Yale University

Distributed Generation of Consistent Unbiased Values

by

Christopher Zeng

Advisors:
Michael Fischer, Joan Feigenbaum

in the

Department of Computer Science

April 2014
1 Abstract

This paper outlines a protocol and implementation for the distributed generation of consistent unbiased values amongst a group of $n$ peers. The paper outlines naïve protocols that attempt to solve this problem and specific attacks that can bias the outcomes. The protocol applies Shamir Secret Sharing and Feldman’s Verifying Scheme to generate consistent random values within a group. The protocol guarantees that for a group of $n$ members with limit $k$ and no more than $\text{Min}(k,n-k)$ conspiring untrustworthy members, a generated value will be unbiased by any of the members. This protocol is implemented using C++ and Qt along with the Qt Cryptographic Architecture and the GNU Multiple Precision Arithmetic Library.

2 Introduction

Generating trustably random values is an important problem for cryptography. Most cryptographic protocols require the generation of some random number. Attackers with knowledge or even control of these random numbers can often use specialized attacks that can (in some cases) break these cryptosystems. Therefore it is very important to be sure that these random values are not biased or controlled by anyone. On a single closed system, this is not so difficult. The random number can be generated by any number of well defined cryptographic random number generation algorithms. On a distributed system, however, each other member cannot necessarily trust the other users to generate truly random numbers. In fact, a member could simply pick any number they wish and claim that they have picked the number randomly. Generating a shared random value in a distributed manner, that can be trusted by a group of users to not have been controlled or biased by any other member, poses a difficult problem for which this paper intends to provide a protocol to solve.

A shared trustably random value has many applications in distributed computing. When a distributed system needs to select a consistent random value between its members, it needs to trust that the value has not been tampered with. NIST has been developing a Randomness Beacon that would distribute, at regular intervals, a block of 512 bits every 60 seconds. These random blocks would be generated by a quantum source for truly random values and then be signed by NIST for publication to the web. The blocks would be timestamped and contain a hash of previous values to keep the values from being falsified. Applications of such a beacon include secure-multiparty-computation like a secret ballot and massively-multiplayer online games [1]. This beacon still requires the user to trust that NIST is truly generating these values randomly. Perhaps a group does not trust the creators of this beacon to be actually generating random values. A group might want to trust that the random number isn’t being controlled by any third party. The protocol in this paper allows a group to select a random value without access to a third trusted party like the NIST beacon.

3 The Protocol

First, let’s define the problem this protocol attempts to solve. Let’s say that we have a group of $N$ members and they want to generate an uncontrollably random bit $b$ between them such that if there are at least $k \leq N$ trustworthy members who will correctly follow the protocol, then no member could control the outcome of $b$ found by all the trustworthy members.

Naïve approach number one is to assign one member $m$ of the group to flip a coin and tell everyone what the result is. This quite obviously fails these requirements as no other member of the group can confirm that $m$ truly flipped a coin and reported the correct value. In fact $m$ could just simply tell the other clients whatever he wants or needs the bit to be.
Naïve approach number two seems much more fair, but in reality can still be manipulated. Each member $i$ from 0 to $N-1$ chooses its own bit value $b_i$ and sends it to every other member in the group. Once a member has received every other member’s $b_i$, the member finds $b$ by taking the XOR of every $b_i$. If members $m = 1$ to $m = n - 1$ conspire together to try and force $b$ to be 0, they can select all $b_i$ except $b_0$ to be 0. They cannot predict anything about the outcome of $b$ however as $b$ has a 50% chance of being a 0 and a 50% chance of being a 1. Member $m = 0$’s choice of $b_0$ completely randomizes $b$.

However, the outcome of this protocol can still be biased in a real application. This is because the bits cannot all be sent simultaneously. One member $i$ may wait to make his selection of $b_i$ until after he has received all the other members’ bits. Member $i$ can then choose the $b_i$ that will give it the $b$ it is looking for. It then broadcasts its selected $b_i$ to all the other members with them none the wiser that it has controlled the outcome.

What is needed here is a way for members to lock in their choices before they can read out the final result. This can be achieved using cryptographic commitments. The members all select a value and compute a commitment to that value. This commitment can only be linked to a single value and has an associated key that can unlock the commitment to reveal this value. This commitment is sent out to every other member. Once a member has received a commitment from each other member of the group, it can broadcast its commitment key to the whole group for them to unlock the commitment and reveal each member’s value.

Unfortunately this protocol still has a biasing vulnerability. After a malevolent member Mal has gone through the commitment phase, it can hold on to its commitment key until it has received the rest of the group’s keys. From here, Mal can compute what the final output would be before letting any other member do so. If Mal does not like the result, Mal can simply refuse to send out its value and stop the protocol. Mal can then prevent the protocol from ever generating values it does not want. Since the most likely reaction to a protocol failure is to run the protocol again, Mal can continue to reject unwanted values until she finds an outcome she likes.

Up to a certain point in the protocol, adversaries may be allowed to misbehave and force the protocol to stop. After this point, the outcome of the protocol will have been determined and the adversary should not be able to stop the protocol. This point is called the barrier. In the previous protocol, we would like to erect this barrier at the point that all of the commitments are received. We want to prevent an adversary from being able to halt the protocol after this barrier.

To solve this problem, the protocol used in this paper divides the commitment key into a separate share for each member of the group. A group of $k$ of these shares can be combined to recreate the commitment key. The barrier point in this protocol would be when each member has received a share of every other member’s commitment key. After this point, each member broadcasts the key shares it has received to every other member. Once a member has gathered $k$ shares for a given key, the key can be reconstructed and used to unlock that member’s commitment to reveal his selected value. The protocol pseudocode is shown in Appendix A.

3.1 Protocol Limitations

Before the barrier, any single adversary could refuse to send out its commitments or key shares to the group causing the protocol to come to a halt. This is alright, however, as the adversary cannot know anything about the final outcome. All of the other members’ selected values are hidden in commitments and the adversary does not have enough key shares (for $k > 1$).
to recreate the key and unlock the commitment. After the barrier, a single adversary could wait until it has unlocked every other member’s commitment to send out the key shares it has acquired but this will not stop the protocol (for $k < N$). The other members of the group have enough shares between them to recreate all of the commitment keys and unlock all of the selected values. In this way, there is nothing a single adversary can do after this barrier to prevent the protocol from finishing.

For $1 < k < N$, this protocol will be unbiased if all members are working alone. However subgroups of $k$ or more members working together will be able to bias the outcome of the protocol. A subgroup of $k$ members can gather $k$ shares for every commitment before the barrier. From these $k$ shares, they can reconstruct all of the keys and unlock all of the commitments before they have to send out any of their own key shares. They can then halt the protocol after learning the final outcome.

A subgroup of more than $N - k + 1$ members, however, can bias this protocol after the barrier. After the barrier, none of the members will be able to reconstruct a key without at least one share from this subgroup. The subgroup could then wait to broadcast their key shares until after they have collected enough shares to reconstruct the keys and unlock the commitments. The subgroup can then choose to halt the protocol after they have found the final result. The combination of these two limitations gives us a limit on the number of conspiring untrustworthy members this protocol can handle: $\text{Min}(k,N - k + 1)$. To maximize this value, $k$ should be set to $N/2$. This protocol will never be unbiased in groups with more than half the members conspiring together.

4 Secret Sharing and Commitments

For this implementation, three different numbers are selected. Two primes $p$ and $q$ are selected using such that $p = 2q - 1$. A third value $g$ is selected that is a generator in modulo $q$ space. Then a polynomial $f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{k-1} x^{k-1} \pmod{\phi(p)}$ is selected where $a_0$ is the initial chosen value. Instead of calculating one commitment on $a_0$, commitments are calculated on the $k$ coefficients in the polynomial $f(x)$. The commitments are calculated as

$$c_i = g^{a_i} \pmod{p}. \quad (1)$$

The difficulty of the discrete logarithm problem prevents $a_i$ from being found from $c_i$. All $k$ commitments are then broadcast to every member the group. Since the chosen value is a coefficient that was committed, the value cannot be changed without violating this commitment.

The key shares are then generated as the pair $x, f(x)$ for integer values of $x$ from 1 to $N - 1$. They represent points on the function $f(x)$. Since $f(x)$ is a polynomial of degree $(k - 1)$, the polynomial can be uniquely defined using $k$ points [2]. Therefore a collection of $k$ shares can be used to recreate the coefficients $a_i$ of the polynomial $f(x)$ using the equation

$$a_x = \sum_{j=0}^{k-1} y_j \prod_{i \neq j} \frac{x-x_i}{x_j-x_i}. \quad (2)$$

This includes then chosen value $a_0$.

Feldman’s Scheme [3] ensures that a given key share correlates to the polynomial $f(x)$ that was previously committed. In order to verify that a given share $x, v$ is valid, a member can check that $g^v = a_0 c_1^{x} \ldots c_k^{x} \pmod{p}$ (note that this is all done in some modulo $p$ arithmetic). This works because

$$\prod_{j=0}^{k} c_j^{x_j} \pmod{p} = \prod_{j=0}^{k} g^{a_j x_j} \pmod{p} = g^{\sum_{j=0}^{k} a_j x_j \pmod{\phi(p)}} = g f(x) = g^v \quad (3)$$

where $\phi$ is the Euler Totient function. Because of how $p$ is defined, $\phi(p) = 2q$. Since
$v$ is found in modulo $q$, this equation will hold for either $f(i)$ or $f(i) + q$.

5 Implementation

For this project, the protocol was implemented in C++ using the Qt framework. Qt was used for its simple and easy GUI creation. Figure 1 shows an image of the user interface for one member.

Figure 1: The DRNGDialog box

The top value displays the result of the most recent run of the protocol. $N$ shows the number of peers in the group while the $K$ text box controls the value of $k$ used for the protocol. Pressing the START button begins the protocol for the peers in the group. The bottom text box is used to connect to peers and create a group. Each member is connected to a specific port which is displayed in the window title. To connect to another peer, specify the peer by writing “address:port” in the add peer text box.

The GNU Multiple Precision (GMP) Arithmetic Library is used in this project to handle the large integers used in the protocol. GMP allows for arbitrary precision and integer size. It also has a number of built in functions such as modulo exponent and modulo inverse that were needed for this project. Before GMP was used in this project, I wrote functions to perform these processes on ints. Currently hardcoded into the program is a large Sophie-Germain prime $Q = 22801763489$ that is used in the commitment algorithm along with a generator in modulo $Q$ space $G = 11011736$. A Sophie-Germain prime is a prime $q$ such that $2q + 1$ is also prime. Selection of these values by the protocol would be better but requires a reliable broadcasting algorithm. This was out of the scope of this project and so these values are currently hard coded in.

The Qt Cryptographic Architecture is used for encoding and decoding the initial key share message before the barrier using RSA. This is done to prevent any adversary listening in to the communication of trustworthy members and gathering more than one key share before the barrier.

The UI window is contained in DRNGDialog.cc while the actual generator is contained in DRNG.cc. In DRNG.cc, each incoming message is captured by the function DRNG::incomingMessage(). From this function, the necessary data is sent to the proper function.

5.1 Creating Peer Group

The peer group is maintained by each instance in the Peers List. Each Peer is represented by a Peer object which contains information like a Peer’s address and port number. The object also contains the the Peer’s RSA public key for encoding messages to this Peer. When the protocol is run, the Peer object is where the commitments and shares are held. This way, the commitments and shares are not confused with the wrong Peers.

When a Peer is added in the UI, a Peer is created but it is not immediately added to the Peers list. First the address is resolved using an asynchronous process. Once the address is resolved, it is added to the list (if it is not already in the list). This process is similar to the process is used by the peerster program assigned in the Building Decentralized Systems (CPSC 426) class. After the Peer is added, a message is sent to the newly added Peer telling it to add this instance as a Peer. A message is also sent to every other Peer currently in the group with the new Peer’s information to keep the group updated on the new member. These messages contain
the Peer’s address, port number, and RSA public key.

Messages between Peers are sent as QVariantMaps. The Key values of the map tell you what is contained in the message while the values contain the actual data of the message.

5.2 Protocol Before Barrier
The protocol is started when one of the members hits the START button and sends the protocol to the DRNG::Start() function. In this function the member send a START message to every one of its Peers, which sends these Peers into the DRNG::Start() function as well. This advances the member into stage 1 (it starts in stage 0). From here, the \((k-1)\) degree polynomial is selected and the coefficients are committed using Feldman’s Scheme. These commitments are broadcast to every peer. Then a different share is sent to each of the peers.

When a member recieves commitments from another peer, it is saved to the Peer in the DRNG:SaveCommitments() function. When the first share from a Peer is recieved, the member is sent to the DRNG:InitialShareRecieved() function. In this function, the share is verified using Peer:VerifyShare() which runs the verification check shown in equation (3). If the share passes, it is save in the Peers Initial-Share variable. If a share ever fails to be validated, a FAILURE message is sent to all of the members peers and the protocol is aborted. If the member collects an initial share for each one of its Peers, it sends a SUCCESS message to its Peers. If the member then recieves SUCCESS messages from every one of its Peers, then the protocol moves beyond the barrier point. After this point, no single adversary can cause the protocol to stop.

5.3 Protocol After Barrier
After receiving all of the SUCCESS messages, the member broadcasts all of the InitialShares that it as stored in its Peers. Received shares are sent to the DRNG::AddShare() function. Here they are verified and added to the Peer’s Shares vector. The shares are validated in the Peer::ValidateShare() function. If a member recieves a share that cannot be validated, the share is dropped. Once \(k\) Shares are gathered by a specific Peer, they are combined using the Peer::CombineShares() function. The Peer::CombineShares() function combines the shares in the Shares vector using equation (2) for \(x = 0\). Once a member has combined shares for each of its Peers, it XORs the result to arive at the final value to be displayed. The ResultChanged() signal is emitted to notify the UI of the change.

6 Conclusion
This paper has outlined an implementation of a distributed consistent random number generator. The protocol makes use of Shamir Secret Sharing and Feldman’s Scheme to implement a verifiable secret sharing scheme. Each member of a group of size \(N\) selects a random value and splits the value into \(N-1\) shares with limit \(k\) following Shamir’s Secret Sharing algorithm. The equation from the share splitting is committed to using Feldman’s Scheme and the commitments are spread to every other peer. Then each peer gets a different initial share from each other member. After SUCCESS messages are sent and recieved guaranteeing that \(n-1\) shares are spread for each member, each peer broadcasts all of the initial shares it recieved so that the shares can be combined. Once the shares are combined, they are XORed to get the final consistent result. This protocol was shown to work in groups containing less than \(\text{Min}(k,N-k)\) conspiring untrustworthy members. It was shown that \(k = N/2\) actually gives the largest tolerance for conspiring members. This protocol could have numerous aplications in secure-multiparty computation and distributed computing algorithms.
Bibliography


Appendix A: Protocol Pseudocode

---

**Protocol Pseudocode**

**Member** $i$

Select value $b_i$
Compute commitment $c_i$ to $b_i$ and key $S_i$
Broadcast $c_i$ to group
Split $S_i$ into shares $s_0, \ldots, s_{N-2}$ with combination limit $k$
Send a different share $s_k$ to each member of the group

Wait until a commitment $c_j$ and share $s_k$ is received from and by each member

—-BARRIER—-

Broadcast all received shares $s_k$ to the group

Once $k$ shares of a given key $S_j$ are received
Combine shares into key $S_j$
Unlock commitment $c_j$ with key $S_j$ to find value $b_j$

Once a $b_j$ is found for each member
XOR all $b_j$ to generate the final result

---