Multi Party Computation - Active Adversaries

Computer Science Lab
July 11, 2015

Vitor Enes, José Bacelar Almeida, and Bernardo Portela

University of Minho, Portugal

Abstract. This project aimed at implementing a secure MPC protocol against active adversaries.
We implemented a protocol called SPDZ that belongs to the family of protocols in the preprocessing model. This protocol pushes all the expensive public-key machinery to the preprocessing phase. Then, in the online phase, it only uses cheap primitives, which gives us extremely efficient computations. In addition, this phase is actively secure against a dishonest majority of corrupted players.
This work is focused on the online 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 and provides full active security against a dishonest majority.

The goal of this project is to implement the online phase of SPDZ assuming a trusted dealer. This dealer provides the preprocessed data that would be generated in the offline phase.

Report structure. Bearing this in mind, this report will fistly address the theory behind this work in section 2; It will then in 3 explain the system's architecture and implementation choices; And, in section 4 a final appreciation of the work will be made, together with some suggestions for improvement.

2 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 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.

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

2.1 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.

2.2 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:

$$\begin{aligned}[d] &= [x] - [a] \\ [e] &= [y] - [b]\end{aligned}$$

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 A 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 use only [[.]]-shared values, but this would be really heavy in terms of data stored and rounds of communications. 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:

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

- 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 even if $y \neq y'$).

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.

3 Implementation

To implement SPDZ, it 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.

After a talk 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.

Even though this solution seemed perfect, we opt for another one: go through each gate of the circuit sequentially and evaluate it. The first 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. This along with the complexity the first solution would be bring to the protocol was enough to convince us.

talk about the dealer after provides the triples and [[.]]-shared values, and how the circuit is read, after doing these alterations in the implementation.

Now we start evaluating the circuit. When a multiplication is calculated, 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.

4 Conclusions and Future Work

References

A Protocols

A.1 Open a [.]-shared value

open($[x]$):

- P_i send x_i to all other players
- P_i computes $x = x_i + \cdots + x_n$

A.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 $x' = x$

A.3 Commit 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 $x = s + r$