Multi Party Computation - Active Adversaries

Computer Science Lab
July 26, 2015

Vitor Enes, José Bacelar Almeida, and Bernardo Portela

University of Minho, Portugal

Abstract. This project aimed at implementing SPDZ - a multi party computation protocol secure against active adversaries. SPDZ belongs to the family of protocols in the preprocessing model. These protocols push all the expensive public-key machinery to the preprocessing phase. Hence, in the online phase, only cheap primitives are used, which means extremely efficient computations.
This work has focused on the online phase, relying on a trusted third-party to generate the required data from the preprocessing phase.

Keywords: multi party computation, SPDZ

1 Introduction

Although secure multiparty computation was invented almost thirty years ago, only in the recent years these protocols were implemented and tested in practice. They can be divided in two camps essentially: based on Yao circuits or based on secret sharing. The ones based on Yao circuits are mainly focused on two party computations, but the ones based on secret sharing can be applied to a more general number of players.

The protocols based on secret sharing can be divided on those which consider only honest-but-curious adversaries and those which consider active adversaries. The latter are often presented in the preprocessing model in which is possible to produce random data that will be consumed during the online phase. SPDZ fits this model.

The goal of this project was to implement SPDZ. Since the online phase could be implemented assuming a trusted dealer, we started with it. This dealer provides the preprocessed data that would be generated in the offline phase. In this report we will explain how we did it.

Report structure Bearing this in mind, we will firstly introduce the concept of multi party computation in Section 2; In Section 3, we introduce the theory behind SPDZ; The explanation of the system's architecture and the implementation choices are present in Section 4; In Section 5, a final appreciation of the work will be made, together with some suggestions for improvement.

2 Multi Party Computation

Multi party computation (MPC) can be defined as the problem of n players computing an agreed function in a secure way. The inputs of this function are provided by the players and security means guaranteeing the correctness of the output as well as the privacy of the players' inputs, even when some players cheat. Concretely, we assume we have inputs $x_1, \ldots, x_n$ and the player P_i knows only x_i. We want to compute $f(x_i, \ldots, x_n) = (y_1, \ldots, y_n)$ such that P_i is guaranteed to learn y_i and nothing more.

A classical MPC problem is the Yao's Millionaires' Problem: two millionaires meet in the street and want find out which one is richer. Is this possible without revealing how much money each one has? In this case f is simply the comparison between two integers. In the end, they will know who has more money, but this should all the information revealed.

In our case, f will be an arithmetic circuit, and x_i the inputs of this circuit. Let $z = y_1 + \cdots + y_n$ be the output of this circuit. In the end, we will open z. This means, that P_i besides knowing y_i, it will know z. But this opening will be done in a secure way, guaranteeing that the players did not cheat during the execution of the protocol.

3 SPDZ

As stated before, SPDZ is divided in two phases: offline and online. One key aspect of the offline phase is that it can occur without knowing neither the function to be computed[1] nor the inputs. This is good because both may not be known until the online phases starts. This asynchrony allows us to have very efficient and secure computations.

The protocol also supports full reactive computations: after one function is evaluated, another can be executed depending on the output of the first. This can go forever until the preprocessed data runs out.

Before we proceed, we shall first introduce some notations. Let n be the number of players and $\mathbb{F}_q$ the finite field over which we will perform computations. Each player P_i has a share $\alpha_i \in \mathbb{F}_q$ of a secret shared value $\alpha = \alpha_1 + \cdots + \alpha_n$. This α is known as the fixed MAC key. Each player P_i has also a secret key β_i.

$[x]$ $x \in \mathbb{F}_q$ is [.]-shared if P_i holds a tuple $(x_i, \gamma(x)_i)$ where x_i is an additive secret sharing of x, i.e. $x = x_1 + \cdots + x_n$, and $\gamma(x)_i$ is an additive secret sharing of $\gamma(x) := \alpha.x$, i.e.

$$\gamma(x) = \gamma(x)_1 + \cdots + \gamma(x)_n$$

$[[x]]$ $x \in \mathbb{F}_q$ is [[.]]-shared if P_i holds a tuple $(x_i, \gamma_1(x)_i, \ldots, \gamma_n(x)_i)$ where x_i is an additive secret sharing of x and $\forall k \in [1, n] : \gamma_k(x)_i$ is an additive secret sharing of $\gamma_k(x) := \beta_k.x$, i.e.

$$\gamma_k(x) = \gamma_k(x)_1 + \cdots + \gamma_k(x)_n$$

[1] It does not need the whole function description, but it needs some function related data, such as the number of multiplication gates.

3.1 Offline Phase

The goal of this phase is to produce raw material to the online phase. With this, functions in the online phase can be computed more efficiently. In our implementation, this phase is replaced by a dealer that provides the data needed.

If $rnd(x)$ returns a random $y \in \mathbb{F}_q$, then given an arbitrary x, the dealer can generate a tuple with shares of x with

$$(rnd(x), rnd(x), \ldots, x - \textstyle\sum_{i=1}^{n-1} x_i)$$

In order to multiply secret shared values during the online phase we will use Beaver's multiplication triples, i.e. shares of random values $[a]$, $[b]$, $[c]$ such that $c = a.b$. These triples will be supplied by the dealer. As we will see further in the document, to achieve commitment functionalities and to do batch-MAC-check of opened values, the dealer will also have to deliver $[[.]]$-shared values to players. In appendix A can be found all the data the dealer has to provide.

3.2 Online Phase

Although it's possible to compute other arithmetic operations, our online phase only supports addition and multiplication. These operations can be applied to open values and to shared values. They can also be performed on one value which is shared and on one value which is opened resulting in a new shared value.

$$[x] + y = \begin{cases} (x_i + y, \gamma(x)_i + \alpha_i.y) & \text{if } i = 1, \\ (x_i, \gamma(x)_i + \alpha_i.y) & \text{if } i \neq 1 \end{cases}$$

$$[x] + [y] = (x_i + y_i, \gamma(x)_i + \gamma(y)_i)$$
$$[x].y = (x_i.y, \gamma(x)_i.y)$$

The operations presented above can be computed locally. But if we want to multiply secret shared values we need the parties to interact. In order to compute $[x].[y]$ we take a precomputed multiplication triple $\{[a], [b], [c]\}$ and each party calculates:

$$[d] = [x] - [a]$$
$$[e] = [y] - [b]$$

After this, players run open($[d]$) and open($[e]$) and each player now has the opened values d and e.

$$[x].[y] = [d].e + [a].e + [b].d + [c]$$

$$\begin{aligned} x.y &= d_1.e + a_1.e + b_1.d + c_1 + \cdots + d_n.e + a_n.e + b_n.d + c_n \\ &= d.e + a.e + b.d + c \\ &= (d + a).(e + b) \\ &= x.y \end{aligned}$$

The protocol open of a $[.]$-shared value mentioned above can be found on appendix B along with the open of a $[[.]]$-shared value and the commit protocol we used. Note that

the open used in the multiplication is not safe and that is why we do a batch-MAC-check later, contrarily to the open of a [[.]]-shared value, which can be done safely. We could remove all [.]-shared values of the protocol and only use [[.]]-shared values, but this would be really heavy in terms of data stored and rounds of communication. Since we can do a batch-MAC-check before opening sensible data, the approach used is not a problem.

Batch-MAC-check Each player has $[x_0], [x_1], \ldots, [x_t]$ and a set of opened values $\{x'_0, x'_1, \ldots, x'_t\}$ and we want to check whether $x_i = x'_i$. Given random $e_0, \ldots, e_t$ (we will discuss how to chose these e_i), each party computes locally:

$$\begin{gathered}[y] = [x_0].e_0 + \cdots + [x_t].e_t \\ y' = x'_0.e_0 + \cdots + x'_t.e_t \\ d_i = \alpha_i.y' - \gamma(y)_i\end{gathered}$$

- P_i commits to d_i with $[[r_i]]$
- P_i runs open($[[r_i]]$)
- P_i knows $\forall i \in [1, n] : d_i$ and computes $d = \sum_{i=1}^{n} d_i$
- accept if $d = 0$

$$\begin{aligned}\sum_{i=1}^{n} d_i &= \alpha_1.y' - \gamma(y)_1 + \cdots + \alpha_n.y' - \gamma(y)_n \\ &= \alpha_1.y' + \cdots + \alpha_n.y' - (\gamma(y)_1 + \cdots + \gamma(y)_n) \\ &= \alpha.y' - \gamma(y)\end{aligned}$$

Since $\gamma(y) := \alpha.y$, only if $y = y'$, d will be zero (there is a probability at most of $1/q$ of accepting when $y \neq y'$).

The problem with the random e_i still remains. The dealer could supply $\forall i \in [1, t] : [[e_i]]$ but this would mean an increase of spatial complexity in the protocol. Instead we will use only one $[[u]]$. When needed, the parties run open($[[u]]$) (which is done safely) and define $e_i = u^i$. Since $e_i = e_{i-1}.u$, this is very efficient.

4 Implementation

To implement SPDZ is not enough to understand the cryptographic theory behind. There is an implicit distributed computing problem. There were several paradigms we took into consideration to solve it, such as Multi-Threaded, Event-Driven and Message-Oriented. We chose the first because it was the one we felt more comfortable with.

After a discussion with professor Paulo Sérgio Almeida, we came up with a possible algorithm. The idea was to attribute to each gate a number of local dependencies and a number of distributed dependencies (the number of messages the player has to receive in order to be able to evaluate the gate). Both addition and multiplication gates would have as local dependencies the number of the inputs of the gate. The addition gate would have zero distributed dependencies. The multiplication gate would have $n-1$ distributed dependencies. Then, to evaluate the circuit, there would be an arbitrary number of workers that would have two types of tasks: evaluate a gate or send messages to players. These workers would take tasks from a bounded buffer. If an input of some gate is calculated, we subtract by one the number of its local dependencies. If the player receives a message related to some gate, we subtract by one the number of its distributed dependencies. If the number of local

dependencies of a multiplication gate is zero, we would put in the bounded buffer a task to send a message to all players with the opening of the d and e relating to this gate. If the sum of both dependencies is zero, we would put in the bounded buffer a task to evaluate the gate. Eventually all gates would be evaluated and consequentially, the circuit too.

Another option would be go through each gate of the circuit (one by one, with all players following the same order) and evaluate it. This is simpler to implement because there is only one thread doing the whole work sequentially. Since our test environment would be our machine and the first solution means more threads (if we have more than one worker), it would not increase the performance of the protocol (it might even worsen it). Besides this, the first solution solution would probably only compensate in circuits with a number of additions way superior to the number of multiplications. The reason for this is that the computation of the circuit has to stop when we evaluate a multiplication in order to all parties synchronize. For all the reasons presented, we chose the second option.

We have created two commands: one to run players and one to run the dealer.

```
player --port=3000
player --port=3001
player --port=3002

dealer --circuit-inputs-number=200000 \
       --players=localhost:3000,localhost:3001,localhost:3002
dealer --circuit=/path/to/circuit \
       --players=localhost:3000,localhost:3001,localhost:3002
```

After starting the players, they wait until the dealer sends them the data needed to compute the circuit and which circuit to compute. As we can see from the examples, it is possible to pass a circuit as an argument (the syntax used can be found on appendix C) or ask the dealer to generate one with a given number of inputs. The dealer also generates random inputs and then sends the shares to the players. This only makes sense as a proof of concept of SPDZ's online phase. Ideally, there should be some protocol where players contribute with inputs and distribute shares between them.

As soon all players have all the data provided by the dealer they start evaluating the circuit. Additions are evaluated locally. To evaluate multiplications, player have to exchange messages. We used JSON as a serialization format. After calculating a multiplication, P_i stores (d_i, d) and (e_i, e). In the end, we have the whole circuit evaluated and as an output, a secret shared value. Before opening it, we batch-MAC-check all (d_i, d) and (e_i, e) using the protocol already described. If all checks, the players open the circuit's output.

Results All the tests were run locally in a machine with 8 processors Intel i7-3612QM 2.10GHz, 8GB RAM running Ubuntu 14.04.2 LTS x86_64. The tests were executed 3 times for each row in the following table. The dealer generated random circuits with the given number of inputs and the online phase was run with three players.

Inputs	**Offline phase**	**Online phase**
10000	1.7s	1s
100000	14.9s	11s
200000	31.3s	22.7s
400000	71.2s	48.7s

5 Conclusions and Future Work

This project was successful in implementing the online phase of SPDZ protocol. The source code of the implementation can be found on GitHub. The next step is remove the need of a trusted dealer, implementing the offline phase of the protocol.

It would be good to see it deployed in different machines evaluating a well known circuit to see how our implementation handles real world networks and then compare it with benchmarks of other implementations. The achieve acceptable results we would probably need to use a lighter serialization format than JSON, maybe Google's protocol buffers. And if we have the protocol deployed in different machines, it makes sense to implement the system's architecture with workers and compare both approaches.

Our implementation of SPDZ only supports two type of gates. In the future, if we want evaluate more complex circuits, we will need to support other gates, such as bit-decomposition.

References

1. Ivan Damgård and Ronald Cramer. Multiparty computation, an introduction. In *Contemporary Cryptology*, pages 41–85, 2005.
2. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pages 1–18, 2013.
3. Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 549–560, 2013.

A Data from dealer

Before generating any data, the dealer will be given a circuit or be asked to generate a random circuit given a number of inputs. Let m be the number of inputs of the circuit. The dealer will generate random $x_1, \ldots, x_m$ that will be used as inputs of the circuit. But these are not the values that are sent to players. First, it generates $\alpha \in \mathbb{F}_q$, the fixed MAC key. With this, it is possible to have $[x_1], \ldots, [x_m]$.

Since the dealer has the circuit, it knows how many multiplication gates there are. Hence, it knows how many Beaver's multiplication gates will be needed. Although it would be possible to generate data independently of the circuit to be computed, that does not happen in our implementation. The dealer only delivers the data that will be needed in the online phase.

We only batch-MAC-check once. To do a linear combination of all [.]-shared that were opened during the computation of the circuit, we use $[[u]]$. Besides this, P_i has to commit to a certain value using $[[r_i]]$ (we assume the dealer also delivered r_i to P_i). This means $n + 1$ $[[.]]$-shared values have to be provided by the dealer.

B Protocols

B.1 Open a [.]-shared value

open($[x]$):
- P_i send x_i to all other players
- P_i computes $x = x_i + \cdots + x_n$

B.2 Open a [[.]]-shared value

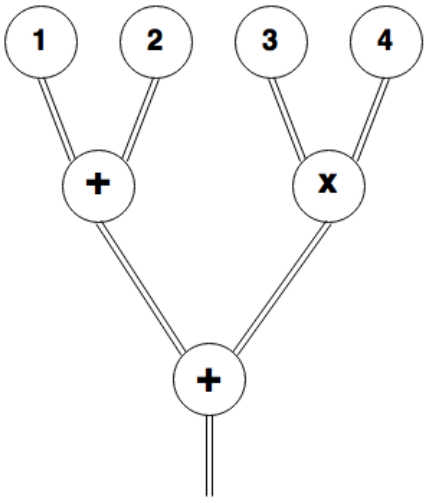
open($[[x]]$):
- P_i sends x_i to all other players
- P_i computes $x' = x_i + \cdots + x_n$
- $\forall k \in [1, n] \wedge k \neq i$, P_i sends to P_k $\gamma_k(x)_i$
- P_i computes $d_i = \beta_i.x' - \sum_{l=1}^{n} \gamma_i(x)_l$
- since $d_i = \beta_i.x' - \gamma_i(x)$, if $d_i = 0$ then $x' = x$

B.3 Commit to a value using a [[.]]-shared value

commit(x):
- we assume the dealer delivered $[[r]]$ to all players and r to player who wants to commit (e.g. P_1)
- P_1 sends $s = x - r$ to all other players
- when P_1 wants to open x, all players run open([[r]])
- P_i computes $x = s + r$

C Circuit's Syntax

```
4
+ 1 2
x 3 4
+ 5 6
```



In the first line we indicate the number of inputs. The following lines contain the type of gate (addition or multiplication) and its inputs. This syntax does not support the indication of the outputs gates because we assume that there is only one output, i.e. we only work with circuits that form a tree.