CS523 Project 1 Report

Eric Michel Alexandre Jollès, Ugo Damiano

I. Introduction

The goal of this project is to implement an additive secret sharing protocol in Go programming language. To provide operations such as multiplication we need the parties to establish secrets beforehand. This principle will guide our implementation. In Part I (section II) we will assume that some trusted third-party produces the secrets. In part II (section III) we will implement a new protocol to avoid the use of the third-party, through the means of homomorphic encryption.

II. Part I

A. Threat model

The adversary :

- cannot corrupt the third party that generates the triplets for the multiplication gates
- can only access the communications between peers
- cannot corrupt any of the parties

Since a party broadcasts all of his shares except his, an adversary cannot reconstruct a secret. In order to avoid the beaver triplets being reconstructed by the adversary, and secrets being leaked, the third party shouldn't be corrupted. If any of the parties is corrupted, then the secret of that party is leaked but the secret of the other parties isn't. Note that this doesn't hold if the adversary can corrupt all parties except one since it allows the adversary to compute the secret of the last party based on the protocol output.

B. Implementation details

Our implementation of MPC supports basic arithmetic circuits containing numerous gates such as addition or multiplication gates. Each multiplication gate needs a Beaver triplet to operate, which is split between the different parties. As a first step the trusted third party counts how many multiplication gates there are in the circuit. It creates a Beaver triplet for each gate and shares it across the different parties when declaring a new protocol with the appropriate function.

One instance of the protocol keeps track of its shares of the triplets, the shares received from the other peers, its own input, the specification of the other peers such as IP address, ID and port. When the protocol is run, it connects to the network and broadcasts the shares of its own secret. Once this is done it reads from the channel the shares from the other peers.

Every time gates like the reveal and multiplication gate must be evaluated, the previous steps are executed. After recovering the shares, that are computed for a specific gate, it can be evaluated.

If during the evaluation of a gate two messages are received from the same peer the second message is put at the end of a channel and read later. Here we assume, and so does our implementation, that the parties evaluate the gates in the same order. Hence, a party receives the messages of the other in the same order as it uses them. The order in which the messages are sent is preserved by the fact that a TCP connection is used.

To evaluate the circuit, we compute the result of each gate and store it into a map that relates the gate ID and the output value. We assume that when evaluating a gate all previous dependencies have been evaluated. As an example, consider a circuit that contains a simple two input adder. We assume that the two input gates are listed before the addition gate, thus we evaluate first the two inputs before the addition, hence their results are already available.

Lastly for all gates that require a constant to be added, the constant is added to the gate output only by the peer with the smallest ID.

C. Circuit design

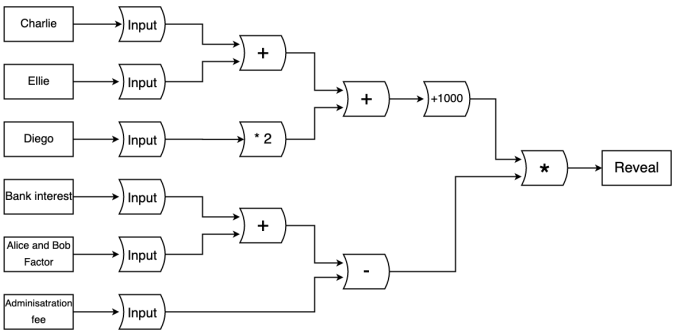

Fig. 1: An example of a circuit implementation.

We are going to present you our circuit through a concrete example of its use.

Alice, Bob and their friends: Charlie, Diego and Ellie, want to open a bank account for Alice and Bob's child. In order to avoid endless grudges due to each other relation to money, they decided to use our additive secret sharing protocol.

- Charlie, Diego and Ellie put money in the account.
- The grandma Constantine has a lot of money so she reveals that she will put 1000$ on the account.
- Diego has a bank account at the bank so the amount he puts is multiplied by two.
- Alice and Bob say they will multiply the amount on the account and so does the bank. Since the bank doesn't want to show how much financial power it has, it decides to input secretly its participation.
- Every year a fee must be paid on the account. Since the administration also wants to keep secret how much it will be taking it takes part in the protocol.

The final circuit is as follow and shown in figure 1 :

$$(1000 + Charlie + 2 \times Diego + Ellie) \times (Bank\ interest + Alice\ and\ Bob\ factor - Administration\ fee)$$

III. Part II

A. Threat model

The adversary :

- can only access the communications between parties
- cannot corrupt any of the parties

Here the conclusion is the same as in section II with a minor change. The homomorphic encryption scheme removes the need to trust a third party and the assumptions that came with it.

B. Implementation details

For the evaluation of the circuit, the protocol stays the same. However, in order to establish Beaver triplets, we use homomorphic encryption with BFV.

The two protocols are run one after the other. Namely, the protocol for the triplet generation is run and its output values are then passed as input to the MPC protocol.

Triplets are generated by taking each component of the a, b, c vectors established during the protocol. This means that the degree of the polynomials used in this phase is the limit in the number of multiplication gates a circuit can have. It would be possible to run the protocol twice in order to accommodate for a larger number of multiplication gates. However, the actual degree allows already enough freedom and this would require to adapt the implementation with further details that didn't seem relevant to us (i.e. the degree in our implementation is 8192).

IV. Evaluation

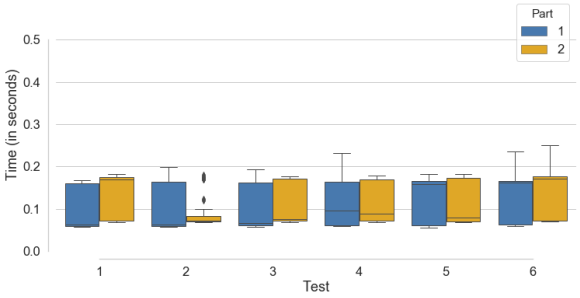

Fig. 2: Runtime (in seconds) for given tests 1 to 6 (tests without multiplication gates) over 50 runs.

The number of peers should directly impacts performance. Since the number of messages sent by each party is of the order $O(n)$ for a sharing phase, where n is the number of parties, the total number of messages is in the order of $O(n^2)$.

We expect the following. The gates that affect performance the most are the ones that rely heavily on networking to establish shared secrets. There are two such gates:

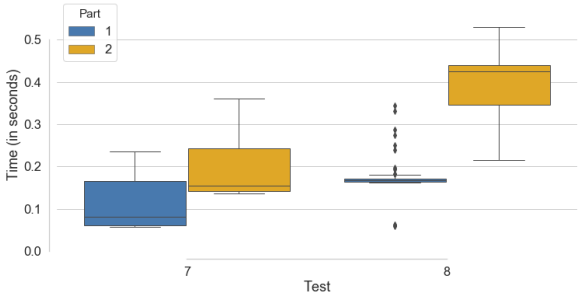

Fig. 3: Runtime (in seconds) for given tests 7 and 8 (tests with multiplication gates) over 50 runs.

- The reveal gate,
- The multiplication gate.

Since there is exactly one reveal gate per circuit we can safely remove it from our comparison. Each multiplication gate adds two sharing phases.

Moreover, when we add the triplets generation in part II, the "communication penalty" of the multiplication gates increases again due to the communications to establish the triplets.

The number of sharing phases is summarized in table I.

The total number of sharing phases can be computed as follows, where m is the number of multiplications :

$$1 + 1 + 2 \times m + 1_{m>0}$$

Note that the last term appears only in part II. A final remark here is that the third party in our implementation is done locally and doesn't uses the network to share the triplets. Making this third party share the triplets via TCP connections would reduce the gap between part I and II, but part I would still be less communication intensive as **only** the third party must send $O(n)$ messages to the other peers in the triplets generation phase.

Gates	Inputs	Reveal	Multiplication	Others
Sharing phases	0*	1	2*	0

TABLE I: Gates that introduce sharing phases and how many. * means that if at least one of this type of gate is used the total number of phases for the circuit increases by 1.

However, we can see in figure 2 that we don't have a big difference between a circuit with 2 peers (Test 2) and with 4 peers (Test 6). It is important to note that all tests were executed on the same host thus making the communication overhead negligible.

We can see in figure 4 that the homomorphic encryption scheme introduces a huge overhead that scales with the number of peers (on the figure 3 we have 3 peers for test 7 and 5 peers for test 8). There might be different explanations. It could be related to the ciphertext we used between peers in the Beaver's triple generation protocol. Indeed it contains thousands of bytes, whereas we send only integers in part I. Another reason would be the amount of computations. We

have to account for the large amount of additions, multiplications, encryptions and decryptions of the messages.

V. Discussion

An interesting part of privacy preserving schemes is how using a third party becomes tricky. Here, it is easy to see that in the first implementation, the third party can easily recover some intermediate results as long as the circuit includes multiplications.

As an example, assume we have a simple circuit with only a multiplication gate and its appropriate inputs. Then, at some point the first party sends $x_1 - a_1$ where x is the first party input and same goes for the second party. If the third party can see these messages then it can compute $(x_1 - a_1) + (x_2 - a_2) + a \mod p = x$. The role of the a is to obfuscate x. Same happens for y and the the third party recovers the two secrets.

Since this is the case, the first implementation is useless, as everyone might as well send their share to the third party which computes the result and sends it back. In no case should the third party be able to access the messages of the second protocol if it is not fully trusted. So clearly, if we take this approach, communications have to be encrypted. Also note that the "forward privacy" of the protocol is dependent on the forward secrecy of the encryption.

We should note that the above example changes a bit the trust model. Since the third party would have access to *all* inputs and is trusted to compute the whole circuit correctly. In the original assumption the third party can only manipulate the triplets and obtain the operands that are related to multiplications.

As told before if the third party cannot be trusted then if the messages of the protocol aren't encrypted it can recover some results. Part II can perform the additive secret sharing without encrypting the protocol messages. Since no one knows all the shares of the triplets, nobody can recover the secrets. Clearly homomorphic encryption allows the full protocol to work without any outside intervention.

The first part has better utility in a network with low bandwidth as said before. We could also imagine that the third party bootstraps the protocol by providing the circuit structure along with the triplets after the parties agree on the computation to be performed.

The scheme using homomorphic encryption needs a better network and could be used in a more adversarial environment where no third party can be established. And if the system is not timing critical.

VI. Conclusion

This project enlightens two important points. First, such privacy preserving schemes can be implemented and are not to difficult to realize. The second point is that these schemes are quickly limited and need the addition of other tools in order to keep the threat model simple. However, this comes with disadvantages. Sometimes the solution, here homomorphic encryption, is also limited and extreme care must be taken in usage and implementation. Also such schemes come with high overhead. As always we need to make a trade-off.

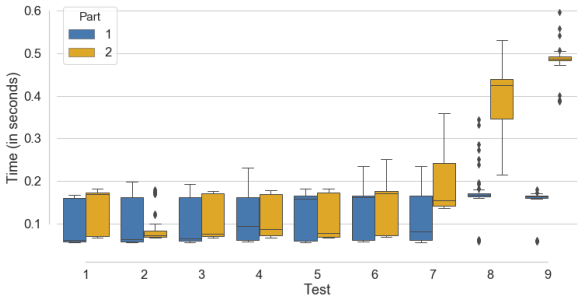

Fig. 4: Runtime (in seconds) for all given tests (tests 1-8) and our circuit (test 9) over 50 runs.