthread_mutex_unlock() for party 1, and does nothing for other parties. Note that this is probably not the most efficient way to implementat a mutex. If thread i is running ahead in party 1, it will win even though other parties are still catching up. It is possible that in the meantime, some other thread became ready for all parties, and could have executed. Our proposed implementation does not take this into account, although it is possible to fix that by using another round of communication.

```
void execDebugProtocol (ProtocolDesc *pd,
                        protocol_run start, void *arg)
{
  pd->currentParty = ocCurrentPartyDefault;
  pd->feedOblivInputs = dbgProtoFeedOblivInputs;
  pd->revealOblivBits = dbgProtoRevealOblivBits;
  pd->setBitAnd = dbgProtoSetBitAnd;
  pd->setBitOr = dbgProtoSetBitOr;
  pd->setBitXor = dbgProtoSetBitXor;
  pd->setBitNot = dbgProtoSetBitNot;
  pd->flipBit = dbgProtoFlipBit;
  pd->partyCount= 2;
  currentProto = pd;
  start(arg);
}
```

Figure 6: Implementation of the debug protocol

5. Implementing Protocols

So far we have focused on using Obliv-C with Yao's garbled circuits protocol for semi-honest adversaries. However, Obliv-C is designed to enable easy experimentation with any protocol that operates on individual bits for most of the computation (although other types may also be used for specific parts). This section presents two simple examples to illustrate how Obliv-C can be used to execute different protocols. Beyond these examples, there are many other protocols that could be implemented as functions for use with Obliv-C. This includes the cut-and-choose based protocols [19, 28], those in the LEGO family [6, 25], as well as those not using garbled circuits such as NNOB [24], Sharemind [2], and those based on the SPDZ family [4] (either as a full protocol restricted to Boolean gates, or as a sub-protocol for parts with many arithmetic operations). We have not yet implemented these other protocols for Obliv-C, but all of them execute in ways that fit well with our design.

5.1. Debugging Applications

The easiest way to discuss adding new protocols is to discuss one that performs no cryptography at all. All it does is that it provides a new function execDebugProtocol() which replaces the usual execYaoProtocol(). It simply executes the Obliv-C computation in plaintext. This speeds up the execution and makes it easier to debug Obliv-C programs. No further changes in code are necessary. After testing the program using execDebugProtocol(), we can just change that one line to execYaoProtocol() (or any other protocol launcher) to make it a secure computation.

```
void dbgProtoSetBitAnd(ProtocolDesc* pd,
    OblivBit* dest,const OblivBit* a,const OblivBit* b)
{
  dest−>value = (a−>value && b−>value);
}
```

Figure 7: Debugging protocol callback for an AND gate.

It is easy to write new execProtocol() functions like this for launching custom protocols for use with Obliv-C. Implementing a new protocol is just a matter or defining functions for various protocol-level runtime hooks that we provide. These hooks are called do input, output, and compute a single Boolean logic gate. They simply call the user-provided Obliv-C callback function. We have already defined the various operations in terms of Boolean logic gates, so to implement a new protocol we just need to provide new implementation of these operations.

For example, Figure 6 shows the implementation for execDebugProtocol(). All of the first eight lines are simply setting callback functions that define various aspects of the protocol. Figure 7 shows how one of these callbacks could be implemented (our own implementation also keeps track of stats such as gate count etc.). OblivBit is just a C struct that represents a single **obliv bool** value. For secure computation protocols, this function would also perform other initializations like setting up pseudo-random seeds and executing base OTs. After all the initializations, the last line simply invokes the Obliv-C function provided by the user as a parameter.

We also allow developers switch out TCP/IP with their own custom transport mechanism. For example, in our experience, we often did not want to have to worry about networking issues when writing code, especially when writing a new protocol. So, when running both parties locally on the same machine, we would just pipe the data through standard input and output. In fact, even when running over a network, we can just pipe over SSH. To support this, we also provide hooks for the primitive send() and recv() functions used by various protocols, which can be replaced with arbitrary functions. This could also be used to easily inspect the network traffic for debugging purposes or to package transmissions to improve efficiency.

Note that implementing the new protocols did not require any changes to the Obliv-C compiler. In fact, the compiler does not even need to know which protocol we are planning to execute: that can be determined later at runtime in the main() function written in C. This design makes is very easy to conduct experiments that run the same benchmark with different protocols.

5.2. Dual Execution Protocol

Another protocol we have implemented for Obliv-C is the *dual execution* variant of Yao's protocol [21, 33]. It provides stronger security in that it allows at most one bit of private data to be leaked to a malicious adversary, but requires twice the total computation since the base Yao's protocol is executed twice. Although there are even stronger protocols that provide complete privacy against malicious adversaries [8, 14, 17, 19, 24, 25, 28], they all require substantially more expensive techniques.

The basic idea for dual execution is to execute a secure computation by running Yao's garbled circuits protocol twice, but having the parties swap roles for the two executions which are run simultaneously. This way, each party gets to be the circuit generator for one execution and the evaluator for the other one. The results of the executions are tested for equality to ensure that both circuits computed the same result.

Changes to the application code needed to use dual execution are minimal. It is only necessary to swap out execYaoProtocol() with execDualexProtocol(), and have two TCP connections instead of just one, for which we provide convenient wrappers (this enables dual execution to use separate threads for circuit generation and execution that proceed in parallel).

This new function execDualexProtocol() works the same way as before, but this time it starts two threads before registering protocol-level hooks. It can now perform additional tasks like swapping roles for one thread and configuring each threat to use different TCP connections. The Obliv-C code to be executed is now launched once from each thread until it is time to perform output. During output, it needs to make sure that the output is only revealed to the evaluator side of each thread. At the same time, it accumulates a hash of the garbled wire labels, joins the two threads, performs an equality check, and returns an error to the user if the check failed.

Ideally, we want all application code to be portable across protocols. In reality, however, protocols often involve some quirks and users will have to write code carefully to achieve portability. Every protocol is expected to document its rules of usage. For example, some features like ORAMs are protocol-specific, and will not be supported in dual execution protocols. On the other hand, purely circuit-based optimizations such as integer range-tracking (Section 4.1) can be used with any protocol.

Other rules involve input/output timing and thread-safety. Since dual execution uses two threads, care needs to be taken when using shared memory. Dual execution has a simple restriction: the computation needs to strictly follow the "input, then compute, then output" execution model. For a semi-honest protocol, it is perfectly acceptable to reveal outputs or feed additional inputs in the middle of the protocol, interacting with the protocol as it runs. This is not supported in the stronger protocol: in general, if we want a party to obtain an output, process it locally, and then feed it back, it is quite hard to ascertain if the data was tampered with. In theory, one could do zero knowledge proofs, but it usually is easier (and faster) to just execute the whole computation inside the secure computation protocol. Moreover, the possibility of early outputs opens the door for leaking additional information through selective failure attacks. This is a general theme for all protocols against stronger adversaries, not specific to Obliv-C, but an example of the kind of protocol-specific issue that must be adhered to when implementing applications with Obliv-C.

6. Implementation

The Obliv-C compiler is implemented as a modified version of CIL [23], which transforms Obliv-C code to plain C. Our source code is available under an open source license at <*http://oblivc.org/*>.

We make some changes to the CIL front-end parser to support the new language keywords and control structures. Some additional changes also were made to keep track of additional information such as the lexical depth at which a variable was declared (the default version of CIL discards this information order to simplify internal representation and processing).

Once the type-checker has completed successfully, code generation is straightforward. Figure 8 shows a simple example. An internal header file, "obliv_bits.h" is automatically included in the generated output files which provides the function prototypes and type declarations for the auto-generated function calls will be available during the later stages of compilation. The generated files can then be compiled normally by a standard C compiler (our oblivcc wrapper uses gcc for this).

Because of the way we implemented Obliv-C as a preprocessor on top of C, all of the normal C constructs are still available including structures, pointers, and indirect function calls. We also can trivially support separate compilation—two separate files can be independently transformed and then compiled and linked as usual. This allows us to have a feature-rich language without having to design the whole development tool chain from scratch.

Implementing obliv types. The code generator replaces **obliv** types with corresponding types that are defined as C **struct**s that represent the ciphertext for data bits, the operators get replaced with corresponding function calls. For example, the **obliv int** type is replaced with obliv_c_int which is defined as:

```
typedef struct { OblivBit bits[32]; } obliv_c_int;
```

Operations involving **obliv** types are replaced with corresponding function calls implemented by the provided library. For example, c = a + b is transformed into obliv_c_setAdd(&c, &a, &b).

Functions like obliv_c_setAdd() obliv_c_setLessThan() are defined in a runtime library that is linked with the generated C files. These functions are all defined in terms of bit operations (e.g., AND, OR, NOT). The bit operations, in turn, are implemented in some protocol-specific way, which means these back-end functions are usually written in plain C. To change the protocol, all we need to do is provide new implementations of these operations (Section 5 presents an example).

Transforming conditional code. Code generation is done differently inside an **obliv if** or **obliv** function, since all assignments now must be done conditionally. To ensure that uninitialized garbage values to not interfere with conditional assignments, all local **obliv** variables are initialized to zero.

Nested **if** conditions are handled by AND-ing the new condition with the current, enclosing one. Whenever an **obliv** function is called, the current condition simply gets passed in as a hidden parameter, so that the function can continue to perform proper conditional assignments. When an **obliv** function is called outside of any **obliv** scope (that is, not under the control of any condition), the hidden parameter is just set to true, effectively making it unconditional. This is why **obliv** functions and non-**obliv** functions have different signatures in our language: internally, they accept different parameters. Similarly, Obliv-C supports two flavors of function pointers corresponding to these two flavors of functions. Thus, this transformation eventually removes all control dependencies related to **obliv if** structures.

None of these transformations interfere with the usual control structures of C (**if**, **for**, **while**, etc.). All behave as expected without any transformation. For example,

```
obliv if (cond) writeArray (arr, size, index);
```

is compiled to:

```
void millionaire (void *args) {
    ProtocolIO *io = args;
    obliv int a, b;
    obliv bool res = false;

    a = feedOblivInt(io->myinput, 1);
    b = feedOblivInt(io->myinput, 2);

    obliv if (a < b) res = true;

    revealOblivBool(&io->result, res, 0);
}
```

(a)

```
void millionaire (void *args) {
    ProtocolIO *io = args;
    obliv_c_int a, b;
    obliv_c_bool res;
    memset (&a, 0, sizeof(obliv_c_int));
    memset (&b, 0, sizeof(obliv_c_int));
    memset (&res, 0, sizeof(obliv_c_bool));

    a = feedOblivInt(io->myinput, 1);
    b = feedOblivInt(io->myinput, 2);

    obliv_c_bool cond;
    obliv_c_setLessThan(&cond, &a, &b);
    obliv_c_condAssign(&cond, &res, &obliv_c_true);

    revealOblivBool(&io->result, res, 0);
}
```

(b)

Figure 8: Obliv-C code for the millionaires' problem, before and after it is transformed to plain C by our compiler (reformatted for readability).

```
  writeArray (cond, arr, size, index);
```

Something more complicated like:

```
obliv if (x < y) {
    for (int i = 0; i < n; ++i) {
        if (i % 2 == 0) {
            a[i] = b[i];
        }
    }
}
```

compiles to:

```
obliv_c_setLessThan (&cond, &x, &y);

for (int i = 0; i < n; ++i) {
    if (i % 2 == 0) {
        obliv_c_condAssign (&cond, &a[i], &b[i]);
    }
}
```

Note that the conditional assignment is needed only for **obliv** variables and ++i did not need any change. This works because any code that attempts to make problematic modifications to non-**obliv** variables inside an **obliv** scope will be rejected in our type-checking phase. Moreover, the conditional assignment only uses the conditions of enclosing **obliv if**s. We do not need to separately account for non-**obliv** conditions like i < n or i % 2 == 0 since those control structures are not oblivious and will execute normally.

Since loops and function calls remain in code as is, we never need to unroll or inline them into full circuits for execution, unlike other systems [10, 20]. Hence, we can run programs involving billions of gates without worrying about running out of memory. Memory management is not different in our system, since we still have full access to the usual C runtime library functions (although sometimes protocol-specific restrictions can apply, as seen in Section 5).

The last new feature we need to support is unconditional segments. Code written inside such a segment is simply rewritten as if it appeared outside any conditional context. Inside an unconditional segment, all code is executed unconditionally. Before this block is executed, however, the new variable of type **obliv bool** is simply initialized with a copy of the current condition so that it is available to the code in the body of this segment.

Security argument. Our design makes it easy to provide a strong argument that an Obliv-C program never leaks any secret information (so long as the underlying secure computation protocol is secure). Since **obliv** variables are encrypted data, there is no risk that they will be leaked or used in a way that leads to an implicit leak since the semantic value is not even visible to the executing program. The only way a semantic value is produced is through a call to a reveal() function that can convert from **obliv** variables to the non-**obliv** ones.

The code generator never generates a reveal() function, except where the corresponding function was used in the input program. So, we can never accidentally leak information from **obliv** variable if the type system is flawed. An error in the type system can result in incorrect code and surprising behavior, but never an information leak. For example, if the type system mistakenly allows an externally visible non-**obliv** variables to be modified in an **obliv if**, the resulting program would modify the variable regardless of the **obliv** condition (without branching). This emphasizes that our system relies on cryptography at runtime to provide security; the type rules are designed only to prevent programming mistakes.

7. Related Work

Many frameworks for secure computation have been published in recent years. Broadly speaking, they can be classified into two categories. First is the family of low-level frameworks that provide a library of cryptographic primitives that can be used to develop arbitrary protocols. Examples include FastGC [11], SCAPI [5], and L1 [27]. The advantage of using these frameworks is that they provide a high degree of customizability over the actual protocol execution. On the downside, however, users are generally expected to be experts either in cryptography, or in circuit structures, or both. The frameworks provide little or no type safety to prevent semantic errors, and it is difficult (or in some cases, impossible) to write applications in a way that it is portable across different protocols. In comparison, applications programmed in Obliv-C are fully portable across all protocols that work on Boolean circuits (unless they are written to deliberately use protocol-specific extensions). Moreover, the Obliv-C type system prevents accidental mistakes on the part of the programmer, without being so restrictive that it prevents programmers from writing useful functions.

The second family of frameworks entail high-level languages that try to completely abstract away the cryptographic parts, and allow the user to code in a special language as if it was ordinary programming. Examples include include Fairplay [20], CMBC-GC [10], KSS [16], PCF [15], Wysteria [26] and PICCO [36]. Unlike Obliv-C, these languages provide little opportunity for users to extend or alter protocols short of modifying the compiler directly. For example, none of these would allow a user to write custom ORAM protocols (since they manage all network traffic) or implement custom data structure libraries (since they manage all memory allocation) as we demonstrated was straightforward with Obliv-C. Some like Wysteria, though, provides very strong static type system that we do not — our type system is only intended to prevent mistakes, and relies on the underlying cryptography for security.

Thus, we consider Obliv-C to be somewhere in between the two previous families of secure computation frameworks, obtaining the best of both worlds. It provides sufficient control to enable rich extensibility, without requiring a programmer to design low-level circuits or understand the underlying cryptography.

Although our current implementation provides fairly good performance, it still does not incorporate all the optimizations that have been proposed recently. This includes using AES-NI instructions [1, 16] to garbled each gate, or OT-extension for malicious adversaries [13] (our current dual execution implementation does not use OT-extension). The design of Obliv-C makes it easy to incorporate those optimizations, and any newly discovered ones, without making any changes to the compiler.

Holzer et al. [10] attempted to leverage C in secure computation, but did not support most of the C language, while Obliv-C is a strict extension of C. Finally, since their approach generates a full circuit representation before actually executing it, it cannot scale to large circuits.

Finally, there are many other implementations that use a custom designed intermediate language to address memory issues such as PAL [22] and PCF [15]. These frameworks do not support custom sub-protocols the way we do. In this respect, they are closer to the other high-level languages that we have mentioned previously, since they abstract away the data-oblivious nature of computation and provide something closer to ordinary computation. Without a full static type system, they had to take draconian measures such as not allowing function calls within an if statement that depends on secret input, for example. This greatly limits the general applicability of these systems, and requires programmers to build applications in unnatural and tool-specific ways.

8. Conclusion

Multi-party secure computation is a vibrant and rapidly advancing research area, but progress is impeded by the difficulty in experimenting with protocols, applications, and implementation techniques with current systems. Researchers with new ideas for implementing secure computation protocols, or for optimizing applications, tend to find it necessary to implement a new protocol from basic primitives since previous frameworks lack the necessary expressiveness to experiment with new ideas at multiple levels of abstraction. Obliv-C provides an extensible programming tool for secure computation that provides a new option by exposing the important aspects of data-oblivious computation, while providing a high-level language and the ability to seamlessly integrate with standard C code.

References

[1] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient Garbling from a Fixed-Key Block Cipher. In *IEEE Symposium on Security and Privacy*, 2013.

[2] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *European Symposium on Research in Computer Security*, 2008.

[3] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally Secure Constant-Rounds Multi-Party Computation for Equality, Comparison, Bits and Exponentiation. In *Theory of Cryptography*. 2006.

[4] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *Advances in Cryptology—CRYPTO*. 2012.

[5] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. SCAPI: The Secure Computation Application Programming Interface. *IACR Cryptology ePrint Archive*, 2012.

[6] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. Minilego: Efficient Secure Two-Party Computation from General Assumptions. In *Advances in Cryptology—EUROCRYPT*. 2013.

[7] Tore Kasper Frederiksen and Jesper Buus Nielsen. Fast and Maliciously Secure Two-party Computation using the GPU. In *Applied Cryptography and Network Security*, 2013.

[8] Shafi Goldwasser, Silvio M. Micali, and Avi Wigderson. How to Play Any Mental Game, or a Completeness Theorem for Protocols with an Honest Majority. In *19th ACM Symposium on Theory of Computing*, 1987.

[9] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure Two-Party Computation in Sublinear (Amortized) Time. In *ACM Conference on Computer and Communications Security*, 2012.

[10] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure Two-Party Computations in ANSI C. In *ACM Conference on Computer and Communications Security*. ACM, 2012.

[11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *20th USENIX Security Symposium*, 2011.

[12] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. GPU and CPU Parallelization of Honest-but-Curious Secure Two-Party Computation. In *ACM Annual Computer Security Applications Conference*, 2013.

[13] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending Oblivious Transfers Efficiently. In *Advances in Cryptology—CRYPTO*, 2003.

[14] Stanislaw Jarecki and Vitaly Shmatikov. Efficient Two-Party Secure Computation on Committed Inputs. In *Advances in Cryptology—EUROCRYPT*, 2007.

[15] Ben Kreuter, Benjamin Mood, abhi shelat, and Kevin Butler. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *22nd USENIX Security Symposium*, August 2013.

[16] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Billion-Gate Secure Computation with Malicious Adversaries. In *21st USENIX Security Symposium*, 2012.

[17] Yehuda Lindell and Benny Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *Advances in Cryptology—EUROCRYPT*. 2007.

[18] Yehuda Lindell and Benny Pinkas. A Proof of Security of YaoâĂŹs Protocol for Two-Party Computation. *Journal of Cryptology*, 22(2), 2009.

[19] Yehuda Lindell and Benny Pinkas. Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer. *Journal of Cryptology*, 25(4), 2012.

[20] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay-Secure Two-Party Computation System. In *12th USENIX Security Symposium*, 2004.

[21] Payman Mohassel and Matthew Franklin. Efficiency Tradeoffs for Malicious Two-Party Computation. In *Public Key Cryptography*. 2006.

[22] Benjamin Mood, Lara Letaw, and Kevin Butler. Memory-efficient Garbled Circuit Generation for Mobile Devices. In *Financial Cryptography and Data Security*. 2012.

[23] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Conference on Compiler Construction*, 2002.

[24] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A New Approach to Practical Active-Secure Two-Party Computation. Crypto ePrint Archive, 2011. http://eprint.iacr.org/2011/091.

[25] Jesper Buus Nielsen and Claudio Orlandi. LEGO for Two-Party Secure Computation. In *Theory of Cryptography Conference*, 2009.

[26] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. 2014.

[27] Axel Schropfer, Florian Kerschbaum, and Gunter Muller. L1 — an Intermediate Language for Mixed-Protocol Secure Computation. In *35th IEEE Annual Computer Software and Applications Conference*, 2011.

[28] abhi shelat and Chih-hao Shen. Two-Output Secure Computation with Malicious Adversaries. In *Advances in Cryptology—EUROCRYPT*, 2011.

[29] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with O((logN) 3) Worst-Case Cost. In *Advances in Cryptology—ASIACRYPT*. 2011.

[30] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *ACM Conference on Computer and Communications Security*, 2013.

[31] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2), 1996.

[32] Xiao Shaun Wang, Yan Huang, TH Chan, Abhi Shelat, and Elaine Shi. Scoram: Oblivious ram for secure computation. In *ACM Conference on Computer and Communications Security*. ACM.

[33] Yan Huang and Jonathan Katz and David Evans. Quid Pro Quo-tocols: Strengthening Semi-Honest Protocols with Dual Execution. In *33rd IEEE Symposium on Security and Privacy*, 2012.

[34] Andrew C. Yao. Protocols for Secure Computations. In *23rd Symposium on Foundations of Computer Science*, 1982.

[35] Samee Zahur and David Evans. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *34th IEEE Symposium on Security and Privacy*, 2013.

[36] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: A General-Purpose Compiler for Private Distributed Computation. In *ACM Conference on Computer and Communications Security*, 2013.

Appendix

Given the difficulty of formally describing the full C language as described in the ISO standard, we describe our rules using a simplified C language where int and bool are the only data types, along with their obliv counterparts. The only control structures are if, obliv if and while loops. It includes functions, but we exclude structures, pointers, arrays, and other complex features. We also restrict declarations so that they may each introduce only a single new variable.

The rules are shown in Figure 10. The notations we use are given in Figure 9.

At the global level, we use a slightly different notation: $\Gamma \vdash D \Downarrow \Gamma'$ denotes the fact that that D is a valid global definition under Γ, and it produces a new type environment Γ' (we will skip rules for global declarations without accompanying definitions, but they are similar). These rules are shown in Figure 11. These simply specify the Γ and Δ values with which each function body is processed. Additional notations used here:

$\Delta_{\mathrm{n}}(t) = \{\text{return-}t, \text{nObl}\}$
$\Delta_{\mathrm{o}}(t) = \{\text{return-}t\}$
$\Gamma'_{\mathrm{F}} = \text{freeze}(\Gamma')$

While these rules just presented are enough to create the type-checker for our language, we also need the semantics of our constructs formally specified in order to prove any properties about not leaking information. In this case, we take an operational approach by specifying rewrite rules. These are shown in Figure 12. The function convert(c, code) takes in Obliv-C code and rewrites it to produce plain C code, assuming they need to be executed in the context of the condition c. For brevity, we provide a simplified version of these rules that just includes the interesting cases. After these transformations, an additional pass is made to replace the type names with C structures representing the obliv basic types.

At this point, it is easy to see that we never emit anything to convert an obliv variable to a non-obliv one at any point, and the generated code has no control dependecy on obliv data. This, along with the type system, proves that our language never leaks any data about obliv variables (other than by using reveal() family functions).

s, t are used for types
$t_{1\dots n}$ is a shorthand for the sequence $t_1, \dots, t_n$
$(t_{1\dots n}) \to t$ is the type of a function that takes n arguments of types $t_{1\dots n}$ and returns a t
$(t_{1\dots n})$ obliv $\to t$ is the type of the corresponding obliv function
e is for expressions.
f is for the name of a function.
x, y for variables
σ for a single statement, S for a sequence of statements, ε for the empty sequence
$O = \{\text{obliv int}, \text{obliv bool}\}$
$F = \{\forall t \notin O : \text{frozen } t\}$
D is the set of all syntactically correct declarations
Γ is the current type mapping
$\Delta \subseteq \{\text{loop}, \text{nObl}, \forall t.\text{return-}t\}$ holds information about the current set of allowed control flow operations.
$\langle\Gamma;\Delta\rangle \vdash S$ means S is a valid sequence of statements under $\langle\Gamma;\Delta\rangle$
$\langle\Gamma;\Delta\rangle \vdash e : t$ means e is a valid expression of type t under $\langle\Gamma;\Delta\rangle$
$(x\!:\!t)_{1\dots n}$ is a shorthand for $x_1\!:\!t_1, \dots, x_n\!:\!t_n$
$\text{freeze}(\Gamma) = \{x\!:\!\text{freeze}(t) \mid \forall x\!:\!t \in \Gamma\}$
$\text{freeze}(t) = (\text{frozen } t)$ if $t \notin O$, t if $t \in O$
$\text{unfreeze}(t) = t'$ if $t = (\text{frozen } t') \in F$, t otherwise. $\text{unfreeze}(\Gamma)$ is analogous.

Figure 9: Notations used in Figure 10.

$$\frac{\langle \Gamma, x\!:\!t\,;\Delta\rangle \vdash S \quad t\;x \in D}{\langle \Gamma;\Delta\rangle \vdash t\;x; S}$$

$$\frac{\langle \Gamma;\Delta\rangle \vdash \sigma \quad \langle \Gamma;\Delta\rangle \vdash S \quad \sigma \notin D}{\langle \Gamma;\Delta\rangle \vdash \sigma\, S}$$

$$\frac{}{\langle \Gamma;\Delta\rangle \vdash \varepsilon}$$

$$\frac{\langle \Gamma;\Delta\rangle \vdash S \quad \sigma = \{S\}}{\langle \Gamma;\Delta\rangle \vdash \sigma}$$

$$\frac{\langle \Gamma;\Delta\rangle \vdash e : t \quad t \notin O \quad \text{obliv}\ t \in O}{\langle \Gamma;\Delta\rangle \vdash e : \text{obliv}\ t}$$

$$\frac{\langle \Gamma;\Delta\rangle \vdash e_1 \!:\! t \quad \langle \Gamma;\Delta\rangle \vdash e_2 \!:\! t}{\langle \Gamma;\Delta\rangle \vdash (e_1\ \text{op}\ e_2) \!:\! t}$$

$$\frac{\langle \Gamma;\Delta\rangle \vdash x\!:\!s, e\!:\!t \quad s \in O \vee t \notin O \quad s \notin F}{\langle \Gamma;\Delta\rangle \vdash x = e;}$$

$$\frac{\langle \Gamma;\Delta\rangle \vdash \sigma \quad \langle \Gamma;\Delta\rangle \vdash e\!:\!t \quad t \notin O}{\langle \Gamma;\Delta\rangle \vdash \text{if}\ (e)\ \sigma}$$

$$\frac{\langle \Gamma;\Delta,\text{loop}\rangle \vdash \sigma \quad \langle \Gamma;\Delta\rangle \vdash e\!:\!t \quad t \notin O}{\langle \Gamma;\Delta\rangle \vdash \text{while}\ (e)\ \sigma}$$

$$\frac{\text{loop} \in \Delta \quad \sigma \in \{\text{break}, \text{continue}\}}{\langle \Gamma;\Delta\rangle \vdash \sigma;}$$

$$\frac{\text{return-}t \in \Delta \quad \langle \Gamma;\Delta\rangle \vdash e\!:\!t}{\langle \Gamma;\Delta\rangle \vdash \text{return}\ e;}$$

$$\frac{\langle \text{freeze}(\Gamma)\,;\emptyset\rangle \vdash \sigma \quad \langle \Gamma;\Delta\rangle \vdash e}{\langle \Gamma;\Delta\rangle \vdash \text{obliv if}\ (e)\ \sigma}$$

$$\frac{\langle \text{unfreeze}(\Gamma), x\!:\!\text{obliv bool}\,;\Delta\rangle \vdash \sigma}{\langle \Gamma;\Delta\rangle \vdash \sim\!\text{obliv}(x)\ \sigma}$$

$$\frac{\langle \Gamma;\Delta\rangle \vdash (e\!:\!t)_{1\ldots n} \quad \langle \Gamma;\Delta\rangle \vdash f : (t_{1\ldots n})\ \text{obliv} \rightarrow t}{\langle \Gamma;\Delta\rangle \vdash f(e_1,\ldots,e_n)\!:\!t}$$

$$\frac{\langle \Gamma;\Delta\rangle \vdash (e\!:\!t)_{1\ldots n} \quad \langle \Gamma;\Delta\rangle \vdash f : (t_{1\ldots n}) \rightarrow t \quad (\text{nObl}) \in \Delta}{\langle \Gamma;\Delta\rangle \vdash f(e_1,\ldots,e_n)\!:\!t}$$

Figure 10: Type rules for our language extensions for simplified C.

$$\frac{\Gamma_1 \vdash D_1 \Downarrow \Gamma_2 \quad \Gamma_2 \vdash D_2 \Downarrow \Gamma_3}{\Gamma_1 \vdash D_1 D_2 \Downarrow \Gamma_3}$$

$$\frac{\Gamma \vdash t}{\Gamma \vdash t\ x; \Downarrow \Gamma \cup \{x : t\}}$$

$$\frac{\Gamma \vdash t_r, t_{1\ldots n} \quad \Gamma' = \Gamma \cup \{f : (t_{1\ldots n}) \to t_r\} \quad \langle \Gamma', (x : t)_{1\ldots n} ; \Delta_{\mathrm{n}}(t_r)\rangle \vdash S}{\Gamma \vdash t_r\ f(t_1\ x_1, t_2\ x_2, \ldots, t_n\ x_n)\{S\} \Downarrow \Gamma'}$$

$$\frac{\Gamma \vdash t_r, t_{1\ldots n} \quad \Gamma' = \Gamma \cup \{f : (t_{1\ldots n})\ \text{obliv} \to t_r\} \quad \langle \Gamma'_{\mathrm{F}}, (x : t)_{1\ldots n} ; \Delta_{\mathrm{o}}(t_r)\rangle \vdash S}{\Gamma \vdash t_r\ f(t_1\ x_1, t_2\ x_2, \ldots, t_n\ x_n)\ \text{obliv}\ \{S\} \Downarrow \Gamma'}$$

Figure 11: Rules for type checking global declarations.

$$\text{convert}(c, v_1 = v_2;) \equiv \begin{cases} \text{obliv\_c\_copy}(\&v_1, \&v_2); & \text{if } c \text{ is statically true} \\ \text{obliv\_c\_condAssign}(\&c, \&v_1, \&v_2); & \text{otherwise} \end{cases}$$

$$\text{convert}(c_1, \text{obliv if } (c_2) B_1 \text{ else } B_2) \equiv \text{convert}(c_1 \& c_2, B_1)\, \text{convert}(c_1 \& ! c_2, B_2)$$

$$\text{convert}(c, f(e_1, \ldots, e_n);) \equiv \begin{cases} f(c, e_1, \ldots, e_n); & \text{if } f \text{ is an obliv function} \\ f(e_1, \ldots, e_n); & \text{otherwise} \end{cases}$$

$$\text{convert}(c, \sim \text{obliv}(v) B) \equiv \text{convert}(\text{true}, \text{obliv bool } v = c; B)$$

$$\text{convertFunc}(t_r\ f(t_1\ v_1, \ldots, t_n\ v_n)\ \text{obliv}\ B) \equiv$$
$$\quad t_r\ f(c, t_1\ v_1, \ldots, t_n\ v_n)\ \text{convert}(c, B) \text{ where } c \text{ is fresh}$$

$$\text{convertFunc}(t_r\ f(t_1\ v_1, \ldots, t_n\ v_n) B) \equiv$$
$$\quad t_r\ f(t_1\ v_1, \ldots, t_n\ v_n)\ \text{convert}(\text{true}, B)$$

Figure 12: Rewrite rules for compiling Obliv-C to plain C.