Implementation of Trusted Initializer as a Web Service

Ibrahim Diabate
University of Washington, Tacoma
Tacoma, Washington
diabai@uw.edu

Ming Hoi Lam
University of Washington, Tacoma
Tacoma, Washington
mhl325@uw.edu

Matthew Subido
University of Washington, Tacoma
Tacoma, Washington
subidomd@uw.edu

ABSTRACT

There exists a problem with Secure Multiparty Computations in the field of Cybersecurity: to enact many of the protocols that exist for it, one must have shares of initialized random values spread throughout everyone who wants to compute. However, due to the nature of Secure Multiparty Computations, this Trusted Initializer of the shared random values often does not exist. This can be substituted for two-party computations, at the cost of the computations unconditional security. But what if there were a Trusted Initializer that was truly unowned, and could be trusted to give these shares as it was disconnected from any individual party? Our implementation is just that: a RESTful Web Service implementation of a Two-Party Trusted Initializer for an additive multiplication Secure Multiparty Computation protocol written in Java, called the Trusted Initializer.

KEYWORDS

Trusted Initializer, Secure Multiplarty Computation, finite field, additively-secret, moniker, player

ACM Reference Format:
Ibrahim Diabate, Ming Hoi Lam, and Matthew Subido. 2018. Implementation of Trusted Initializer as a Web Service. In *Proceedings of ACM Tacoma conference (TACOMA'18).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Since the introduction of the Millionaire's Problem by Andrew Yao in the year 1982, the cryptograhy community of Computer Science has scrambled to bring an answer to this question: how can people perform operations on some value that each person has *without* any person being able to see another's value? The answer can be found in the field of cryptography called Secure Multiparty Computation. Secure Multiparty Computation focuses on the creation of protocols capable of solving this problem, and allowing parties to jointly compute some value with their individual values without being compromised to one another. Since its inception, this field has created many viable options for computations with multiple parties, such as Secret Sharing and the Garbled Circuit. However, the issue of computation with *strictly* two parties has been discussed at length.

The problem with Two-Party Secure Multiparty Computation is that in order to achieve a true Two-Party nature, they both must sacrifice a level of cryptographic security. The only way to fix this is if there was a third party, capable of performing some calculations while also being trusted to be indifferent to the computations between the two playing parties. We will call this third party the Trusted Initializer, and the party attempting to compute with their values the players. If there were a Trusted Initializer both players trusted enough to actually help perform computations, then this problem would be irrelevant, as the Trusted Initializer could simply take both players input and output the desired result, in this case being multiplication of inputs. Even if the Trusted Initializer simply initialized the computation, they would also have to be always present and available to either party. This is unrealistic for a true third Party, and so we must look elsewhere for inspiration.

We present the design and implementation of the following: a RESTful Web Service that fulfills the role of a Trusted Initializer, for a Secure Multiparty Computation protocol capable of performing multiplication securely for a two-party computation. This Web Service would eliminate the need of an actual third-party, approximating its duties. It would also be capable of being always present and accessible to any player pairs wishing to perform a computation. Lastly, it would be impartial to any specific pair's computations, simply initializing the share and then never interacting with either party again for any given multiplication.

2 DESIGN AND IMPLEMENTATION

In this paper we will detail more in-depth the Secure Multiparty Computation protocol that our Trusted Initializer will initialize for, as well as the design and implementation of said Trusted Initializer as a Web Service, with the incorporation of a client-side website interface, as well as a database for use by the Trusted Initializer itself.

2.1 Secure Multiparty Computation Protocol

The following is a description of the Secure Multiparty Computation protocol that our Web Service will initialize for. It is based on the concept of *additive shares*, in which an integer N is split into two integer n_1 and n_2 such that $n_1 + n_2 = N$:

We have two players A and B, who hold secrets s and t respectively, and also agree on a finite field $F > st$. Keep in mind that all computations detailed throughout this section will be modulus F. They wish to calculate st without A knowing t or B knowing s. They do this by computing additive shares $s_a + s_b = s$ and $t_a + t_b = t$. A then sends their s_b share to B, while B sends their t_a share to A, so that now A holds all a shares and B holds all b shares. Each player then requests shares from the Trusted Initializer over a field F, which both A and B agree upon. . This Initializer will return additive shares of three numbers u, v and $w = uv$, one half of which are given to A and one half of which are given to B. More information on this share generation and initialization will come later when we

TACOMA'18, March 2018, El Tacoma, Washington USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

detail the Web Service. Now each player holds their given shares, as well as their share of u, v and w.

With their shares, A will compute $d_a = s_a - u_a$ and $e_a = t_a - v_a$, while B will compute $d_b = s_b - u_b$ and $e_b = t_b - v_b$. Each player then sends their newly computed numbers to the other player, where both compute $d = d_a + d_b$ and $e = e_a + e_b$. These integers are significant, as $d = s - u$ and $e = t - v$. With these new shares, A computes:

$$x = (d * e) + (u_a * e) + (v_a * d) + w_a$$

and B computes:

$$y = (u_b * e) + (v_b * d) + w_b$$

. A sends the new integer x to B, and B sends their new integer y to A. By adding these two shares together, you get:

$$(d * e) + (u * e) + (v * d) + w = (d + u) * (e + v) = st$$

2.2 Trusted Initializer

The part our Trusted Initializer will take care of is the creation of the random shares of values within a finite field F for two players. To do this, it will need to generate the shares for the players, as well as remember player pairs to accurately send the proper shares back. Generating the shares is a simpler matter; to generate shares for a Two-Party computation, the Initializer needs to randomly compute two number u and v such that $u, v \in F$ and $u, v \neq 0$, as well as integer $w = uv \pmod F$. It must then compute additive shares of $u = u_a + u_b$, $v = v_a + v_b$, and $w = w_a + w_b$. The first person to request their shares will receive the a shares, and be assigned the partner role of A mentioned in **2.1**, and the Web Service will hold on to the later shares for A's partner, who will be assigned the B role. This will constitute a POST method of our RESTful service.

In order for the Web Service to work, A and B must know information about each other before requesting their shares. To perform the Secure Multiparty Computation, both parties must know each other's IP addresses to directly communicate. But for our Web Service, referring to people using IP addresses could be detrimental to consistency, as IP addresses for personal uses are not static, and IP addresses can change for more mobile players. We abstract away the necessity for direct IP address knowledge between parties by allowing someone to post some information about themselves, a *moniker*, to more easily identify themselves and their partners. Our implementation for this *moniker* is a simple ID and password, where each player registers a unique ID code and a password with the system for clarity of reference. When A makes their initial request, they sent their field, their ID, their password and B's ID. The Trusted Initializer then checks to see if A's ID and password match. If they do, it stores B and A's ID along with B's shares, and returns A's shares. When B makes their request later down the line, they sends the same information, which the Initializer then compares to their database of stored shares; because we used B and A's *moniker* to store the shares, retrieving and returning them with B's input is simple. Thus, the two have their halves of their shares and can compute their secret together.

2.3 Server-Side

2.3.1 Choice of Implementation Language and Framework. Despite what we have learnt in the web service course is all based on C# with ASP.NET framework, we still choose Java as the language for implementing the server side of the web service. This is due to one main reason; the core implementation of the trusted initializer requires the BigInteger Class that is embedded in Java. Even though C# also has the BigInteger class, we have more experience with implementing cryptographic algorithms in Java. As a result, we have come to a conclusion during our project planning meeting that we will use Java for implementing the web service for better performance. For the server-side service framework, we have chosen Jersey RESTful Web Services framework. Before working on implementing the service-side service with Jersey framework, we have tried to implement the service with Spring Web Services framework; however, we were not able to get Spring Web Services work with Google App Engine. Because we are able to get Jersey working with Google App Engine platform, we continued to use Jersey framework for our server-side service.

2.3.2 Choice of Deployment Platform. For the deployment platform, we have chosen Google App Engine as a platform for deploying our web service. There are two reasons why we have chosen Google App Engine as a platform for deploying our web service. The first reason is that Google App Engine has provided excellent web service template, which is good for developers like us who are new to developing web service in Java. Moreover, the Google App Engine website has provided detail documentation on how to setup the web service. Since we do not have a lot of time for developing the web service, Google App Engine is definitely a choice for us to efficiently develop our web services. We can easily build and expand our server-side web service based on the template provided by Google App Engine. For the second reason, since it is mostly related to the client-side service, we will explain about it in Section 2.5.

2.3.3 Server-Side Architecture. The server-side of the web service consist of three main components, which are Account, Share Request, and Database Connection. Since the Database Connection component will be discussed in Section 2.4, we will not mention about it in this section. An overview of the server-side architecture is shown in Fig. 1. All the REST methods in our web service take a XML request in POST operation and return a XML response. The XML response will consist of a response code and response description. The response code in the XML response is different from the HTTP response code; it is used for the server to tell the client-side application whether the requested action is successful, fail, or error.

2.3.4 Account Component. The Account component consists of two methods: account validation and registration. The account validation method is responsible for verifying a user's username and password. When a request is received, it will first check the validity of the request body to ensure that username and password are not empty. After that, it will connect with the database to validate the username and password. If the validation is successful, it will return the corresponding response code and description. Otherwise, it will response a response code and description indicating whether the username or password is incorrect, or that there is a system error.

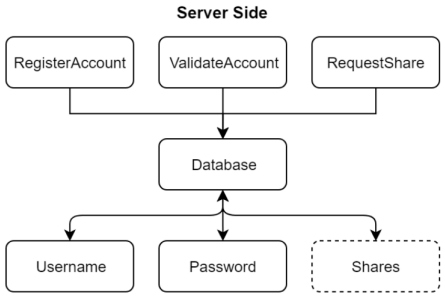

Figure 1: Overview of Client-Side Architecture

On the other hand, the account registration method is responsible for adding a new user to the database. Similar to account validation method, it will also check the validity of the new user's username and password. If they are valid, it will connect to the database to ensure that the username doesn't already exist. In the case that the username has already exists, it will return a response code and description indicating the situation. Otherwise, it will add the new user to the database and return a response code and description indicating the registration process is successful.

2.3.5 Share Request Component. The share request component consists of one method: share request. It is responsible for computing two shares, one for the current user and one for the user's partner. When this method receives a request, it will first verify the validity of the request body and the existence of the user's partner. If the partner doesn't exist, it will terminate the request with a response code and description indicating that the partnerâĂŹs username doesn't exist. Otherwise, it will check there is share already existed between the current user and the user's partner. In the case that there is no share found, it will start computing the share and store the partner's share into the database. If the share request process is successful, the response body will include the current user's share. When the partner request for share through this method, it will return the stored share and delete the share from the database for security reason.

2.4 Database

We used a database to store key components of our system. The components that were relevant to the implementation of this project were the names of the users, their passwords and their shares. Our database was hosted on the AWS (Amazon Web Services) servers in order to enforce the distributed aspect of our system. We believed using AWS as a host would cover any use case related to scalability, reliability and availability, among other needs. We used a T2 micro instance with a maximum capacity of 20 GB. For the purpose of our system, 20 GB proved to be a sufficient amount of space to accomplish our goals. The DBMS (database management system) we chose was mySQL. We chose mySQL because of its flexibility and known high performance. We used two schemas to store information related to our users and their shares.

- The first schema was the Clients schema:

Clients(username, password)

- The second was the Shares schema:

Shares(user, partner, share1, share2, share3)

Based on our set up of the above tables, only registered users could be added to the Shares table and no two users with the same name could register to the system.

We also implemented method directly in our Java source code to cover edge cases in order to ensure smooth transactions and prevent preventable Exception to be thrown.

The following methods were implemented to ease communication beetween the database and our system:

- createConnection(): This method is used mainly to establish initial connection with the database. It takes the credentials required to connect to our mySQL instance via Java's JDBC API, an API used to facilitate client access to a database using the Java language. Without the body of this method, no connection can be started between our system and the database.

- getConnection(): Most of the time this method is called, a connection with the database would have already been made, it is a helper method to save us computation time by rapidly resuming a connection with the database. If connection has been lost, it is re-created via the createConnection() method described in the previous bullet point.

- closeConnection(): For the purpose of following recommended practices of software development, every stream we open is closed upon completion of its task each time its purpose is accomplished. Our intent here was to decouple resources needed for this system as much as possible. As well as it is recommended, it also enforces reusability.

- addUser(username, password): Whenever a user wants to register to our system, we make sure that it is not an already existing user to prevent a SQLException to be thrown. Due to constraints applied to our Clients and Shares tables, where usernames of the Client tables are PRIMARY KEYS and names of the Shares tables are references to the Clients table's user names, adding an already existing error will cause a SQL duplicate error to be thrown. Although this is the expected behavior whenever such queries are processed on data stored with similiar constraints, it did not align with our views for this system. We ensured that preventable SQL Exception (exceptions) of any sort would not be thrown.

- verifyUser(name, password): For the purpose of security and authentication, we implemented this method to make sure that access to a registered user's account would be made only with the right combination of username and password. Status code in the range [1-4] are returned to help us assess the aftermath of calling this method with 1 denoting a successful verification of a user account, a 2 denoting the inexistence of a user account in our database, 3 whenever the password provided for the user name was incorrect and 4 when an unprenventable SQL Exception was to be thrown.

- hasStoredShares(player2, player1): As its name implies, this method is allows us to determine whether player2 has shares stored in our database with player1 as its partner.

- getSharesData(): From all the methods dedicated to communication between our system and our database, this method is one of the most intensive. Upon completion, it returns a Hashmap of String arrays and BigInteger arrays. The former storing references the all second players (player2) usernames, the latter containing their respective sets of shares.

- removeShares(player1, player2): We implemented this method to remove a user's shares from the Shares table after they had been retrieved for further computation. We did this because we intended to keep a workspace as resourceful as possible, in other words, there was no reason to keep storing data that was no longer needed for the purpose of this system.

- clearAllTables(): This method clears all tables in our database. It is done by using mySQL's TRUNCATE operator. We implemented this method for the event where all tables need to be emptied in the database.

- getShares(player2, player1): This method solely returns the shares of player2 is player1 is its parter. We ensured that player2 would always have shares by calling hasStoredShares() method described a few points above prior to calling this method. Unlike methods that manipulate the Shares table and do similar tasks as this one, here we are only returning one set of shares. In a much larger system, known at all time which parameters to pass for computation could prove to save lives, money, etc... another reason decoupling was of utmost priority for us.

- storeShares(player2, player1, shares): This method simply stores shares for a player (player2), from another player (player1). At some point during the back-end computations, once shares have been computed for a certain user, we stored them into a small array and use the relevant parameters to map them into our database. Organization was a key asset in the implementation of this complex system.

- isPresent(username): This method plays an important role in helping catch preventable exception. Prior to executing certain queries, this method is called to determine whether the query should be processed at all.

Overall, we believe our use of a database in this project allowed us to experience developing a system benefiting of a dynamic front-end and a robust back-end.

2.5 Client-Side

2.5.1 Choice of Implementation Language and Framework. We have initially used Java as the implementation language for our client-side service; however, we have to create a Google Web Tools application in Java. This application will be deployed separately from the server-side service, and there are many limitations for cross-domain request call. Because of these limitations, we have to deploy the client-side service in the same server in order for the client-side service to have the same domain as the server-side service. After looking up the Google App Engine documentation, we have concluded that develop a client-side service within the server-side service is the only solution to the cross-domain limitation problem. For this solution, we will have to develop a static website using HTML and JavaScript, and we use JQuery framework for the REST request due to it simplicity and efficiency, which allows us to develop such a client-side service in such a short time.

2.5.2 Choice of Deployment Platform. As mentioned in Section 2.3, we have chosen Google App Engine as our deployment platform. In here, we will explain the second reason why we have chosen Google App Engine. The Google App Engine is designed to be used either for deploying static website or web services; therefore, it allows developers to develop a static website within the web service. The project is main directory: one contains the client-side HTML and JavaScript source codes and one contains the server-side Java source code. When the web service is deployed, both the client-side and server-side web service will be in the same domain, which will not be limited by the cross-domain request policy in JavaScript. Before we start working on deploying our web service on Google App Engine, we have tried Amazon Elastic Beanstalk; however, comparing it to Google App Engine, Amazon Elastic Beanstalk is very complicated and doesn't work well with embedding the client-side service into the server-side service. As a result, we continue to deploy our web service with Google App Engine.

2.5.3 Client-Side Architecture. The client-side architecture consists of two main components: the HTML component (UI design) and JavaScript component (controller). An overview of the client-side architecture is shown in Fig. 2. The HTML Component also include CSS source code, which is responsible for decorating the HTML pages. For security reason, we decided to use cookies for temporarily information storing instead of using URL to pass information. Using URL with reveal user's username and password in the URL string when the client-side service is switching from the login page to the share request page; using cookies can easily achieve privacy and security.

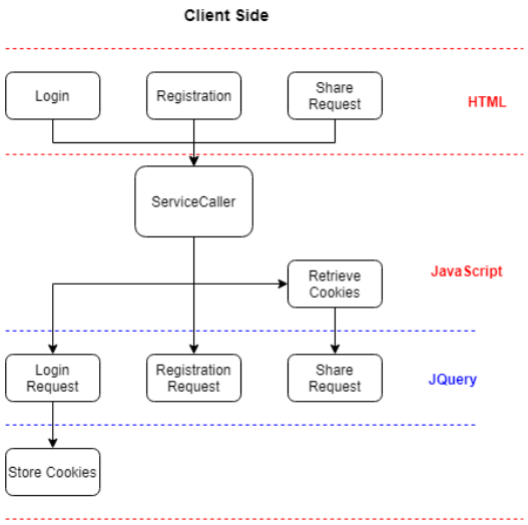

Figure 2: Overview of Client-Side Architecture

2.5.4 HTML Component. The HTML component is responsible for all the front-end user interaction. It consists of three main pages: Account Login page, Account Registration page, and Share Request page. Each of them has a main panel that is located in the center of the web page. The panel is then divided into two sub-panel. One of the sub-panel contains a form, which allows users to input information; the other sub-panel is the information section, which will display important information about the current page. Once the submit button in the form is clicked, The JavaScript controller will take action to interact with the server-side service.

2.5.5 JavaScript Component. The JavaScript component is responsible for making REST request to the server-side service and controlling the HTML Component according to the response. As shown in Fig. 2, the three main REST request methods are constructed using JQuery framework while the rest are constructed with normal JavaScript. All of the REST request methods will validate user's input before making request in order to minimize the input handling in the server-side service. Input validation includes empty field and unqualified length for username and password. After getting response from the server-side service, each method will either display an error message or go to the next page. As mentioned above, the data transfer between each page will be done with cookies. In the Login Request method, after user has logged in with the correct username and password, which the server-side service will response will a corresponding response code, the Store Cookies method will be called to store the account information into the cookies. After the Share Request method is called, it will first check if the cookies contain any username and password before making request to the server-side service. In the situation when there is no cookie found, an error message will be shown to prompt user to login again.

3 DISCUSSION

It may be easy to assume that this work is rather useless after some thought: when the result of this computation is returned, any of the players can simply divide the result by their secret to obtain the other player's secret, thus invalidating the protocol. However, this is not always the case. Imagine a scenario where both parties have no secrets; instead, they have shares of a secret given to them by a third party. This protocol allows them to perform computations with their shares without either of them discovering the original secret. This is the basis of two-server secure computations, a protocol that is becoming more commonplace as of late which allow users to ask a service to perform some kinds of operations or computations on their input without knowing the input, which is the building block for many applications, including Privacy-Preserving Machine Learning.

In addition to this, our protocol is also the building block for comparison between parties. We can use multiplication over the bits of input instead of the input as a whole, and create a protocol which emulates an *argmax* method; we provide to inputs from two parties, and it returns which of the inputs is the greater of the two. This in turn can be extended to create a library of comparison algorithms over encrypted data. As with the two-server secure computations, this can be a building block for many applications, including again Privacy-Preserving Machine Learning.

In our implementation of this project, we utilize a Website to obtain the shares from the Trusted Initializer Web Service and provide those shares as input to a program which performs the actual computation of secrets. This was less than ideal, as the extra steps of having to communicate between UI and command line, as well as having to deal with two separate programs to perform a single task, were tedious to deal with in quick succession. For single computations, it may be fine, but these aforementioned building blocks often require multiple sequential computations, making our implementation inadequate for use in the expanded library. A simple fix would be to simply make all of it command line based: we have our command line program make all interactions with the Web Service, and the only thing the user would have to do is provide the information with which to make the call. However, the optimal end result of this would be an entire application, complete with a GUI, that represented each of the protocols in one easy space for the client-side. This would allow a user to easily compute with a partner who possessed the same application, and even allow for more optimization of the interaction between partners.

Surprisingly enough, the most difficult section of this to implement was the client-side aspect; implementation of both the database and actual protocol were relatively simple. Navigating the server-side logic in Jersey, a Java library designed for implementing Web Services, was also difficult, but ended up taking less time to implement. None of the authors of this project has any experience operating in Jersey beforehand, which added an element of difficulty to the creation of said Services.

4 CONCLUSIONS

We have detailed our implementation of a Trusted Initialization Web Service, and the comprising components: a server-side implementation of number generation, a database to hold stores for players, and a client-side implementation of both a Website and a command line implementation of the protocol. With this Web Service in place, people now have a secure, arbitrary and unbiased Initializer with which to initialize the implemented command line protocol. As stated in section three, this protocol is the building block to the secure comparison of integers between two parties. Tying back to the introduction of Yao's Millionaire problem, our protocol paves the way for a computationally trivial solution to this decades-old problem. The implementation of such a protocol could be left to future work, but the next section will detail ways to improve our building block such that it is more secure and capable to be expanded upon.

5 FUTURE WORK

In the future, we plan on extending the potential of our system by developing a multi-shares feature, strengthening the security of transactions and a combination of the user-friendly website interface with command line capabilities.

5.1 Batch Processing of Shares

This feature would allow players to process a large number of shares for multiple computation simultaneously. This would be the most ideal version of this system as it could be integrated in day to day applications such as banking, wire transfer, file sharing, etc.

5.2 Improving Cryptographic Security

Currently, our web service communicates in the clear. By that we mean that the security of our transactions depends on the security of the frameworks and tools we use to develop our web service. We would heighten the security level by devising a system such that minimal information is in the clear. There exists many data encryption tools we could possibly use or mimic to accomplish this task such as VeraCrypt, GNU Privacy Guard, 7-Zip, etc.

5.3 Website and Command Line Access

The system in this case would be operable as it currently is from the website and also from the command line of any of following popular Operating Systems: Windows, Linux, OS X. Depending on the specific of such implementation, the system's access via command line could increase the current computation potential, given that a computer's file system is more accessible that way. The website could also benefit from additional features, particularly the accounts of registered users. We would create a 'History' section for each user in which he or she is able to view past transactions and ideally repeat any by the click of a button.

REFERENCES

[1] Eric Lawrence *Same Origin Policy Part 1: No Peeking*., (Auguest, 2009). Available at https://blogs.msdn.microsoft.com/ieinternals/2009/08/28/same-origin-policy-part-1-no-peeking/