Oblivious Computation TinyGarble, Frigate and Obliv-C

Osman Cihan Kilinc - okilinc@ucsd.edu
Sanjay John - sjohn@ucsd.edu

Introduction

Secure Multiparty Computation is a field within cryptography that aims to solve a difficult problem; the calculation of a function over several participants’ inputs without revealing any of their inputs to the other participants. While traditional cryptography aims to conceal information from an adversary outside of the system, here we assume the adversary actually controls actual participants.

SMC was formally introduced as the Millionaires’ Problem by Andrew Yao in 1982, where two millionaires (Alice and Bob) wish to find out which of them is richer without revealing their actual wealth. The protocol consists of 6 steps as follows:

1. The underlying function (e.g., in the millionaires' problem, comparison function) is described as a Boolean circuit with 2-input gates. The circuit is known to both parties. This step can be done beforehand by a third-party, and is mainly being investigated here.
2. Alice garbles (encrypts) the circuit. We call Alice the garbler.
3. Alice sends the garbled circuit to Bob along with her encrypted input.
4. Bob through oblivious transfer receives his encrypted inputs from Alice.
5. Bob evaluates (decrypts) the circuit and obtains the encrypted outputs. We call Bob the evaluator.
6. Alice and Bob communicate to learn the output.

Problem Statement

In this project, we aim to compare 3 different SMC frameworks: TinyGarble, Frigate and Obliv-C. We will do so with the implementation of three benchmarks: Bubblesort, Mergesort and Dijkstra’s Shortest Path Algorithm. Additionally, we will be attempting to reduce the number of AND gates needed to implement these algorithms as much as possible, and through this or by other means reduce the runtime required. The implementation of these algorithms are also of concern to us as these applications are not straightforward, and require workarounds to deal with the inherent limitations of these systems; these will be discussed throughout the report.

TinyGarble

TinyGarble views circuit generation for GC as an atypical logic synthesis task addressable by conventional hardware synthesis tools. By posing the circuit generation for Yao’s protocol as a hardware synthesis problem, it benefits from the algorithms and techniques already incorporated in existing logic synthesis solutions. TinyGarble’s libraries aim to reduce the cost of building these circuits by optimizing and using as many XOR gates instead of traditional AND gates as possible.

Limitations

While this can be done through high-level language conversions, our aim to reduce the number of gates can be hindered by this process (since HLS tools are known to be inefficient in their conversions) and therefore requires that we try to write hardware synthesis solutions ourselves in Verilog. This brings in a set of challenges, most of which stem from our inexperience in writing Verilog. We are unable to perform non-deterministic traversal of an array (i.e. move along the array given a condition) because of the set hardware limitations that these tools describe.

Bubblesort

Implementing Bubblesort is fairly straightforward; since the traversal of the array is determined beforehand, the main idea is to reduce the number of operations needed during the swap. This was done using XOR operations to swap the variables, i.e.

```
X = X XOR Y
Y = X XOR Y
X = X XOR Y
```

However, we find that this operation does not seem to reduce the number of gates as opposed to a traditional temporary variable method. Regardless, this approach was used in order to curb the need for a temporary register.

Mergesort

Mergesort becomes more difficult as we cannot use recursion nor a conditional array traversal. One attempt at making this work involved using nested conditional statements that simply checked each element of the array against the others and followed a chain of logic according to this condition, i.e.

```
if (dat1>dat4) begin
        array[0] = dat4;
        if (dat1>dat5) begin
                array[1] = dat5;
                if (dat1>dat6) begin
                        array[2] = dat6;
```

```
                              array[3] = dat1; \\etc
```

However, this approach was not parameterizable at all, and the complexity of the code heavily depended on the number of elements used. Therefore to make this parameterizable, we followed an approach that is also used in Frigate and will be discussed in that section.

Dijkstra’s Algorithm

Implementing Dijkstra’s Algorithm in Verilog is no easy task, and one attempt at solving the problem involved using Finite State Machines, i.e. separating the task into states and calling back these states to simulate the loops required, like below:

```
 always@(posedge clk) begin
        if(reset == 0) begin
         state        <= graph_initialization;
         counter       <= 9'd0;
         sub_counter    <= 9'd0;
        else if(state == graph_initialization && counter == NUMVALS) begin
         state        <= distance_initialization;
        counter       <= start;
        else if(state == dsn_init) begin
         state        <= min_cost; \\ find minimum distance
         counter       <= 9'd0;
        else if(state == min_cost && counter != NUMVALS) begin
         state        <= upd_minc; \\update minimum distance
         sub_counter    <= 9'd0;
        else if(state == upd_minc && counter != NUMVALS) begin
         state        <= upd_dist; \\ update distances for nodes previously not known
        else if(state == upd_dist && counter != NUMVALS) begin
         state        <= min_cost; \\go back and find minimum distances
         counter       <= counter + 9'd1;
        else begin
         state        <= done; \\finish        end
```

However, as with Mergesort, we found this method unable to be parameterized, unable to deal with larger inputs and therefore unusable in the context of comparisons of performance. We ended up following a similar algorithm used in Frigate, which will be explained more below.

Frigate

Frigate was created to address the issues that authors have encountered in the previous frameworks, namely integrity and efficiency. It includes the optimizations that were proposed after the creation of these frameworks. Therefore, it is faster than the previous circuit compilers. Authors also claim that Frigate is verifiable, and the results are more accurate than the previous frameworks. Moreover, it was developed using a principled, simple and modular design, aimed to make it easily extensible. Authors also validated the compiler through systematic and rigorous unit tests.

Limitations

The dependencies and compatibility issues were the first of the many limitations that we encountered. Frigate requires Ubuntu 14.04, and does not work with other versions. This is not mentioned in the documentation. After examining other projects, we concluded that this was indeed the issue and it was later confirmed by Benjamin Mood (the author of Frigate). The inputs and outputs to Frigate is binary. Therefore, to test our implementations we had to write a special input generator for Frigate. However, interpreting the output of Frigate remained an issue. It is hard to interpret the binary output as the output text file only includes the description of the circuit. It was hard to understand where an element finished and the other started, and in what order they were written. It is mentioned in the documentation that comparisons require signed numbers, however it does allow comparisons to be made with unsigned numbers; it does not throw an error. The compiler also throws a warning when signed numbers are used as an index to access the arrays. We observed that unsigned integers were safely used on *for loops* in other projects. Therefore, we used unsigned integers on the *for loops*, which also includes a comparison to stop. We do not know how accurately Frigate is handling such comparisons. Frigate does not allow recursive functions to be defined, let alone use it in the main function. Therefore, all the implementations had to be written in an iterative manner. Moreover, all the array accesses have to be predefined; it can not be input dependent. For merge-sort, and dijkstra, the array accesses are input-dependent. Therefore, we developed a deterministic implementation, which is slower. Finally, and most importantly, we observed that the inputs were read from right to left. When giving an input right to the output, the outputs seems to be exactly the same as the input. However, when making some operations on the input, we observed that the output wires become flipped, making it harder to interpret.

Bubblesort

Bubblesort is straightforward algorithm that uses a predefined scheme to access the arrays. Therefore, we did not have any problems in implementing bubblesort. The implementation follows the standard pseudocode for Bubblesort, which accesses all the array elements twice, and swaps the elements according to a condition. The runtime is $O(n^2)$.

Mergesort

Mergesort algorithm is not as straightforward as Bubblesort. The accesses made to the arrays are input-dependent. Frigate does not allow such accesses, therefore we implemented an iterative version of batcher sort, also known as odd-even merge sort. Our implementation can be further optimized. However, given the restrictions we could not deliver such optimizations with the limited time. Our implementation is $O(n\log^2 n)$. Since mergesort includes a lot less comparisons than bubblesort, it also has less number of Non-XOR gates.

Dijkstra’s Algorithm

Dijkstra’s algorithm depends on the many conditions. Thus, requires many AND gates. The algorithm traverses through the graph using the neighbors with the minimum distance. However, we are restricted in terms of array accesses and finding the minimum, as we can not reveal intermediate results to other parties. Therefore, in our implementation, we access all the neighbors starting from the neighbor with the smallest index and we check whether the distance is smaller than the previously visited neighbors. We also record our accesses and the parents of nodes with respect to the start node. Our implementation was $O(n^3)$ and compared to other frameworks Frigate performs the worst by far.

Obliv-C

Obliv-C is a gcc wrapper that guarantees secure computation if the underlying protocol is secure. It is more extensible and the language is very similar to C, but includes a new *obliv* datatype. It is written to be practical and the programmer does not need to be an expert in cryptography. It provides a higher-level programming abstractions, which simplifies the developing process.

Limitations

Obliv-C is less restrictive compared to Frigate. However, it requires an OCaml compiler that is up-to-date. It took us several hours to get it running on an Ubuntu 14.04. Obliv-C restricts any changes under the *obliv if* and *obliv* function statements. Therefore, it restricts input dependent array accesses. The error messages are not informative, and error handling was insufficient. Some of the built-in functions, such as *debugExecProtocol()* gives a segmentation fault, without any explanation. Under such circumstances, it was difficult to debug the implementations and understand the problems.

Bubblesort

The implementation of bubblesort follows the standard pseudocode of the algorithm. The array accesses are not input-dependent. Each and every element are compared to one another. Swaps are made according to a condition. Our implementation has a runtime of $O(n^2)$.

Mergesort

The development process of mergesort was problematic. After hours were spent on understanding the problem and several workarounds was implemented, we concluded that the mergesort can be best solved with batcher sort algorithm. This conclusion came after we examined similar projects on the tests folder. Our implementation uses the *batcherSort(.)*

algorithm found *“~/obliv-c/test/oblivc/psi.oc”*, where authors of obliv-c implemented a recursive version of Batcher’s sort (odd-even mergesort) algorithm. The runtime is O(nlog^{2}n). The algorithm sorts the arrays using a predefined array access scheme, similar to mergesort, without revealing any intermediate results or the accesses to other parties.

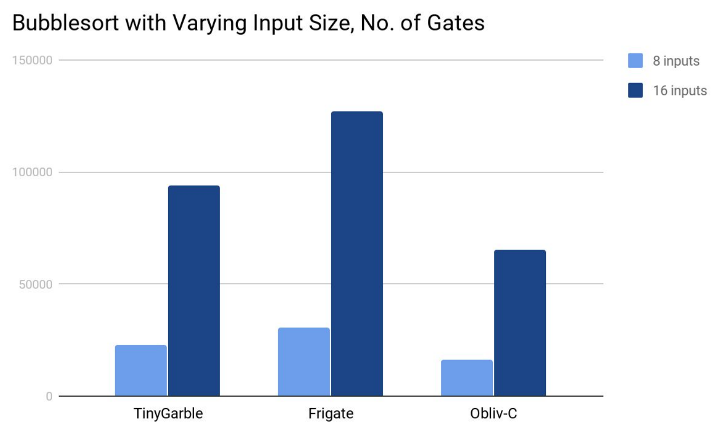
Dijkstra’s Algorithm

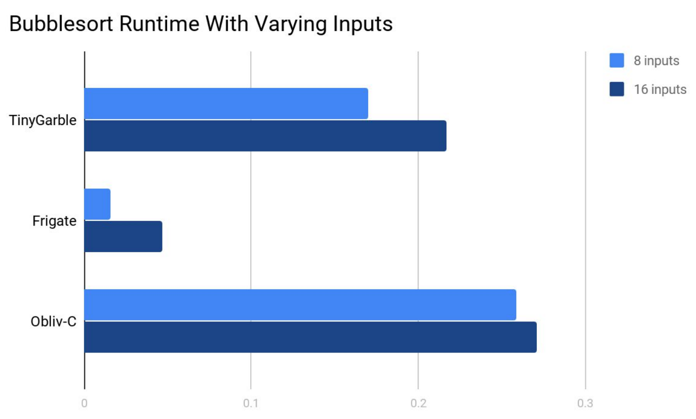
Dijkstra’s algorithm relies on the use of many conditions; thus, requires many *AND* gates. Similar to mergesort, the standard pseudocode depends on the inputs for array accesses. After initialization, the algorithm traverses to the neighbor with the minimum distance and records the minimum distances of each node with respect to the start node. Since we can not reveal any intermediate results, we use a predefined scheme in which we traverse to the neighbor with the smallest index and is within a smaller distance than the previously visited nodes. Using this array access scheme, we explore all the paths between the nodes. Thus, our runtime is O(n^3).

Results

No. of gates (32 bits, 16 nodes/party)	**TinyGarble (non-XOR)**	**Frigate (non-XOR | Total)**	**Obliv-C (non-XOR)**
Bubblesort	93717	126946 | 540450	65536
Mergesort	36069	8066 | 28351	12224
Dijkstra’s (6 nodes)	34100	5172744 | 19023468	38596

Runtime (seconds) (32 bits, 16 nodes/party)	**TinyGarble (seconds)**	**Frigate (seconds)**	**Obliv-C (seconds)**
Bubblesort	0.217	0.047	0.270
Mergesort	0.145	0.044	0.245
Dijkstra’s (6 nodes)	0.075	1.57	0.259





Conclusion

We can see that for the most part, Obliv-C keeps the number of gates to a minimum but its runtime is longer, however the runtime does not change as much with input size as the other frameworks. Frigate makes quick work of running the algorithms, while its interface leaves much to be questioned. TinyGarble sits squarely in the middle in terms of non-XOR gates used as well as runtime, and provides perhaps a good compromise between both save for its accessibility in terms of writing functions in Verilog. We also found that circuit sizes over an amount were unable to be compiled; for example, 32 element input arrays were unable to be compiled with Synopsys Design Compiler.

The main task in implementing these benchmarks stemmed from the inability to write non-deterministic nor recursive algorithms; the workarounds ended up increasing the number of gates and runtime. The iterative odd-even mergesort algorithm used in both TinyGarble and Frigate highlight the nature of these workarounds as they are not straightforward or readable; this may be a reason to push for more development of C-friendly SMC frameworks that allow recursive operations and input-dependent conditions and array traversal. More work in optimization of these algorithms can be done if attention is paid to exchanging regular software implementations to bit operations; since these algorithms get translated into binary logic circuits, these operations will suit these frameworks and allow for less gates and runtime.

References

[1] Songhori, Ebrahim M., et al. "Tinygarble: Highly compressed and scalable sequential garbled circuits." *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015.
[2] Mood, Benjamin, et al. "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation." *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016.
[3] Zahur, S.: Obliv-C: A lightweight compiler for data-oblivious computation (2014). https://github.com/samee/obliv-c

Appendix

TinyGarble

Bubblesort

8 element arrays
Gate Count: 22653 (non-XOR) - Run Time = 0.01565 seconds
16 element arrays
Gate Count: 93717 (non-XOR) - Run Time = 0.0470444 seconds

Mergesort

8 element arrays
Gate Count: 11889 (non-XOR) - Run Time = 0.023142 seconds
16 element arrays
Gate Count: 36069 (non-XOR) - Run Time = 0.044793 seconds

Dijkstra’s Algorithm

6 element arrays
Gate Count: 30 (non-XOR) - Run Time = 1.57699 seconds

Frigate

Bubblesort

8 element arrays
Gate Count: 30690 (non-XOR) || 103057 (All) - Run Time = 0.01565 seconds
16 element arrays
Gate Count: 126946 (non-XOR) || 540450 (All) - Run Time = 0.0470444 seconds
32 element arrays
Gate Count: 516066 (non-XOR) || 2201282 (All) - Run Time = 0.177845 seconds

Mergesort

8 element arrays
Gate Count: 8066 (non-XOR) || 28351 (All) - Run Time = 0.023142 seconds

16 element arrays
Gate Count: 8066 (non-XOR) || 28351 (All) - Run Time = 0.044793 seconds
32 element arrays
Gate Count: 8066 (non-XOR) || 28351 (All) - Run Time = 0.1594 seconds

Dijkstra’s Algorithm

6 element arrays
Gate Count: 5172744 (non-XOR) || 19023468 (All) - Run Time = 1.57699 seconds
8 element arrays
Gate Count: 5172744 (non-XOR) || 19023468 (All) - Run Time = 1.58084 seconds
16 element arrays
Gate Count: 5172744 (non-XOR) || 19023468 (All) - Run Time = 1.59381 seconds
32 element arrays
Gate Count: 5172744 (non-XOR) || 19023468 (All) - Run Time = 1.64 seconds

Obliv-C

Bubblesort

8 element arrays
Gate Count: 16384 - Run Time = 0.258252 seconds
16 element arrays
Gate Count: 65536 - Run time = 0.270788 seconds
32 element arrays
Gate Count: 262144 - Run Time = 0.327137 seconds

Mergesort

8 element arrays
Gate Count: 4032 - Run Time = 0.246502 seconds
16 element arrays
Gate Count: 12224 - Run time = 0.245326 seconds
32 element arrays
Gate Count: 34752 - Run Time = 0.262338 seconds

Dijkstra’s Algorithm

6 element array
Gate Count: 38596 - Run Time = 0.26 seconds
8 element arrays

Gate Count: 96856 - Run Time = 0.282097 seconds
16 element arrays
Gate Count: 842056 - Run time = 0.483835 seconds
32 element arrays
Gate Count: 7017256 - Run Time = 2.236623 seconds