# Project Report for "Secure Multiparty Computation at scale" Fast MPC Network Distance by Symbolic Optimization of Unrolled Loops

Kinan Dak Albab[a], Rawane Issa[b]

[a]*Boston University*
[b]*Boston University*

**Abstract**

Secure Multiparty Computation (MPC) is a growing field with many important applications. The main technical obstacle hindering its use in a wide range of domains is its performance (when compared to equivalent "open" computation). In this paper, we attempt to apply MPC to securely compute network distances over a large network formed by multiple private sub-networks. We apply two techniques for speeding up the computation: (1) Performing the computation in stages, where the first stage is done privately by each party, and the second stage is an MPC computation over a much smaller network. (2) Optimizing the MPC stage by unrolling the execution of the algorithm into a symbolic expression, and optimizing the expression to reduce the number of min operations.

*Keywords:* Secure multiparty computing, Network distance, Symbolic expressions, Unrolling loops.

## 1. Introduction

Secure Multiparty Computation (MPC) provides parties with the ability to compute aggregates and functions over a set of inputs. The inputs of each party are considered secret and are not revealed during the course of the computation. The only information released by the computation is the final output (could be different to each party). Clearly, MPC can have great impact over many domains where the input data is sensitive but aggregating data from multiple source can provide beneficial information to participants. However, MPC performs much slower than evaluating the functions over the inputs directly in the open (by a trusted party). This added cost comes from the added cost of communication and the additional complexity of performing operations over shares of data instead of the complete input.

In this paper we develop a solution for computing distances in a network. The network consists of many sub-network each belonging to a party, as well as public connections between some number of "gateway" nodes in these sub-networks. Each participating party will know only the distance between each of its own nodes and the closest "threat" node. Threat could be used to

express any useful metric like security leaks, publicly accessible nodes, crashed nodes, etc..

We provide two techniques to improve the efficiency of the computation, we believe these techniques can be applied to a variety of problems.

1. *Reducing the size of the MPC:* We split the computation into stages. The function to be computed (Network distance) is applied locally by each party. Selected pieces of the local outputs (Gateway nodes) are then secretly shared between the parties and consist the input of the MPC stage. The gateway nodes need to be augmented with additional distance information to properly represent their local sub-networks in the secure computation. The results of the MPC stage are put back into the local sub-network, and the parties apply another round of local computation to compute the final distance for each node.
2. *Optimizing the MPC by unrolling the loops:* The network distance problem is iterative in nature, each iteration propagates the "threat" metric from each node to its neighbors. Nodes take the minimum threat from each neighbor. This require us to compute a large number of min operations in every iteration, the result of which are feed into the next iteration. Given that min is an expensive operation to perform in MPC. We unroll the loops into symbolic expression, and optimize that expression by exploiting the properties of min and addition.

The rest of the paper is as follows: In section 1 we provide the detailed problem definition along with the security assumptions and guarantees. In section 2 we discuss some of the difficulties that make this problem hard to solve efficiently in MPC, and briefly discuss our proposed solution to each. Section 3 describes the stages of the computation. Section 4 details our technique to optimize the MPC stage by optimizing its equivalent symbolic expression. We provide details about our implementation and some benchmarks in Section 5. We discuss future work and other possible applications of our technique in section 6. We conclude in section 7.

## 2. Problem Definition

Given an un-directed graph $G$ with a set $V$ of nodes. Nodes can be either safe or vulnerable. We must find the shortest distance between each node and the closest vulnerable node. We assume initially that vulnerable nodes have distance 0, and safe nodes have distance $\infty$. More formally, the function we are trying to compute is:

$$f(n) = \min_{v \in VN}(dist(n, v))$$

Where $VN$ is the set of vulnerable nodes, and $dist(n, v)$ is the shortest distance between nodes $n$ and $v$.

Algorithm 1 gives a simple algorithm for computing network distance. $nb(n)$ is the set of neighbors of $n$. $d_i[n]$ is the distance computed at iteration $i$ for node $n$. Notice that initially $d_0[n] = 0$ or $\infty$ if n was vulnerable or safe respectively. The number of iterations can be set to the number of nodes. However, it is sufficient to set it to the diameter of the graph. The runtime of the algorithm is $O(n \times d \times k)$, where $n, d, k$ are the number of nodes, diameter, and degree respectively.

**Algorithm 1** Network Distance

1: $i \leftarrow 0$
2: **while** $i <$ `iterations` **do**
3: **for** $n \in V$ **do**
4: $m \leftarrow \min_{v \in nb(n)} (d_i[v] + 1)$
5: $d_{i+1}[n] \leftarrow min(d_i[n], m)$
6: $i \leftarrow i + 1$

Notice that if $n$ is fixed then $d$ quickly decreases if $k$ is increased (by Moore's upper bound [1]).

We consider input networks that are composed of the following parts:

1. Several private networks, each belonging to a party. The number of nodes in these networks and their connections is unknown. Each private network interacts with the rest through a number of publicly-known gateways (a small subset of the private nodes).
2. A public network that connects gateways of different parties. The number of gateway nodes per party and their connections with gateway nodes of other parties is assumed to be public information. Notice that an edge between two gateways of the same party is considered private and not part of the public network. The total number of gateways is assumed to be much smaller than the total number of nodes in the entire network.

### *2.1. Security Assumptions*

The security model is assumed to be semi-honest[2]. We briefly discuss active security in the end of the paper. Each party is supposed to learn only the distances for its own nodes, thus our MPC provides different outputs to each party. The parties are interested in hiding the following information:

1. The topology of a party's private network. Including the total number of nodes and their connections.
2. The shortest path along which a vulnerable node propagated to any node in a party's output.
3. The origin of a vulnerability, i.e. which node or party network is a vulnerability coming from.

Each party will know only the distances from each of its nodes to the closest threat. If the party is connected to multiple parties, or if the public network connecting the parties is sufficiently complex, then there will be many possible paths along which the vulnerability could propagate, thus hiding the original source.

## 3. Difficulties

Attempting to find the distances using MPC over the entire network faces many difficulties: Given our security assumptions, the computation can not publicly refer to the internal topology of

the parties' networks. Thus, the number of nodes in each party network and their connections must be secretly shared as inputs to the computation as well as the initial distance (0 or $\infty$). This will add a huge communication cost, in addition to complicating the computation since it will operate on "secret" networks. This will be unpractical to implement for a variety of reasons, especially given networks of realistic sizes.

Even if we relax our security assumptions to allow leaking the number of nodes or similar information, running MPC on the whole network will still be unrealistic due to the size. The diameter of the whole network is unknown (unless if the parties reveal the internal connections between nodes), therefore we would have to use some lose upper bound (like the total number of nodes) as our number of iterations. This will cause the algorithm to use many more iterations than it has to, blowing it up to something close to $O(n^3)$ for dense networks.

We propose doing the computation in multiple stages. Each party will compute the distances of their own private networks (as if in isolation), in particular the distances of the gateway nodes, which will form the input to the MPC. This will greatly reduce the input of the MPC from the entire network to a smaller subset of nodes. Since the network connecting the gateways is public, we avoid the complications of secret sharing the network into the MPC, thus we can use Algorithm 1 described above for all stages of computation.

Our algorithm for computing the distances involves computing a min over the previous distance of the nodes and the distances of its neighbors (with 1 added since they are one edge away). This min is carried to the next iteration where it will serve as one of the arguments to another min. In particular, we will get one min per node per iteration, each of these mins has as many arguments as neighbors for a node. Performing min is expensive in MPC since it require performing comparisons without code-branching, since inspecting which instructions are executed will reveal the result of the comparison.

We propose to use symbolic values instead of secret shares initially. Executing the algorithm will cause the additions and minimums to be carried over "symbolically" throughout the iteration. Eventually yielding a single expression per node. These expressions are equivalent in size to the actual execution of the algorithm. We will then optimize these expressions to reduce the number of of min operators that are required. Eventually reducing the expression to a single min with many arguments.

## 4. Computation Stages

### *4.1. Local Stage - Input*

In the first stage of the computation, each party will perform the algorithm locally on its private network. We are only interested in computing the distances of the gateways. It may be useful to use a modified version of the algorithm that computes the distances for only the gateways without

having to worry about distant nodes that do not contribute.

A party's gateways may be connected internally inside its private network. These connection must not be ignored by the MPC stage since they can affect the outcome of the computation. For example, imagine a case where two parties networks are not connected directly. Instead, they are connected to two different gateways of a third party, these gateways are then connected internally by some path in that third party's local network. Failure to provide the MPC stage with information about the connection between these two gateways will cause vulnerabilities in the first two parties to be "invisible" to the other party, and blocking them from propagating in their direction.

Therefore, each party compute the shortest distance between every pair of gateway nodes. If two gateways are unconnected, the distance is set to be $\infty$. These distances will server as "weights" to imaginary edges between gateways in the MPC stage. The actual value of the weights remain unknown as they are secret-shared. This will not reveal the existence (or in-existence) of a path between two gateways, nor will it reveal its length. Thus, no information about the internal topology of any private network is leaked.

### *4.2. MPC Stage*

When all parties are done with their local computations, they will enter into a multiparty computation between them. Each party will then secret share (using Shamir secret sharing [3]) the distances of each gateway and the "weights" between every pair of gateways.

Each party will run Algorithm 1 symbolically, accumulating min and addition operators into symbolic expressions. The expressions are optimized as will be discussed in the next section. When the expressions are ready, each party will evaluate them using the shares of the inputs it possess. The result of each expression is then shared with the party that owns the the expression's respective node. Thus, only the party to which a gateway node belongs to will learn the distance of that node to a vulnerability.

### *4.3. Local Stage - Output*

Finally each party will proceed onto a new local stage. The output distances of each gateway learned in the MPC stage is plugged into the result of the first local stage. Each party will compute algorithm 1 locally. Thus, propagating the distances to local vulnerabilities as well as vulnerabilities learned in the MPC stage (external vulnerabilities).

## 5. Expression Trees

Each of the symbolic expressions attained by running the algorithm symbolically (as described in the previous section) is represented as a parse tree. Given the algorithm we are using, the parse tree is defined as follows:

$$exp ::= \quad < int > \quad | \quad X_n \quad | \quad W_n^m$$
$$| \quad min(< exp >, ..., < exp >)$$

$$| \quad < exp > + ... + < exp >$$

Where $X_n$, $W_n^m$ are symbolic values / variables representing the secretly-shared distance of gateway $n$, and "weight" between gateways $n$ and $m$ (all computed in the local stage by parties).

Furthermore, the tree would be made of alternating levels of min and addition operators. Where we have a min on top (representing a single iteration for a single node in the algorithm), and its children are addition operators (each neighbor's expression + 1 or each gateway's expression from the same party + a weight). And the leafs are $X$ variables representing the input.

Our goal is to optimize the expression by applying reductions to the parse trees, the reductions will utilize the algebraic properties of + and *min*. We would like to reduce the number of *min* operators while pushing them all the way up to the root of the tree. For this we will use the given reductions:

1. ***Addition-Min Reduction:*** Given a parse tree of the form $min(< exp0 >, ..., < expn >) + (< expA > + ... < expN >)$, we will reduce this expression by taking the addition inside the min operator. The resulting parse tree will be $min(< exp0 > + < exp >, ..., < expn > + < exp >)$ where $< exp >=< expA > + ... < expN >$. Notice that the resulting min will have the same number of arguments as the min we started with. Also, $< expA >$ to $< expN >$ are guaranteed to not contain a min operator since the algorithm only adds 1 or "weights".
2. ***Min-Min Reduction:*** Given a parse tree of the form $min(min(< exp0 >, ..., < expn >), ..., min(< expm >, ..., < expk >))$, we will reduce this expression to a single min operator by adopting all the children of the nested mins. The resulting expression will be $min(< exp0 >, ..., < expn >, ..., < expm >, ..., < expk >)$.
3. ***Early-Min Reduction:*** Given a parse tree of the form $(min(< exp0 >, ..., < expn >)$, assuming that the arguments do not contain any min operators, we will attempt to reduce this min to an equivalent min with less than or equal number arguments. We will look for arguments that cannot be be the desired min and remove it. In other words, we will remove argument $< expi >$ if we can find another argument $< expj >$ such that $< expi > ¡ < expj >$. Duplicate expressions are removed except for one.

We apply the first two reductions in alternation starting from the bottom of the parse tree up, since the parse tree is itself alternating between addition and min operators. This will simplify the parse tree to an equivalent single min operator without removing any argument to a min operator (that is not a min itself). Note that the algorithm itself can be modified to produce such an expression immediately by applying the reductions in every iteration.

We apply the third reduction to the resulting expression made of a single min operator. The third reduction will remove an argument $< exp >= X_n + W_0 + ... + W_m + i$ in the following cases:

- Another argument of the form $X_n + W_0 + ... + W_m + i'$ exists and $i' < i$.
- Another argument with $X_n$ exists which has $i' \leq i$ that uses a strict subset of weights.

- Another argument with $X_n$ exists which has $i' - i = k > 0$ but utilizes a strict subset of weights with at least $k$ less weights (since all weights are $\geq 1$).

For cases where each node has exactly one gateway (no "weights" exist). We are guaranteed that we will get an expression with a single min and as many arguments as nodes. Since all the arguments are on the form $X_n + i$. The number of arguments is larger than that for networks with weights and it depends on the structure of the network, since expressions with many weights are easier to remove if they include high integer constants. We discuss possible ways to achieve more optimization when many weights are involved in the future work section.

## 6. Tool-set and Implementation

We implemented a python library "ExpressionMPC", which allows programs to quickly code segments that execute symbolically during program execution. We provide python classes (called simplifiers) that implement the reductions described in the paper. As well as a VIFF[4] based evaluator. The evaluator uses VIFF to secret share all the input distances and weights between parties, then interpret the expressions using the generic VIFF operators, the results are opened and shared with the rightful parties only. The users of the library can write their own simplifiers or evaluators and use them in combination with the provided ones.

We used operator overloading to accumulate operators into expressions. The simplifiers and evaluators are visitor-pattern like. Code written using this library will not show direct references to MPC constructs or parse-tree manipulations. However, smart use of simplifiers inside the code (in between iterations for example) can help reduce the time needed to process and optimize the symbolic expressions.

We have attempted to run our solution on a total network consisting of 24 nodes, 38 edges, and 3 parties. The public network consists of 6 gateways and 7 edges. Our solution required 0.12 to optimize the expression, and 16 seconds to evaluate it. An equivalent program with the use of symbolic optimization took 24 seconds on the same input. We plan on running more extensive benchmarks (larger networks, more parties, on the cloud) soon.

### *6.1. Active Security*

VIFF provides the ability to change the security configuration (through the VIFF Runtime). VIFF will automatically ensure that all parties are following the provided protocol, notice that even if a malicious party attempted to deviate from the protcol in the MPC stage, it can only do so by either attempting to manipulate its shares, or evaluating a different expression (i.e executing a different set of instructions). VIFF already has guarantees for these kinds of behavior.

However, although the local stages are put in place only for efficiency benefits, doing the computation in stages differ (slightly) from doing the whole computation in MPC with regards to security guarantees.

Notice that our protocol requires each party to provide as inputs to the MPC both the locally-computed gateway distances as well as gateway-pairs weights. A malicious party could provide weights that do not match the gateway distances and their internal network. The part can pick the distances and weights separately for whatever malicious purpose. This can be divided into two cases:

1. The provided weights and distances do not match the party's actual internal network, but they do match some other network(s): this case is equivalent to the part "lying" and providing one of the matching networks as input to the equivalent computation done entirely in MPC.
2. The provided weights and distances are inherently inconsistent and cannot match any network: for example, the weights do not satisfy the triangular inequality. This case cannot arise if the computation was done entirely in MPC, since the party cannot pick the weights. Additional checks must be added to the MPC stage to check consistency of the input. It is unclear what the malicious party may learn in doing this but could not learn by picking consistent inputs.

## 7. Future Work

We hope to apply the techniques discussed in the paper to more examples. We believe that organizing the computation in stages to reduce input size, as well as symbolically optimizing the computation are general enough to apply to a variety of problems. As long as the algorithms has a publicly available termination condition. In particular, we want to investigate minimum spanning trees and max-flow (details to be worked out).

We want to extend the library to include more expressions and operators, including support for conditionals and loops. As well as provide additional simplifiers for different use-cases. Many operators have algebraic properties that we can use to come up with useful reductions.

Finally, we hope to be able to transform our code into an all-purpose general library, which will allow the user to apply built-in protocol and simplifications without worrying about the underlying cryptographic, MPC, and symbolic optimization work. We would like to release the library's source code as well as making it easy to install using the python package index (pip). We hope this can make MPC more accessible to developers, and more efficient for wider application and use.

## 8. Summary

We implemented an optimized solution for computing shortest distances between nodes and vulnerable nodes in a network with many parties. We described two general techniques for doing this optimization: (1) performing the computation in stages some of which are local, such that each stage reduce the size of the input. For the network distance problem, we computed locally the network distance and weights of gateways of private networks. Then we computed the actual distances of these gateways together in MPC. In the third stage, we propagated the computed

distances locally into the private networks to compute the distances of every node. (2) unrolling iterative algorithms into symbolic expressions that are optimized and then evaluated in MPC. We utilized problem-specific reductions that exploit the distributivity of min and addition as well as attempting to remove arguments by syntactic comparisons to reduce the number of min operators that must be evaluated. We believe these techniques can be applied to many problems in addition to network distance, causing the MPC solutions for more problems to become more efficient, simple, and therefore practical.

## References


[1] A. J. Hoffman, R. R. Singleton, On moore graphs with diameters 2 and 3, IBM J. Res. Dev. 4 (1960) 497–504.

[2] O. Goldreich, Foundations of Cryptography: Volume 2, Basic Applications, Cambridge University Press, New York, NY, USA, 2004.

[3] A. Shamir, How to share a secret, Communications of the ACM 22 (1979) 612–613.

[4] I. Damgård, M. Geisler, M. Krøigaard, J. B. Nielsen, Asynchronous Multiparty Computation: Theory and Implementation, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 160–179.