Poster: Obliv-C: a Fast, Lightweight Language for Garbled Circuits

Samee Zahur and David Evans
University of Virginia
[*samee, evans*]@*virginia.edu*

***Abstract*—We present Obliv-C (http://oblivc.org), a new language for developing secure computation protocols. It is a very lightweight extension layer on C that gets rewritten into standard C for compilation directly to native binary programs. The system provides two main benefits. First, it uses state-of-the art optimizations and native code execution to attain very fast execution speeds (over 2.2 M non-linear gates per second in our LAN experiments). Second, it has full access to existing C libraries. This means no effort had to be invested in building up libraries for multithreading, cryptography, networking etc. Here we show how to easily write up application programs, and how to easily add new features in the form of user libraries.**

I. Motivating Example

Figure I shows how to set up a quick and simple secure, two-party computation function in Obliv-C, and integrate it to an ordinary C program. The first function is the Obliv-C part that performs the actual secure computation, while the second is just a main driver program, written in plain C. They can both reside on the same file, since any C program is also valid Obliv-C. In this example, the function just compares two integers securely (a.k.a. the millionaires' problem) and outputs a boolean result.

The main thing that Obliv-C introduces is the 'obliv' type qualifier. When used with basic C types (e.g. obliv int or obliv char), it represents variables which are stored in

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

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

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

        execYaoProtocol (&pd, millionaire, &io);
        printf ("Result: %d\n", result);
        // ... cleanup
23  }
```

The full program code for the millionaires' problem in Obliv-C

cryptographic form. As such, they are unintelligible to the execution system of either party on their own. These values will only be revealed if both parties agree to do so, e.g. calling revealOblivBool at line 10 here to reveal the result. This reveals an obliv bool value to produce an ordinary bool value, which can be returned (or as in here, stored in io->result).

At execution time, both parties will be executing this function synchronously with their own inputs stored in the struct UserIO. The struct is a just a user-defined plain C data type defined at the top.

Lines 7,8 show how feedOblivInt calls are used to input private inputs. Line 7 reads the input from party-1's copy of io->myinput, and store it in a, which is now secret since it is of type obliv int. Similarly, the next line takes input from party-2. At this point, we can perform arbitrary computation on them as usual, after which both parties invoke reveal on the value they both agree should be shared.

Finally, this protocol can now be integrated into any ordinary C program. The main function forms a very simple driver program. It simply performs some initialization and setup tasks. For example, the command line is parsed to check if we are currently party-1 or party-2, and the private input is written into the user variable io.myinput so that it can be read later during the protocol execution. After that, we can kick off the protocol using execYaoProtocol.

Figure I is all the code required for an end-to-end execution. The Obliv-C file is compiled to C using our own compiler, which is then passed on to GCC. Note that this directly produces a standalone program for a particular application. Unlike most other existing applications, there is no separate 'circuit file' that needs to be path-configured or lugged around with the executable. It also eliminates a need for an additional layer of interpreter, which is part of the reason we were able to achieve a high performance. In a fast LAN setting our Obliv-C implementation can go up to 2.2 million non-linear gates per second (not counting any XOR or NOT gate executed, as usual [2]). On a slow network, we will need around 160-bits for every non-linear gate in the default 80-bit security level. This is achieved using the recent halfgates garbling scheme [4].

II. Developing Libraries

One of the advantages about being closely tied to an established language is that it is very easy to integrate with a

Benchmark	Gate count (g)	OT ext., sec (s)	Gate exec., sec (t)	Total, sec ($s+t$)	Gate rate (million gates/second) (g/t)
AES, with key expannsion	385,056	0.260	0.154	0.414	2.5
Levenshtein, 200x200 chars	6,678,412	0.26	2.04	2.30	3.3

Table I
PERFORMANCE OF TWO COMMON BENCHMARKS.

large body of existing libraries. As a result, the user of Obliv-C can add many new features without having to modify the compiler or backend libraries at all. For example, if someone wants to integrate oblivious RAM into garbled circuits [1], [3], there are just way too many ways of constructing oblivious RAM, and there is no one 'right approach' for all purposes. So we simply allow Obliv-C users to implement such techniques as simple library functions.

Other features have also been developed as a library, separate from the compiler, showing that they can be done without any backend modifications. These include optimization utilities such as limited-width integers for arithmetic or circuit structures for ordered memory accesses.

And it is not just SMC-specific tasks that benefit from this. Even standard system tasks such as multithreading part of computation is rarely supported in new languages with the full flexibility of existing languages. Here, we will elaborate on two specific examples: mutexes and range-limited integers.

Mutex. Almost all multithreaded programs will require synchronization at some point in order to ensure correctness. Standard UNIX distributions obviously provide a large number of primitives such as mutex, semaphores and barriers. Although Obliv-C threading support is not quite complete, we demonstrate how mutexes can be easily added into Obliv-C.

The challenge with this task is that ordinary mutexes will not work during a protocol execution — if two threads are competing for a lock, we need to make sure that the same thread wins in both parties. A simple approach is shown in Figure 1. There are ways to do improve efficiency, but this implementation explains the main ideas well. Here we are wrapping the standard pthreads mutex call so that they can be used during protocol execution. The approach is that only party-1 has an underlying pthreads mutex handle. When party-2 needs a lock, it simply waits for a signal from party-1 to decide which thread moves forward. So if multiple threads call `oblivc_mutex_lock` on the side of party-2, they all wait until one of the receive-dummy-message operations finish. Threads from party-1 send out this dummy signal as soon as they win the lock. The corresponding `obliv_mutex_unlock` call should be a no-op for party-2.

```
void oblivc_mutex_lock(pthread_mutex_t* m) {
  if(ocCurrentParty()==2) recvDummy(1);
  else {
    pthread_mutex_lock(m);
    sendDummy(2);
  }
}
```

Figure 1. A possible mutex implementation

```
struct RangeInt
{
    int lo,hi;
    obliv int value;
};
```

Figure 2. Range-tracked integer type

Thus, we see how a user who wants to invoke an existing synchronization mechanism in Obliv-C will not have to wait for us to implement all the myriad different primitives into this new language. It is very easy for the user to just write a simple library wrapper.

Range-limited Integers. By default, all 'obliv int' variables in Obliv-C are 32-bits. This means that every arithmetic operation would require add, multiply etc. circuits for 32-bit integers. However, in a typical program, many integers are often small enough to be represented in fewer bits, allowing more efficient arithmetic operations. We now describe how a user-defined type (Figure 2) can be used in place of the built-in types in such cases.

The definition has two ordinary, publicly known, integers fields lo and hi, indicating the range in which the secret 'value' is known to be. So when doing any arithmetic on these values, we can now write library functions that determine how to perform computation based on the known range. E.g. when adding two integers, a library function now has two do two things. First, it needs to add the 'value' field of the two RangeInt objects with a smaller circuit. Second, it needs to add the corresponding lo and hi values to produce new ones for the result RangeInt object. This allows circuit structure (e.g. width) to adapt automatically according to the application, in the sense that the library can track integer range dynamically through the operations. Similar libraries can be implemented in Obliv-C for other circuit structures, such as stacks, queues, maps etc.

III. PERFORMANCE AND CONCLUSION

Table I shows our performance over a LAN connection between single-threaded programs[1] running between two desktops, for 80-bit key sizes, for semi-honest security. The circuit sizes could be further reduced by manual optimization. All numbers are averages over 10 executions, and standard deviations were less than 2% in all cases. Thus Obliv-C provides fast performance, while making further experimentation easy.

[1] some parts of OT extension is parallelized, but we don't count that in the gate execution rate