CS523 Project 1 Report

Wicky Simon, Nunes Silva Daniel Filipe

***Abstract*—This project consists in the implementation of N-party multiparty computations systems. Unlike a traditional approach, this aims to compute the result of a given circuit using users secret inputs without requiring them to reveal them explicitely. To achieve it, our Go program must be runned by each of the users who provide their secret values and the ciruit they want to compute together. Then, they share their additive secret sharings across the network they are linked with, parse the circuit, generate Beaver triplets when necessary and evaluate the result before retrieving it. We analyze the scenarios and the consequenting tradeoffs in which the users have access to a common trusted third-party and the one in which they have not. We make use of the Lattigo library for the cryptographic operations and the algebraic structure implementations it provides.**

I. Introduction

The aim of this project is to design, implement and assess two MPC engines using the Go programming language. First, two weeks are dedicated to the understanding of the general architecture, how to use it and how to tweak it to perform computations in a privacy preserving fashion. We implement the additive secret sharings split of the secrets, the circuit parsing, the corresponding gates, the Beaver triplets generation, update the network operations and describe our own complex circuit assuming the presence of a trusted third party during the two following weeks. Two more weeks to adapt our system so that it is able to generate Beaver triplets with no trusted third party but using BFV homomorphic encryption handled by Lattigo. Finally, we dedicate one week to revise our implementations, compare and evaluate them.

II. Part I

A. Threat model

The asset to protect for a party is its input. During the protocol, anyone monitoring the communication should not be able to recover the input of a given party. In particular, a party taking part in the computation can not infer more information about another party's input that what he could have infered if all data had been sent to a trusted third-party making the computation for them and sending them back the result.

All parties follow the "honest but curious" threat model. They do not deviate from the protocol and compute the correct circuit, but try to infer informations about other's input.

At the end of the computation, each party get the correct result of the circuit for the provided inputs.

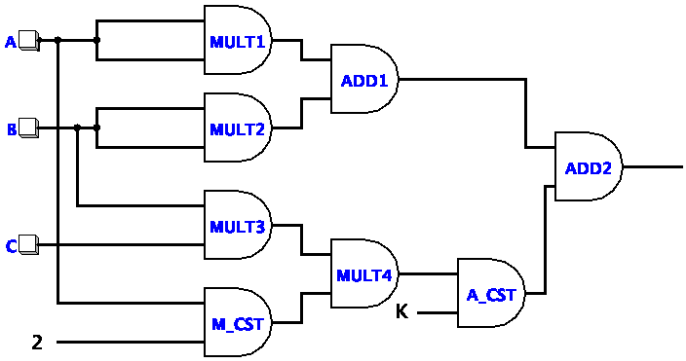
Furthermore, a third-party is trusted to generate the Beaver triplets, necessary for the computation of multiplications.

B. Implementation details

- *main.go* : This file is the entry point of the programm. It handles the creation of a party, the set up of the network, the circuit and launch the computation. The structure has been slightly adapted from the handout, to fit the changes. Note that since each party runs independently, this file is unused in the first part, Beaver triplets are generated in a single location.
- *helper.go* : This file contains two helper methods :
  - Pmod(x,mod) : computes x modulo mod. This method is necessary because the % operator can return negative results.
  - secret_share(secret,n) : Split a secret into n parts using additive sharing.
- *gates.go* : Implements the different gates needed for the circuits.
- *circuits.go* : Contains two methods :
  - SetUpMPC(circuit, trusted) : Create the necessary structure for the protocol to run, and generate the Beaver triplets depending on the chosen setting for *trusted*. This setting is used for modularity between Part 1 and Part 2. During this set up, the circuit is also parsed to determine the number of Beaver triplets to generate.
  - ComputeCircuit() : Uses the gates from *gates.go* to actually performs the computation, and return the result. The circuit must follow this structures :
    - Only one reveal gate is present, and it should be at the end.
    - The number of secrets must be equal to the number of inputs.
    - The output Wire ID must be sequential increasing numbers.
- *mpc.go* : This file is the heart of the protocol. Each party is represented as a MPCProtocol and contains an array of MPCRemote to abstract the other parties. The communication are performed with MPCMessage, containing various field needed. The *Run(trusted)* method works as follows :
  1) It checks if it has to generate Beaver triplets using the *trusted* value. This is mainly used for modularity between Part 1 and Part 2.
  2) The input is split into secret shares and sent.
  3) The shares are collected.
  4) The circuit is computed and the output is made available.
- The circuit added is inspired by the cosine theorem which states, for a triangle with sides a, b, and c with angle $\gamma = \widehat{ab}$, that $a^2 + b^2 - 2ab\cos\gamma = c^2$. To adapt this formula to the available gates and requirements, the following function is computed : $f(a, b, c) = a^2 + b^2 - 2abc + K$.

With random input 42, 3 and 5, f yields 448 as a result. The details of the computation are as in figure 1. To run tests for this part, make sure that *trusted* in *mpc_test.go:25* is set to true.

Fig. 1: Our own complex circuit



III. Part II

A. Threat model

The asset to protect for a party is its input. During the protocol, anyone monitoring the communication should not be able to recover the input of a given party. In particular, a party taking part in the computation can not infer more information about another party's input that what he could have infered if all data had been sent to a trusted third-party making the computation for them and sending the result.
All parties follow the "honest but curious" threat model. They do not deviate from the protocol and compute the correct circuit, but try to infer informations about other's input.
At the end of the computation, each party get the correct result of the circuit for the provided input.
In this part, no trusted third-party exists. The parties generate the Beaver triplets using homomorphic encryption.

B. Implementation details

The implementation of this part is identical to the previous part, with the addition of the *beaver.go* file which is used to augment the protocol as follows :

- Each MPCProtocol now contains a BeaverProtocol which is run before the computation of the circuit, if Beaver triplets are needed.
- BeaverProtocols communicates using BeaverMessage sent using the same port as MPCMessage. To avoid confusion, BeaverMessage are sent preceded by a 0 value, while MPCMessage are sent preceded by a 1 value.
- The protocol is then run and works according to the handout specifications.

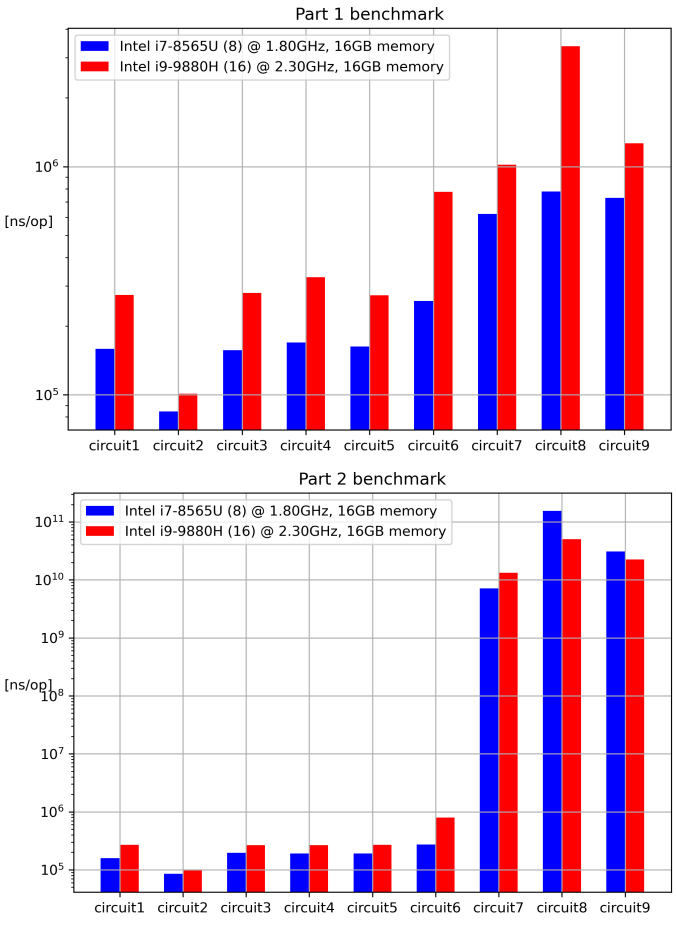
To run tests for this part, make sure that *trusted* in *mpc_test.go:25* is set to false.

IV. Evaluation

Our expectations about the two systems performance are the following :

- The addition, addition with a constant, subtraction and multiplication by a constant are gates which compute their respective output locally have low impact on performace. A quick glance at the code is necessary to see that they take constant time. The reveal operation as well as the multiplication which uses the reveal operation have to use the network to compute their ouput. Hence they are inherently slower and are directly linked the performance of the circuits.
- The difference between Part 1 and Part 2 is the Beaver triplet generation. Part 1 generates them in one place, before the main protocol, acting like a trusted third-party. In Part 2, every party collaborates to generate the triplets. This leads to Part 1 being relatively efficient whereas Part 2 is order of magnitude slower, since Beaver triplets generation requires homomorphic encryption and a lot of network traffic.

Fig. 2: Benchmarks on personal computers



We make use of the testing package provided by the Go programming language to perform benchmarks on

our own personal computers. It helps the programmer to analyze its code performance by running it multiple times and outputting robust measurements in a reasonable amount of time. By doing this, our expectations seem to be fulfilled according to the bar charts on figure 2. We run the benchmarks on each computer for each part and circuits of the project. It features a logaritmic scale that helps visualizing the order of magnitude of overhead due to the kind of system we implement. First, notice that simple circuits which do not require multiplications, on both parts, have a similar complexity. Then, we can compare on each part, simple and complex circuits. Concerning part 1, the time required to execute complex circuits seems reasonable considering their increased complexity. We should keep in mind that the longer a circuit takes to execute the less number of times it will be run for the benchmark, it means that longer measurements are less reliable than the short ones. Nevertheless, the trend is similar on both computers. Finally, we can confirm our concerns about orders of magnitude of overhead related to part 2 when used for multiplications. The additional homomorphic encryption and network traffic have a huge price both in computation and communication. We also notice that part 2 require a lot more memory than available on our computers and it could, therefore, be a bottleneck as well.

V. Discussion

This project and the way the two parts were compared illustrates a principle often told in class that privacy has its cost. At the end, we face the tradeoff between performance efficiency and privacy security. The performance overhead seems to grow exponentially comparing to the complexity of the system. In any setup, simple operations such as additions, additions by a constant or multiplication by a constant remain easy. It is a completely different story when its comes to slightly more complex operations such as multiplications. On one hand it is less trivial to implement it for the programmer even when one has access to cryptographic libraries such as Lattigo, there are still parameters to optimize otherwise the system can simply become unusable. On the other hand, the user also feels considerable side effects when using this kind of systems, especially the second one. It is not desirable for a user to drain its battery or burn its processor for such computation, in other words, the solution should not be worse than the initial problem. Considering our results, we would say that the technology is here and we know how to make use of it but it must fit the appropriate scenario. If there is a third party that we can trust according to the threat model, then considering the performance, we should use the system implemented in part 1. If we are in a more hostile scenario that does not allow to trust a third party then we should trust mathematics and implement the second system if we want to preserve privacy despite computation and communication overhead. This is particularly useful to ensure that a nasty trusted third party does not manipulate the Beaver triplets in order to change the final result according to what some adversary would like to achieve. This should be stated in the threat model and we should assess how critical our application is and how much we trust a potential third party to decide if it is meaningful to accept the overhead.

VI. Conclusion

This project is a good opportunity to have a first hands-on experience with the Go programming language and the Lattigo cryptographic library. It is enjoyable to work with modern tools offering a lot of packages that allow us to easily implement networks, tests and benchmarks. We implemented two MPC systems, one relying on a trusted third party and another one on homomorphic encryption. Then, we benchmarked and compared them. Finally, we discussed about each solutions and the corresponding tradeoffs. We are now able to see concretely how the threat model is central and take design decisions accordingly. Now that we implemented it once ourself, talking about tradeoffs and performance overhead make more sense since we were able to assess different scenarios and choose the appropriate solution