Implementing Secure Multi-Party Shuffling in MpcLib

Aditya Sinha

May 6, 2016

Abstract. Although significant progress has been made with respect to the theory of secure multi party computation, there has been little practical application of its techniques. For this project, I have implemented a framework for simulating the secure computation for arbitrary circuits using quorums in MpcLib. Using this framework, I implemented a protocol to perform secure multi-party shuffling, where a number of parties, each holding a secret value, want to compute a permutation of their values without revealing the owner of each value. By measuring the performance of the secure multi-party shuffling protocol on small number of parties, we can draw conclusions about the performance and specific shortcomings of existing secure multi-party computation protocols.

1. Introduction

A secure multi-party shuffling is a type of multi-party computation, in which each party in a group holds an input value, and the parties want to compute a random permutation of the inputs. For the shuffling to be secure, parties should not be able to guess the party that provided an input with better than a random guess. Secure multi-party shuffling can be an extremely useful primitive for achieving privacy in situations where the inputs themselves are not private information, but the mapping from parties to inputs is sensitive information. For example, two situations are electronic voting\[Nef01\] and secure auctions\[SA99\].

The goal of this project was to implement the secure multi-party shuffling algorithm described in \[MSZ14\] to understand the bottlenecks that hamper the adoption of secure multi-party computation in practice.

Notation

We will let $\mathbb{F}_p$ denote a field with $p$ elements, where $p$ is a prime number. All computations are done in this field unless otherwise stated.

2. Overview of Secure Multi-Party Shuffling Algorithm

2.1. Quorum Building

We assume that we have set of $O(\log n)$ parties in which fewer that $1/3$ of the parties are malicious. We have this set of parties securely generate a random number using a simple scheme in which each party locally generate a random number and then shares the number with all other parties. All parties add the shares locally, then reconstruct the shared value. If at least one of the parties in the protocol was honest, then the resulting number is random. This
random number is broadcasted by every party in the set to every party in the computation. The parties receive a number from each party in the set and use majority filtering to determine the random number. They use this as a seed for a PRNG, whose outputs are used to determine which parties are assigned to each quorum. For example, the first \( \log n \) numbers output by the PRNG correspond to the ID’s of the parties in the first quorum.

2.2. Sorting Network

In order to sort the values, I implemented the sorting network presented by Leighton and Plaxton [LP90]. Leighton and Plaxton describe an algorithm that can be used to deterministically create a sorting circuit with depth \( 7.44 \log n \). The algorithm is based on the idea that a simple tournament, in which each round creates a “winners” and a “losers” bracket is reasonably good at sorting a set of inputs.

2.3. Sorting Network Evaluation

Every gate in the circuit is assigned to a quorum. Initially, every party associates its secret input with a random number. Using Shamir’s polynomial secret sharing scheme, each party secret shares both the random number and its secret input with quorum assigned to the first gate its input must pass through. When a quorum receives all of the inputs necessary to calculate a gate, it will evaluate the compare and swap gate, secretly comparing the random shares, and using the result to swap both the random shares and the shares of the secret input. Then, the quorum will reshare the output with quorum assigned to the next gate. Finally, once the circuit has been evaluated, the quorums associated with the last gates reconstruct the secret inputs, which are now shuffled, and use a majority filtering protocol to broadcast their results to all of the other quorums.

2.4. Quorum Share Renewal

Share renewal is necessary for sharing a secret value from one quorum to another quorum. If the shares held by the first quorum and the second quorum could be used in combination to reconstruct the secret, then a set of malicious parties in the first quorum could work in conjunction with a set of malicious parties in the second quorum to reconstruct the secret. The quorum share renewal process ensures that the second quorum holds shares on a new random polynomial, so shares in the first quorum and those in the second quorum cannot be used in conjunction. The protocol used is described in [MSZ14], which itself is an adaptation of the share multiplication algorithm presented in [GRR98]. Therefore, we can also use the same quorum share renewal algorithm for the multiplication of secret shared values. This algorithm is not fully implemented for the Byzantine case; however, in the simulation, we increase the number of messages sent, the round count, and the total data sent to simulate the results of the Byzantine case.

3. Deliverables Implemented

- **Non-Cyclic Circuit Evaluation** The framework created for running the sorting network is general enough to handle the computation of any non-cyclic circuit. The code can be found in MpcLib/Libs/Circuit

- **Sorting Circuit Creation** The code for the sorting network creation and evaluation can be found under MpcLib/Libs/Circuit.
• **Secure Compare and Swap Protocol** The code for the compare and swap protocol and all associated sub-protocols can be found in *MpcLib/Libs/BasicProtocols* and under *MpcLib/Libs/QuorumSharing*.

• **Testing code** Examples for how to run the full multi-party shuffling protocol, associated sub-protocols, generate a sorting circuit can be found in *MpcLib/Libs/TestApp*.

All of the code can be found at [https://github.com/mahdiz/MpcLib/](https://github.com/mahdiz/MpcLib/). For a list and short description of all of the protocols implemented, see appendix A.

### 4. Simulation Framework

The simulations were performed on a single-threaded event driven simulator, called the *NetSimulator* part of which was already part of MpcLib. Parties are registered with the *Net Simulator*. When the simulation starts, each party begins executing a Protocol. Protocols are allowed to spawn multiple subprotocols all of which are registered with the executing party. This allows the party to handle demultiplexing of messages. Every protocol must implement a *Start* method and a *HandleMessage* method. When a message is sent to another party, the delivery timestamp is at least one greater than the current timestamp of the simulator. The *NetSimulator* handles all messages for a given timestamp before moving on to the next timestamp. Some messages do not incur a one round delay, namely when a subprotocol completes for a party, a loopback message is sent to its parent protocol, which does not incur a one round delay.

#### 4.1. Simulation of an Arbitrary Circuit

As mentioned above, the framework that was created to simulate multi-party shuffling is able to compute any non-cyclic circuit using quorums. In order to simulate a circuit, the user must create a protocol factory that can produce a protocol that simulates a particular gate a gate type and inputs. The user must run a SecureMultiQuorumCircuitEvaluation for every party for every quorum of which the party is a member. The user should run this inside a SynchronizationProtocol to ensure that the circuit completes at the same time for all parties.

### 5. Results

It was infeasible to use verifiable secret sharing to calculate an entire circuit. This is because the VSS algorithm took over an hour and a half minutes to simulate a single secret comparison (versus about 10 seconds for comparison without VSS) when using a 160-bit prime. Therefore, are usually most concerned with round count and communication cost, in this case the processing cost of VSS is prohibitive. Therefore, the simulation can only simulate the communication cost of the algorithm.
Round Count vs. Field Size

Message Count (30 bit field size)
The first and main item of note is the extremely large communication. For example, 8 parties with a 50-bit prime number required over 100GB of communication. While protocols might scale well asymptotically, the constant term is extremely large. Most of the communication cost was not actually performed, but was rather simulated because it stems from the cost of reliable broadcast. The simulated costs are derived from \[\text{[CKS05]}\], in which Cachin et al. present a cryptographic broadcast protocol. Still though, the cost of reliable broadcast is far too high to be practical for protocols that make use in as a primitive.

Second, the round count is approximately linear in the key size. This is due to the implementation of the prefix-OR operation.\(^1\) The prefix-OR operation is naively implemented by disjuncting the bits one by one. This strategy is infeasible, however, if we need to use cryptographic VSS, because the prime length is 160 bits. Damgaard et al. \[\text{[DFK+06]}\] present a protocol for calculating the prefix OR using a constant number of rounds, which would be useful for long prime numbers. The drawback of the constant-round protocol is a significantly greater communication cost due to an increase in the quantity of shared random numbers that need to be generated.

The variation in the round count stems from the random generation process. The subprotocol to secretly calculate the least significant bit of a number requires a random number to be generated bit by bit. So, if we have a \(b\)-bit prime, \(p\), we will generate \(b\) random bits and concatenate them to create random number \(r\) between 0 and \(2^b\). The parties secretly compute \(r < p\), reveal the result of the comparison. If it is revealed that \(r \not< p\), then the random generation process must be restarted. Due to the interdependence of gates in the circuit, failing random generation in one gate will affect the latency of the entire circuit calculation. In addition, in terms of communication, random bit generation is one of the most costly algorithm because each instance of the protocol requires a shared random generation, a share multiplication, and a reconstruction, all of which require broadcast communication. Therefore, when choosing a field size for computation, it is beneficial to choose the largest prime less than a power of 2.

Another point of note is that the sorting network presented in \[\text{[LP90]}\] does not benefit our protocol due to the small number of parties. In fact, the the paper only guarantees that the network only differs from a bitonic sorting network when there are at least \(2^l\) inputs with,

\[
l > \lceil 0.822(l + 2) \rceil + 2 \implies l = 25
\]

\(^1\)Prefix-OR is a binary operation on a number. the \(n\)th bit of the prefix-OR of a number is calculated by taking the OR of the first \(n\) bits.
This is much higher than is feasible to test, additionally and is infeasible in practice. Therefore, our sorting network was a bitonic sorter. The round count is $O \log^2 n$, and the number of gates is $n \log^2 n$ where $n$ is the number of parties.

Finally, using quorums, while increasing the number of rounds moderately, decreases the message count and data transferred significantly. The increase in rounds stems from the need to do resharing. However, on the whole, the decrease in data transfer and message count is more significant.

6. Acknowledgements

A special thanks to Professor Feigenbaum who agreed to supervise this project even with all of her work as department chair and to Mahdi Zamani who helped me understand a lot of the multi-party computation protocols.

References


A. List of Protocols Implemented

An extended description of many of the protocols can be found in [Tur14]

- **BitCompositionProtocol** Given a list of shared bits $b_0, \ldots, b_n$, compute $\sum_{i=0}^{n} 2^i b_i$. This is done locally and requires no communication.
• **BitwiseLessThanProtocol** Given two numbers, each expressed as a list of shared bits $a_0, \ldots, a_n$ and $b_0, \ldots, b_n$, secretly compute $a < b$. Requires bitwise XOR (using **BitwiseOperationProtocol** and **SharedBitXor**), prefix OR (using **PrefixOperationProtocol** and **SharedBitOr**), and **ShareAdditionProtocol**.

• **BitwiseOperationProtocol** Given an arbitrary subprotocol represented by the operator $\circ$, and two bitwise shared numbers $a_0, \ldots, a_n$ and $b_0, \ldots, b_n$, computes the bitwise shared number $a_0 \circ b_0, \ldots, a_n \circ b_n$.

• **CompareAndSwapProtocol** Given a tuple of shares, $(a, b)$, return the tuple $(a, b)$ if $a < b$ and $(b, a)$ otherwise. Requires **ShareLessThanProtocol**, **ShareAdditionProtocol**, and **ShareMultiplicationProtocol**.


• **LessThanHalfPrime** Given a shared number $a \in \mathbb{F}_p$, compute a share of $a < p/2$. This is done by computing the least significant bit of $2^a$. Depends on **LeastSignificantBitProtocol** and **ShareAdditionProtocol**.

• **MajorityFilteringProtocol** Receives values from multiple parties who are supposedly supposed to be sending the same value, and returns the value that was received from a majority of the time. Since we assume that fewer than one-third of the parties are malicious, all of the honest parties should send the same value.

• **MultiPartyShufflingProtocol** The entry point for multiparty shuffling. Each party contributes a private value. At the end of the protocol, each party holds an identical random permutation of the values from every party.

• **MultiPartySortingProtocol** The entry point for multiparty sorting. Each party contributes a private value, and at the end of the protocol, each party holds a sorted list of the private values.

• **MultiQuorumGateEvaluation** Receives input from the quorums evaluating the previous gates in the circuit, evaluates a single gate, and sends input to the next gates in the circuit.

• **PrefixOperationProtocol** Given an arbitrary subprotocol represented by the operator $\circ$, and a bitwise shared number $a_0, \ldots, a_n$, computes the bitwise shared number $b_i = a_0 \circ \cdots \circ a_i$. This can also be used as a reduction operation if we discard all bits of the result except the last bit.

• **QuorumShareRenewalProtocol** Performs the share renewal protocol described above. Used to perform degree reduction and randomization after a multiplication and also to reshare a value between two quorums. Depends on the **ShareRenewalRound**.

• **RandomBitGenProtocol** Generates a random $b \in \{0, 1\}$. Depends on **RandomGenProtocol**, **ShareMultiplicationProtocol**, and **ReconstructionProtocol**.

• **RandomBitwiseGenProtocol** Given an $n$-bit prime $p$, generates uniformly at random an $r \in \mathbb{F}_p$ in terms $r$’s bits $r_0, \ldots, r_n$. Depends on **RandomBitGenProtocol**, **ReconstructionProtocol**, and **BitwiseLessThanProtocol**.

• **RandomGenProtocol** The simplest random generation protocol. Given a prime $p$, it generates a share of $r \in \mathbb{F}_p$ chosen uniformly at random.
• **SecureMultiQuorumCircuitEvaluation** Evaluates all of the gates for a party in a quorum. Depends on **MultiQuorumGateEvaluation**

• **ShareAdditionProtocol** Adds two shares. This is done locally without any communication.

• **SharingProtocol** Used to encapsulate secret sharing a private value \( v \) to a quorum.

• **ShareLessThanProtocol** Given 2 shared numbers \( a \) and \( b \), computes a share of \( a < b \). Depends on **ShareMultiplicationProtocol**, **ShareAdditionProtocol**, and **LessThanHalfPrime**

• **SharedBit\{ And, Or, Xor \}** These three protocols compute the conjunction, disjunction, and exclusive disjunction of shared bit values, respectively. They depend on **ShareMultiplicationProtocol** and **ShareAdditionProtocol**

• **ShareMultiplicationProtocol** Given 2 shared numbers \( a \) and \( b \), compute a share of \( ab \). While the multiplication can be done locally, we must do degree reduction and randomization on the resulting shares using the **QuorumShareRenewalProtocol**

• **ShareRenewalRound** Used to perform a single round of share renewal, which can produce at most 3 times as many output shares as input shares. It may be necessary to perform multiple rounds of share renewal within a quorum in order to if the quorum to which the value is being reshared is much larger than the original quorum.

• **SynchronizationProtocol** Given a list of protocols, execute them and wait for all other parties to execute their corresponding protocols. This is used to ensure that circuit evaluation occurs synchronously as different quorums can complete their gates at different times.