CS523 Project 1 Report

Romain Mendez, Julien Heitmann

***Abstract*—This project aims at designing a N-party multiparty computations (MPC) engine in a semi-honest (passive) adversarial setting in the Go programming language.**

I. Introduction

The project is divided into two parts. The first parts provides a basic MPC framework, that works for generic circuits, assuming the existence of a trusted third-party. The second relaxes this strong condition by using homomorphic encryption to replace the trusted third-party by a pre-processing phase. We define a threat model for both parts, present the implementation details, evaluate the performance of the two approaches and discuss advantages and disadvantages of using either of them.

II. Part I

A. Threat model

In this first part, the system assumes the network links between the parties are safe (i.e no man in the middle attacks). This is because while the shares being sent over the network are provably secure we can't do anything against a man in the middle (no signature scheme deployed) and it would lead essentially to have a malicious peer which we can't do anything against.

For the peers involved in this scheme, they can be honest but curious however they can't be malicious otherwise the result of the computation can't be trusted.

In this first part we also implicitly trust the provider of Beaver triplets for the multiplication as we have otherwise no way of generating them for the time being.

B. Implementation details

- Class structure
- Concurrence issue
- Helper methods
- Unit testing
- Detail the circuit you created at the end of the first part

We'll detail all these points in the following paragraphs.

1) Class structure: In this first part, we implemented the logic of evaluating the circuit in *mpc.go*. It possesses a *run* method that will take into parameters both the beaver triplets and the circuit to start itself. Then since the run is implemented similarly on all the clients they will evaluate the gates in the same order, making sure there is no synchronisation issue between the peers for the intermediate broadcasts for the multiplication.

Also, our evaluation of the circuits was built recursively. This was the easiest way for us to create a robust evaluation method for all circuits was to start recursively from the output and go back up to the beginning. We provide a map for the intermediate results to be stored so that the evaluation can proceed normally.

This recursive approach is built in *operations.go* where the structure was largely the one provided but expanded to include the necessary features we needed.

2) Concurrence issue: In our first version of this engine, we were always just waiting for a set amount of messages to come in. Since our structure was very much built from the structure of *dummy.go* it felt normal to just wait for a set amount of messages arrive. This rapidly revealed concurrence issues, where some peers would start broadcasting their next multiplication intermediate broadcasts before some peers sent a previous intermediate broadcast.

We solved this issue by implementing a queue system. And making sure we always get a set amount of messages from all peers. If a message arrived before the other peers had sent theirs, it was placed in a queue that was continually checked before the incoming new messages in order to guarantee the good order of dealing with incomming messages.

3) Helper methods: We decided against putting all the helper methods in the file with the circuit evaluation code. To us it felt sort of aside from the project, something that could be reused in a bigger project so we created small files like *utils.go* to house the methods like the one creating random 64-bit elements securely.

Also, since the peers shouldn't have the entire circuit with other's inputs in it. We created another *circuit.go* housing a slightly different representation of the circuit with a method designed to parse the real circuit before it is sent to the protocol created in *mpc.go*.

4) Unit testing: In order for us to efficiently develop this project we added along the way unit tests to validate certain parts of the code. These files can be identified because they always have the same name as other files but with "_test" appended to them. We decided against putting them in a separate folder because of Golang's strange implementation of dependencies (certain directories in Windows and others in Linux and so on based on their environment variables). And ultimately there wasn't that much of them and it didn't matter if a simple user came in and just used *go build* since the result would be the same.

C. Our circuit

Our scenario is the following: imagine a new dating website, that computes how well two people match based on a set of binary questions, without revealing the answers to either participant or the website itself. Moreover, the users can select questions that are more important to them, which will increase

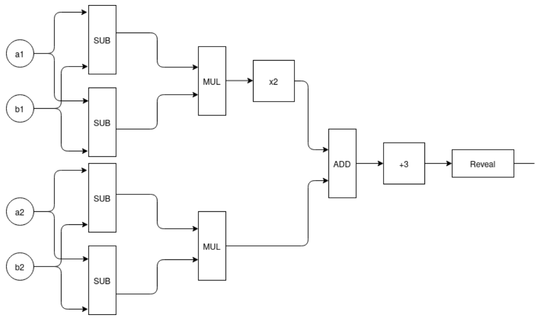

Fig. 1: Private Dating Circuit

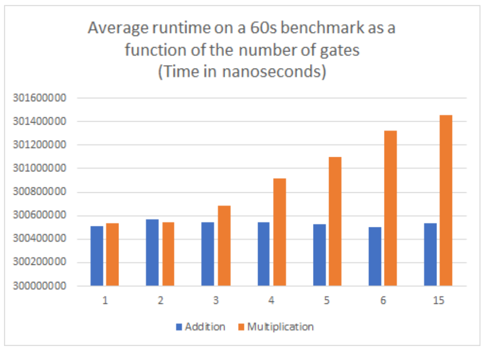

Fig. 2: Performance numbers without triplet generation

their contribution to the final matching score. In a simple example with only two questions answered, the function $f(a_1, a_2, b_1, b_2) = 2(a_1 - b_1)(b_1 - a_1) + (a_2 - b_2)(a_2 + b_2) + 3$ is computed, where $a_1, a_2, b_1, b_2 \in \{0, 1\}$ are the answers of participants A and B respectively. Figure 1 shows how this function is implemented with our circuit logic, which yields a matching score between 0 and 3, 3 implying that the two participants have provided the same answers to the questions. Note that with only 2 questions, it is easy to determine the answers of another user, but this logic can be extended to an arbitrary number of questions, making it increasingly difficult (if not impossible) to guess the answers of another user.

III. Part II

A. Threat model

In this section the threat model is very similar to the previous scenario. However this time we the peers can generate the beaver triplets by themselves, thus do not rely on a trusted third-party. Similarly to the first part we expect no malicious peer, **and** the malicious eavesdropper or else are computationally bounded so that they can't break the homomorphic encryption we're using.

B. Implementation details

The structure of the code was well inspired by the first part (and subsequently *dummy.go*) so we didn't run into concurrence issue this time. But the structure to deal with it is very similar.

So we built a different protocol *beaver.go* to implement the algorithm given in the handout. To complement it we augmented our *utils.go* to house the helper methods needed for this part (the operations on slices for example).

The *run* method will just run the algorithm provided in the handout. In the main code (i.e *mpc.go*) we deal with having the required amount of triplets there. It is also in charge of running this protocol on top of it.

Similarly, we had to change the networking code. Instead of integers we are now sending entire BFV cipher-texts over the network, which are used to perform homomorphic operations. The *lattigo* library provides a handy way to serialize cipher-text, using functions in *marshaler.go* that allow us to encode a cipher-text in binary form, and decode it thereafter. The byte-array we obtain after encoding a cipher-text is sent over the network.

IV. Evaluation

To evaluate the performances of our code we benchmarked it's performances for the two versions. The first one without the generation of the beaver triplets, and the second one with the generation of the beaver triplets. The circuits we're benchmarking are different, first we chained multiplications between two peers, secondly we did the same thing for the addition. And for the second version we did the same with addition but this time increasing the number of peers. These tests were all repeated as many times as possible over 60 seconds (except the mutliplication because performance number were not reliable) using Golang's test package to automate the benchmark process for us[1]. We then ran these benchmarks all on the same computer. The circuits were all executed on the local machine (i.e via the local loopback) so the network operations are abnormally efficient. The conditions made it difficult to try this with something more realistic like a 50ms of delay between the peers.

A. Performance without beaver triplet generation

In Figure 2 we show the average runtime of the protocols as a function of the number of gates. As we expected the multiplication has a bigger impact on performance since there is a synchronisation step for each multiplication gate. With additions there is basically no difference because everything is being executed locally and that number of addition is not large enough to notice a difference in runtime.

B. Performance with beaver triplet generation

In Figure 3 we can see that the runtime is basically unchanged from the previous version which is expected since the logic of the code is basically unchanged, the variations can be explained by normal deviation from the real mean runtime, some randomness in compilation due to the existence of the rest of the code affecting caching, and some windows

[1]Ressource for what we used : https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go

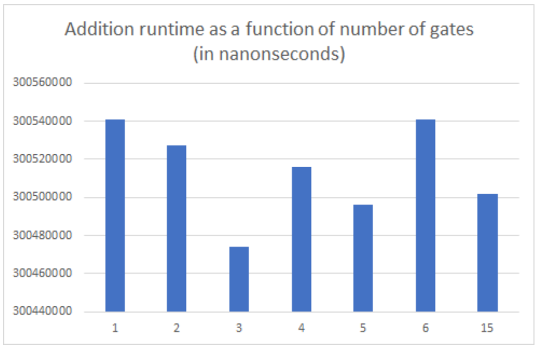

Fig. 3: Addition performance numbers with the final code

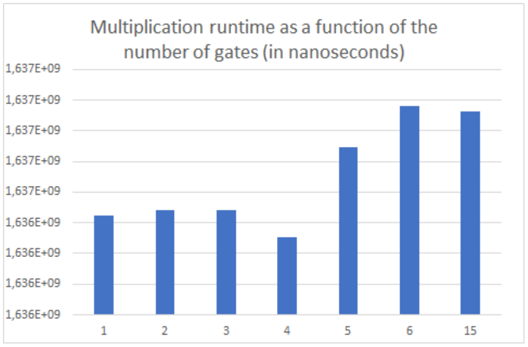

Fig. 4: Multiplication performance numbers with the final code

activities in the background and not enough executions to be close enough to the mean.

This is in stark contrast with Figure 4 where the runtime is much slower than the previous code, here we see that this runtime is most likely dominated by the costly encryptions and ciphertext manipulations as well as network operations. But this is to be expected since it is doing strictly more work then the previous part **and** operations like encryptions and ciphertext manipulations are expected to be costly.

Please note that this benchmark was executed for 10 minutes rather then 60 seconds because of abnormal trends being noticed. While the current results do not show a clear trend, suggesting that 10 minutes may not be enough this is already

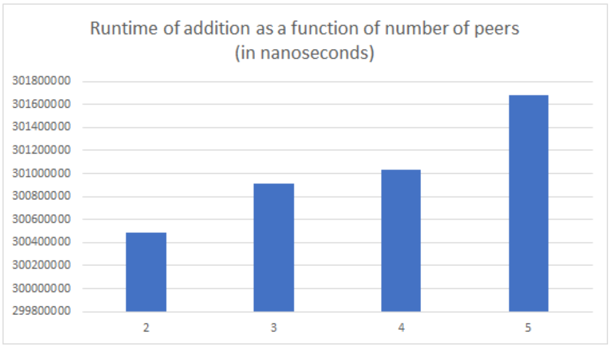

Fig. 5: Performance numbers with the final code compared to the number of peers

much better then before.

We didn't however expect not to find back the increasing trend we saw with the multiplications in the previous parts, we believe that this is a consequence of the encryption dominating the runtime. However the same trend can still be seen but it isn't as clear as before.

And finally, in Figure 5 we see that increasing the number of peers, also increases significantly the runtime of the program. This is to be expected since every broadcast that has to be performed means sending messages to all peers. It is to note here that the underlying circuit for this evaluation was made of additions. Since only 2 broadcasts are needed for that the performance hit is lowered.

V. Discussion

A. Performances

We were not surprised by the comparisons we got between the different settings we tested. We fully expected the runtimes to be dominated by either network operations or encryption operations. Our findings were as expected, the execution speed of this code is dominated by :

- Encryption operations
- Network operations

B. Threat model

The beaver triplet generation algorithm allows us to move from a threat model where all the peers are honest with a trusted 3rd party generating the triplets to simply having honest peers as long as the homomorphic encryption is truly secure. While we believe in the vast majority of cases having a 3rd party is not a realistic scenario there exists instances where the issues of this 3rd party can be mitigated by other factors[2]. A realistic case for triplet generation with an algorithm like the one we implemented could be hospitals trying to compute aggregate statistics without wanting to share their data and without putting the data safety in the hands of a 3rd party.

Also the threat model that this code operates under is that no malicious actor can accomplish a Man In The Middle scenario. This is because our network layer doesn't protect the integrity and authenticity of our packets and could lead to errors down the line. This is especially true with homomorphic encryption since an active attacker could manage to tamper with the triplets being generated or the last additions making the result impossible to verify.

VI. Conclusion

In this project we learned the details of a multi party computation engine. We also learned an algorithm for generating beaver triplets with homomorphic encryption. The implementation of addition and multiplications and the other operations allowed us to really learn all the details of this scheme. We also learned about the theoretic limitations, with choice of modulo for our applications and the implied trust in the peers (which is not adequate in every case).

[2]In Prio (Research paper :https://crypto.stanford.edu/prio/paper.pdf) the authors have a real application of beaver triplets without using such an algorithm for triplet generation