Implementing Topology Hiding Multiparty Computation

Weston Braun   John Grosen   Lilika Markatou

May 17, 2018

1 Introduction

We were interested in exploring the beautiful theory of topology hiding computation. For our project we implemented the topology hiding computation algorithm that was developed by Adi Akavia, Rio LaVigne and Tal Moran [1]. We consider honest-but-curious adversaries.

2 Related Work

There has been plenty of work in the topology hiding computation area. The first result was a feasibility result by Moran, Orlov, and Richelson [7]. This result only works in graphs whose diameter is logarithmic in the number of participants. Another interesting result in this area is by Akavia and Moran [2]. This result works in graphs that are cycles, trees or that have logarithmic circumference in the number of participants. Another notable result is the paper by Hirt, Maurer, Tschudi and Zikas [6] which improves upon the efficiency of [7].

In our project, we chose to implement the algorithm from [1] as it's the only one that works in arbitrary graphs.

3 Algorithm

The specific problem we are working on is the following:

Problem 1 *Given a network with n nodes, compute the OR of all their values, while hiding the nodes' values and the topology of the network.*

3.1 High-level Overview

We give a high-level overview of the algorithm we plan to implement, as described in [1].

The first thing we need to address is how to hide the nodes' values. We will use homomorphic encryption, and more specifically El-Gamal. El-Gamal allows us to hide the nodes' values and it's relatively easy to augment with some convenient functions. For example, we can construct a Homomorphic OR based on El Gamal.

Now that all the values are encrypted, we need to find a way to OR them. To this end, the nodes will send their messages to perform *random walks* across the graph. A random walk is a path that consists of random steps.

More specifically, each node v will initially send its encrypted message to a random neighbor. Once a message is received, v will homomorphically OR it with each own message, and then send it to a random neighbor. The messages will perform random walks for a large enough number of rounds, until we know with overwhelming probability that they have visited each node in the network. The random walks are composed of $8\kappa n^3$ steps, where κ is defined by the user. The larger the κ, the higher the probability that the random walks cover the whole network.

There is a problem with this approach. Given two ciphertexts, encrypted with different public keys, we can't homomorphically OR them. Akavia et al. have a really cool solution to this problem. They define two functions *AddLayer*, and *DelLayer*.

AddLayer can take an encryption of a message m with public key pk_1, and a secret key sk_2, which pairs with public key pk_2 and generate approximately an encryption of m with public key $pk_1 \cdot pk_2$. *DelLayer* reverses the *AddLayer* operation. This way, since all public keys are known, each node can generate an encryption of its own message and an encryption of the received message, both encryptions with the same public key and then homomorphically OR them.

Now, once the messages have completed their random walks, they can simply backtrack their progress, with the nodes removing any encryption layers they added, until the messages return to their origins. Once at the origin, the node can simply decrypt the received message and get the OR of all the nodes' values.

3.2 Primitives

The encryption behind this algorithm is standard El-Gamal, with some additional features, like adding an encryption layer, deleting an encryption layer, a randomization function, and Homomorphic OR.

3.2.1 El-Gamal

Fix the group G and its generator g.

$$\begin{aligned}
&KeyGen() \rightarrow (pk, sk) = (g^x, x) \\
&Enc(pk = g^x, m) \rightarrow (g^y, m \cdot g^{xy}) \\
&Dec(sk = x, c = (c_1, c_2)) \rightarrow c_2 / c_1^x
\end{aligned}$$

3.2.2 Adding and Deleting Layers

We define adding and deleting layers as follows:

$$\begin{aligned}
&AddLayer(c = (c_1, c_2), sk) \rightarrow (c_1, c_2 \cdot c_1^{sk}) \\
&DelLayer((c = (c_1, c_2), sk) \rightarrow (c_1, c_2 / c_1^{sk})
\end{aligned}$$

3.2.3 Randomization Function

The following randomization function is used, for some random coins r:

$$Rand((c = (c_1, c_2), pk, r) \rightarrow (c_1 \cdot g^r, c_2 \cdot pk^r)$$

3.2.4 Homomorphic OR

In order to implement a homomorphic OR, we need two more functions. One to let us homomorphically multiply, and one to let us homomorphically exponentiate.

We can homomorphically multiply two ciphertexts like this:

$$hMult(c = (c_1, c_2), c' = (c'_1, c'_2)) \rightarrow (c_1 \cdot c'_1, c_2 \cdot c'_2)$$

We can use $hMult$ to homomorphically raise any ciphertext to a power by repeatedly squaring. Thus, we can achieve $hPower$ homomorphic exponentiation.

Now, it remains to show how to homomorphically OR. Note that when ORing two numbers, we have to ensure that no information on the numbers leaks given the homomorphic OR. In order to achieve this, we denote that an encryption of 0 is an encryption of the identity and an encryption of 1 is an encryption of some other element.

We assume some random coins r, r'.

Algorithm 1 $HomOR(c, c', pk, r, r')$

```
1: $z = hPower(c, r, pk)$
2: $z' = hPower(c', r', pk)$
3: Return $Rand(hMult(z, z'), pk)$
```

3.3 Algorithm

The algorithm has three phases.

Phase 1: Preprocessing Phase In this phase, we do any preprocessing required.

Let T denote the number of steps each random walk will be. $T = \kappa \cdot 8n^3$, for some κ.

In addition, the nodes generate all the key pairs needed, and all the random permutations needed. The nodes will use the random permutations to decide which neighbor they should send every message they receive to.

The nodes also encrypt their values, with their own public key.

Phase 2: Aggregate Phase This phase lasts T rounds. On each round, each node will send its messages, (ciphertext, public key) pairs, to their neighbors using a random permutation to decide which message goes to which neighbor.

Once a node receives a ciphertext c, and public key pk pair, it applies *AddLayer* on the message and then homomorphically ORs the generated encryption with an encryption of its own value, generating a new ciphertext c' and new public key pk'. Then, using the previously calculated permutations, it chooses a neighbor at random to send (c', pk') to.

Phase 3: Decrypt Phase This phase also lasts T rounds. On each round, each node passes their messages back to the node that that message came from during phase 2.

Once a node receives a message, the node uses *DelLayer* to remove its encryption layer, and then figures out which node it should pass the message back to, depending on what happened during phase 2.

On the last round of this phase, the messages have returned back to the node who first requested the OR. That node can now simply decrypt the message using its secret key and find out the OR of all the nodes' values.

3.4 Complexity of the Algorithm

This algorithm has a round complexity of $O(\kappa \cdot n^3)$, for some user defined κ. On every round of the algorithm, each node sends a message to all its neighbors. Thus, on every round a maximum of n^2 messages is sent over the network. This algorithm has a communication complexity of $O(\kappa \cdot n^5)$.

4 Implementation Details

4.1 Library

Our source code is available under the MIT license at

n	Running Time (s)
2	0.64
3	7.6
4	30
5	105
6	278
7	614

Figure 1: Runtimes with $\kappa = 4$ under the simulator

`https://github.com/jmgrosen/topohiding`.

We chose Python for our implementation language for its simplicity and ease of use. For a real-world implementation, a lower-level and thus faster language would be more appropriate, but for this project, we valued clarity over performance.

We began by implementing the low-level cryptographic primitives: key generation, group operation on pubic keys, encryption, decryption, re-randomization, adding layers, deleting layers, and homomorphic OR. We ran some tests with random elements to ensure these primitives were working. For debugging purposes, we also implemented mock cryptographic primitives that simply log operations (for example, encrypting "0" then adding a layer might look like "AddLayer(Enc(0, k_1), k_2)").

On top of the primitives, we implemented the actual topology-hiding mixing. It is a straightforward translation of the above algorithm description into Python. The library is implemented from the point of view of one node, so that it can be used in actual settings, not just simulation.

4.2 Simulation

The first user of the library is a "simulator" which simply runs the algorithm with multiple nodes in the same process. It is given a description of a graph (implemented using the `networkx` Python library), which it uses to emulate network links. It runs each round sequentially, passing the results from each node to its neighbors at each new round.

To roughly evaluate the real-world performance of the protocol, we timed how long it takes to simulate a complete graph of n nodes for $n = 2$ to 7. The results of the experiment (performed on a laptop) are shown in 1. Clearly, the scaling makes this impractical for nontrivial networks. Improving the performance of the code would bring down the constant factor, but the $O(n^5)$ inevitably keeps the runtime nasty.

4.3 Demo Application

In addition to the simulation, we also developed a proof of concept command line application that can run the topology hiding protocol between multiple networked computers. Each user provides a list of nodes they wish to connect to in IP:Port form, the value to be or'ed, an upper bound on the number of nodes, and a value for κ, which determines the error rate. These parameters all have to be determined via some side channel in the current version of the application. The parameters g and q are public and hard-coded into the application.

The communication between nodes is socket based, and due to the requirements for topology hiding there can be no master node to coordinate communication. To simplify the communication protocol each node opens a server socket to receive connections from its neighbors and opens a client socket with each neighbor. This makes the communication protocol between nodes symmetric. However, due to the this setup there is a need to map received connections on the server socket with opened client sockets.

Once the application is launched there is a discovery phase to allow this matching. Each node generates a unique ID on start and sends this ID as it opens client sockets with the user entered list of neighbors. Upon receiving a connection on its server socket each node reads the received ID and replies with its own ID. Matching these IDs allows each node to create a persistent mapping between connections to its server socket and opened client sockets.

Once a persistent mapping between socket connections is achieved our Python library is used to run the algorithm. Our library already provides the messages produced by each round in a serialized format so there is little overhead in transmitting messages between nodes. The first round for each node requires no message inputs from other nodes. After the first round, each node sends the output messages from a round to each neighbor via its client sockets and receives messages from other nodes via its server socket. The received messages are required for the next round. Synchronization between nodes is ensured by the fact that no node can compute a round without messages from all of its neighbors.

Once the algorithm is completed the result is printed to the terminal for viewing by the user.

5 Potential Applications

The proposed topology hiding computation system from [1] that we implemented only supports the operation of single bit "or". By coordinating multiple rounds we can extend this to an "or" of an arbitrary number of bits. Additionally, by having the users invert the input we can achieve the "and" operation.

Applications that might utilize the primitive of single bit "or" as implemented in our demo application include unanimous approval / veto votingor determining possible exposure to some disease through a social graph. Secure anonymous veto methods have been previously proposed but the membership of the group is known [5]. A topology hiding system such as the one we implemented would allow group members to additional members while keeping the presence and identity of the new members secret and protecting the confidentiality of the newly added member's vote. An additional application that we find promising is confidentially determining possible exposure to communicable diseases. There exist many communicable diseases which carry a social stigma but for which determining exposure to is important from a public health prospective, such as tuberculosis. By representing possible interactions as edges on a graph an extended social circle can determine if its members are at risk without anyone exposing sensitive health information or their social interactions.

Our implementation of the topology hiding computation system does not currently support "or" over multiple bits. However, due to the randomized nature of the cryptography it is possible to run the protocol multiple times in parallel over the same random walk without impacting the privacy guarantees or the probability of the success. This increases the communication complexity by a factor of $O(B)$, where B is the number of bits. The round complexity remains the same.

A potential application we have realized that maps well to this primitive is the *bloom filter*. A bloom filter is a computationally efficient method of checking if an item possibly belongs to a set using a hash function [4]. In a bloom filter items are hashed and then bitwise OR'ed into a persistent bit array. Checking if an item is in the bloom filter consists of hashing it and comparing all the bits of the hash with the value of the bloom filter. Items possibly in the set will have all 1 bits in their hash match with 1 bits in the bit array while an item not in the set will have at least one 1 bit in their hash not match with a 1 bit in the bit array. Bloom filters map well to applications like content filtering where there is a need to check if a large volume of content is part of some restricted list and only rely on "or" to add items.

With the rise of federated social networks such as Mastodon[3] it may be useful to have shared content filters for prohibited content. Topology hiding multiparty computation could allow for multiple instances to add items to a shared bloom filter representing prohibited content without revealing where the content originated and hiding which instances peer with other instances.

It is also possible to use the topology hiding computation algorithm as a generic broadcast method for other multiparty computation methods, as outlined by [1]. This allows the property of topology hiding to be extended to multiparty computation methods beyond or / and. However, given the relatively high computational cost of the topology hiding computation method this may not be the most feasible for many applications.

6 Conclusion & Future Work

In this project we implemented the topology hiding computation algorithm described by [1] to produce a multiparty computation topology hiding system that provides the primitive of single bit or. We created a python library to allow integration of this functionality in to projects and a demonstration command line application that can allow parties on separate computers with internet connectivity to run the algorithm for applications like acceptance / veto voting.

The biggest challenge in utilizing using topology hiding in applications is the unfavorable round complexity, which is $O(n^3)$. Our test cases took several seconds to run on our personal computers for relatively small numbers of nodes.

For future work we would work on a GUI for our demo application to improve usability as well as ways of reducing the algorithm run time. Porting our code base from Python to a higher performance language could provide a constant run time reduction. Additionally, finding methods to reveal select properties of the graph could allow for a new bound on the time complexity at the trade off for some of the topology hiding guarantees and possibly represents a new research area in the field of topology hiding computation.

References

[1] Adi Akavia, Rio LaVigne, and Tal Moran. Topology-hiding computation on all graphs. In *Annual International Cryptology Conference*, pages 447–467. Springer, 2017.

[2] Adi Akavia and Tal Moran. Topology-hiding computation beyond logarithmic diameter. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 609–637. Springer, 2017.

[3] Nolle Anthony. A brief introduction to mastodon, 2018. URL: https://gist.github.com/joyeusenoelle/74f6e6c0f349651349a0df9ae4582969.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. URL: http://doi.acm.org/10.1145/362686.362692, http://dx.doi.org/10.1145/362686.362692 doi:10.1145/362686.362692.

[5] Feng Hao and Piotr Zieliński. A 2-round anonymous veto protocol. In Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe, editors, *Security Protocols*, pages 202–211, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[6] Martin Hirt, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Network-hiding communication and applications to multi-party protocols. In *Annual Cryptology Conference*, pages 335–365. Springer, 2016.

[7] Tal Moran, Ilan Orlov, and Silas Richelson. Topology-hiding computation. In *Theory of Cryptography Conference*, pages 159–181. Springer, 2015.