Implementation of a DKG algorithm over a P2P network

Sahil Siddiq
Department of Computer Science and Information System
BITS Pilani, Pilani, India
f20180293@pilani.bits-pilani.ac.in

***Abstract*—In traditional public key cryptography, the generation of keys takes place in a single location or by a single party. This involves the placement of trust in one location. This also acts as a single point of failure. Many attack vectors exist and have been used to compromise such parties for malicious objectives.**

With threshold cryptosystems and distributed key generation, issues of a single point of failure can be circumnavigated and trust can be eliminated or distributed amongst several parties. This makes the system far more secure. It becomes far harder for an adversary to attack a threshold cryptosystem. In such a cryptosystem, no single party is solely responsible for key generation and the private key is not computed, stored or retrieved in a single place.

This project deals with building a P2P network and implementing a Distributed Key Generation algorithm based on Elliptic Curve Cryptography over a finite field. The algorithm uses Shamir's Secret Sharing for splitting and storing the private key over the network in a distributed manner while the public key is stored in each node.

***Index Terms*—Peer-to-Peer Network, Distributed Key Generation, Threshold Cryptosystems, Public key cryptography, Elliptic Curve Cryptography, Elliptic Curve Discrete Logarithm Problem**

I. Introduction

Distributed Key Generation is a process in which multiple parties come together to generate a public and private key pair. This is different from the more traditional public key cryptographic processes that we are used to seeing in everyday life. With the rapid advancement of technology in today's time, and the impact that it is making in a countless number of sectors, subjects such as security and privacy have also been becoming very significant.

Traditional public key cryptography works with several assumptions including one of trust. Consider the following example.

One common use of public key cryptography is in the implementation of the HTTPS protocol. When a user connects to a website via the HTTPS protocol, the web server presents a digitally signed certificate that is meant to prove the identity of the website. This is based on a chain of trust that originates at the root certificate authority. Each certificate in the chain is signed by the certificate above it and this stretches all the way back to the root certificate authority. However, there is no entity to verify the root certificate authority since they are at the top of the chain. This leads to several problems. For example, trust is placed in the root certificate authority and it is assumed that they will verify a given certificate without problems. But there may be situations in which the root certificate authority cannot be trusted and there have been cases in the past where the authority has issued misleading certificates [1].

A private key is used to sign a certificate. If the root certificate authority's keys are stolen, this might cause a huge problem as the attacker can then use the stolen keys to make valid digital certificates for any domain they like including for malicious websites. One real world example is the case of the Dutch certificate authority – DigiNotar [2]. In this manner, the public key infrastructure places a lot trust in the root authority.

Threshold cryptosystems aim to reduce this trust as much as possible if not eliminate trust altogether. In threshold cryptography, n parties are involved for the generation of public and private keys. With a secret sharing scheme such as Shamir's Secret Sharing, of the n parties that are involved in the generation of keys, at least t parties will be required to retrieve the private key [3]. This private key can be used for applications such as making valid digital certificates or deciphering encrypted information [4]. Such a threshold cryptosystem no longer places trust in a single centralized party. However, this does not fully eliminate trust from the cryptosystem. It merely distributes trust between parties. This distribution of trust makes the cryptosystem more reliable and less vulnerable to attacks. It also helps in getting rid of the issue of a single point of failure unlike traditional centralized cryptosystems.

II. Background

Traditional public-key cryptosystems involve the generation of public and private keys in a single location. This forces the placement of trust in a centralized party that is responsible for the generation of keys and/or is responsible for other related use-cases such as making valid digital signatures or deciphering encrypted information.

A threshold cryptosystem aims to make the process of key generation more secure by distributing trust between the involved parties. Since multiple parties are involved in generating the keys, no single party needs to be trusted. During the generation of the key pair, the private key is never computed, stored or retrieved in a single place.

Secret sharing which is a part of distributed key generation also makes the system more secure. While the public key is known to all the nodes, no node has any information regarding the private key. Using Shamir's Secret Sharing, the secret key is split into shares which are distributed between the nodes and in order to recover the secret, a minimum threshold t of shares is required [3]. Thus, in order for an adversary to attach the system and retrieve the share, the attacker will have to compromise at least t nodes. For large values of t, this is highly infeasible as compromising at least t nodes, each of which might have their own defence systems, is computationally very expensive and nearly impossible. Also as long as less than t nodes are compromised, the attacker can never recover the secret.

The private key can then be used for applications such as making valid digital signatures in a distributed manner without ever having to reconstruct the private key in a single place [4].

The distributed key generation algorithm can be built based on existing cryptographic schemes such as RSA or Elliptic Curve Cryptography. However, as we will see later, the setup time for RSA is quite long [5] as it involves the generation of two random prime numbers in a distributed manner. Also, based on how Elliptic Curve cryptography works, ECC is able to provide the same security strength of the private key as RSA for far smaller key lengths [6]. Hence, with this in mind, our project deals with the implementation of distributed key generation over a P2P network using Elliptic Curve Cryptography.

III. Related Work

The first Distributed Key Generation scheme was proposed by Pedersen [7] in 1991. Pedersen's Distributed Key Generation scheme is based on the idea of having n parallel executions of a protocol in which each node acts as a dealer of a random secret z_i. The protocol that is used in this scheme for distributing the shares of the secret z_i is called Feldman's Verifiable Secret Sharing.

Langford in her paper [8] showed that in a careless implementation of Pedersen's Distributed Key Generation scheme, the attacker could influence the choice of the secret key. In the same paper, she proposed a solution that could prevent this from happening.

Rosario Gennaro et al [4] produced a series of proofs that demonstrated that Pedersen's DKG scheme cannot guarantee the correctness of the output distribution of the secrets in the presence of an adversary. They showed that Feldman's Verifiable Secret Sharing is vulnerable to malicious contributions to Pedersen's distributed key generator. This would allow the scheme to leak information about the shared private key. They showed that an adversary could manipulate the distribution of the shared secret to something that is quite different from a uniform distribution. This allows the adversary to gain some a priori knowledge about the shared secret that would not be possible if the secret is truly chosen at random.

In the same paper, the authors proposed a new Distributed Key Generation protocol that guarantees uniform output distribution of the secrets. This is achieved by designing an initial commitment phase in which each party commits to its initial secret z_i. The commitment phase takes place in each node when it generates its own private secret z_i. It does this by committing to a t-degree polynomial with z_i for the value of the constant coefficient. This prevents an adversary from biasing the output distribution of the protocol.

IV. Proposed Methodology/Solution/Design

Before we are ready to start implementing a distributed key generation algorithm, we need to build a P2P network wherein the nodes can communicate with each other. The nodes communicate with each other not only to keep a track of all the available peers but also to ensure that computation of the keys takes place smoothly in a synchronized manner.

A. *The Peer-to-Peer Network*

The Peer-to-Peer network that has been developed for the implementation of this project is quite straightforward. The nodes that make up the P2P network form a

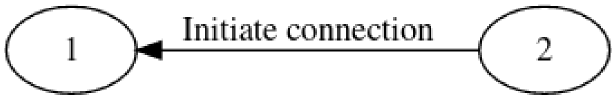


Fig. 1: Node 2 initiates a connection to Node 1

mesh topology. Every node in the network is connected to every other node. The presence of n nodes in the network requires that the network have n^2 links between the nodes.

In the implementation, each node is simply a process that runs with a different port number associated with itself. All the nodes use 127.0.0.1 (the standard IPv4 address for the loopback interface) as their ip address.

1) Node IDs: Every time a new node is to be added to the network, the node has to be initialized, i.e. the process representing the node has to start running. Each node is given a unique identifier called the $UUID$ (universally unique identifier). This is a 128-bit label used for identifying the different nodes in the network. Since it is of fixed length, it not hard to see that, theoretically speaking, two nodes can have the same $UUID$ (pigeonhole principle). However, the probability of there being a collision is extremely small [9] and for practical purposes, the $UUID$ can be treated as being unique for each node.

2) Initial Network Setup: Initially only one node exists. This node acts as a server listening for connections from nodes that might want to join the network later on. To enable a new node to connect to this first node, a new node is initialized with a new port number associated with it. This new node initially acts as a client and tries to connect to the first node as shown in Figure 1. Once connected, the first node adds the second node to its list of peers. The two nodes then exchange roles with the first node acting as the client and the second node acting as the server. Now, the first node connects to the second node as a client from the second node's perspective as shown in Figure 2. It sends the new node the list of peers currently present in the network. The new node then updates its own maintained list of peers in the network. In this way, a simple P2P network consisting of the first two nodes is established as shown in Figure 3 and both the nodes maintain a list of peers currently present in the network.

3) Joining a Network with Multiple Nodes already Present: In the network, each node acts as both the server and the client. Suppose there are currently $n-1$ nodes present in the network as shown in Figure 4. These $n-1$ nodes have ids $X_1, X_2, \dots, X_{n-1}$. Each of these

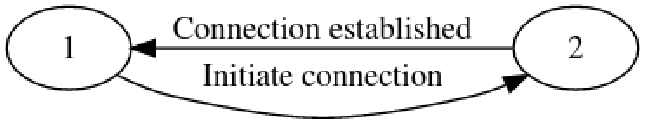


Fig. 2: Node 1 connects to Node 2.

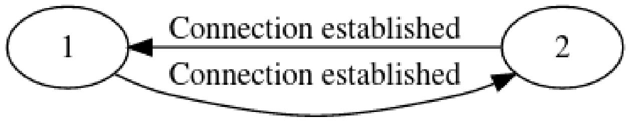


Fig. 3: A P2P network with 2 nodes is established.

$n-1$ nodes is listening for connections from a new node. These nodes also maintain their own list of peers. Each list contains information regarding the $UUID$s, ip addresses and port numbers of the nodes present in the network. Now, a node with id X_n has been initialized and is ready to join the network. X_n can initially choose to connect with any of the $n-1$ nodes.

Let's say X_n connects to X_1 as shown in Figure 5. In this case, X_n sends a message to X_1 asking it to be registered as part of the network. X_1 adds X_n to its list of peers. However, based on the design of our network, X_1 also needs to connect to the other nodes and the other nodes need to update their own list of peers. For this to happen, X_1 sends its updated list of peers to X_n as shown in Figure 6. At the same time, X_1 establishes a reverse connection back to X_n.

Once X_n receives the list from X_1, it tries to connect to every node in the list with which it has no connection as shown in Figure 7. The process of establishing connections between X_n and X_i ($\forall i \in [1, n-1]$) then repeats until no more connections are left to be made in the network. The final network looks as shown in Figure 8.

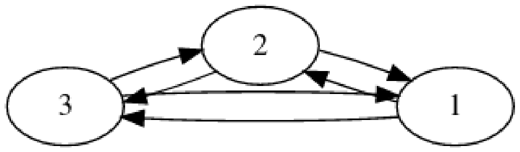


Fig. 4: P2P network

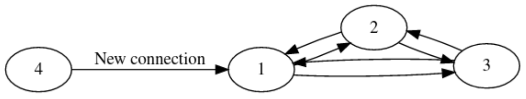


Fig. 5: A new node connects to one of the nodes in the network.

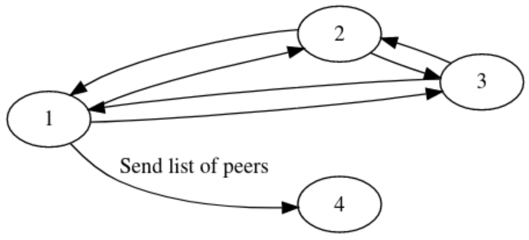


Fig. 6: The new node gets the the list of peers in the network.

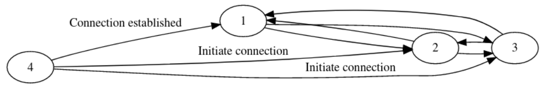


Fig. 7: The new node connects to all the nodes in the network.

B. Cryptographic Schemes

Now that we have the P2P network in place, we can work on designing and implementing the distributed key generation algorithm over the constructed P2P network. The security strength of keys depend on how unpredictable they are. One way of introducing unpredictability is through randomness. This randomness can be manifested through the random generation of a key itself or the random generation of some values which are in turn used to generate the required keys.

Two popular cryptographic schemes on which we can build our distributed key generation algorithm are RSA and elliptic key cryptography (ECC).

C. Traditional RSA

In traditional RSA, the random values that are generated are two prime numbers whose product gives us the RSA modulus N. RSA is an asymmetric cryptographic scheme which means that we need to generate two keys — the public key (for encryption or verifying a signature) and the private key (for decryption or creating a signature).

To generate the private key we need to calculate the totient function, $\phi(N)$, of N. This function gives us the number of numbers that are coprime with N. $\phi(N)$ is calculated as

$$\phi(N) = (P-1)(Q-1)$$

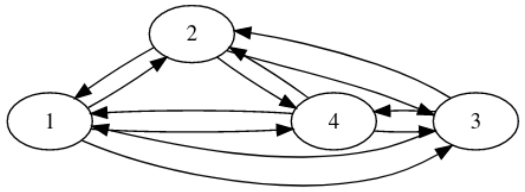


Fig. 8: The final P2P network.

where P and Q are the two randomly generated prime numbers.

The public key should satisfy the property of being coprime to $\phi(N)$ and N. Once the public key is generated, we can generate the private key. The private key is calculated as the modular multiplicative inverse of the public key modulo $\phi(N)$

$$d \equiv e^{-1}\,(mod\,\phi(N))$$

where d is the private key exponent and e is the public key exponent.

D. RSA in the Distributed Context

In the distributed context, the objective is to come up with a way to randomly generate the private prime numbers in a distributed manner such that no single node knows what the prime numbers are. However, this is a very challenging task. The algorithm suggested in H.L. Nguyen's paper [10] on Threshold RSA Cryptography aims to randomly generate the prime numbers by making each node in the network randomly generate two secrets p_i and q_i. The prime numbers P and Q will then be given by

$$P = \sum_{i}^{n} p_i$$

$$Q = \sum_{i}^{n} q_i$$

The common mudulus N will then be given by

$$N = (\sum_{i}^{n} p_i)(\sum_{i}^{n} q_i)$$

However, this poses a problem as the above algorithm does not guarantee that the sums P and Q are actually prime numbers. In order to solve this problem, algorithms as suggested in Boneh and Franklin's seminal paper on Threshold RSA Cryptography would have to be used. Their paper provides an efficient way of checking biprimality of a modulus without knowledge of its prime factors. While such algorithms exist to solve these problems, such computationally heavy tasks might make the overall implementation of the algorithm quite slow. Distributed RSA in particular has an expensive one-time start-up cost that might even make it impractical in some high-performance tasks.

E. Elliptic Curves

While RSA works on the basis of prime factorization, Elliptic Curve Cryptography (ECC) works on the basis of the geometry of elliptic curves. Unlike in RSA where the public key and private key are generated based on computation of other values, in ECC the private key itself is a randomly generated number.

A generic elliptic curve is given by the equation

$$y^2 = x^3 + Ax + B$$

The above equation is also called the Weierstrass form of the elliptic curve. ECC uses elliptic curves over a finite field p. This is denoted as $\mathbb{F}$p where p is a prime number. This limits the field of points to a $p \times p$ grid where all the points lying on the curve have integer coordinates (x, y) and $0 \leq x, y < p$. It should be notes the point $(x, y) = (0, 0)$ is a special point and is called the *point at infinity*.

1) Operations on Elliptic Curves: There are two important operations [11] that are defined over the set of points on an elliptic curve — elliptic curve point addition (also called geometric addition of points), and scalar multiplication. The mathematics and transformations behind this property are quite complex and are not explored in this project to keep things simple.

The first operation of elliptic curves shows that given two distinct points lying on the curve, adding the two points gives us a new third point that also lies on the curve as shown in Figure 9, that is

$$P + Q = R$$

where P, Q and R are points lying on the curve. The addition of elliptic curve points is also commutative in nature.

$$P + Q = Q + P$$

The second operation shows that when a point lying on the curve is multiplied by a scalar value, we get a new point which also lies on the same elliptic curve

$$Q = nP = \underbrace{P + P + \cdots + P}_{ntimes}$$

where P and Q are points lying on the elliptic curve and k is a scalar value. This operation is distributive over addition.

$$Q = nP = (n_1 + n_2)P$$

where $n = n_1 + n_2$.

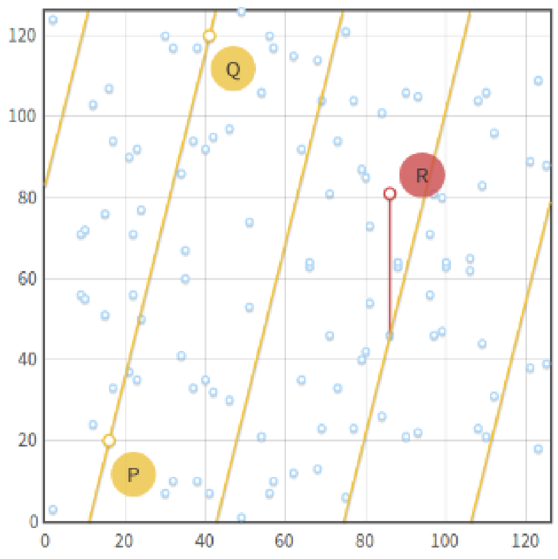


Fig. 9: Addition of points P and Q lying on the elliptic curve gives point R. [11]

2) Order, Cofactor and Subgroups: The set of all points lying on a finite field elliptic curve forms a cyclic group. When two points belonging to the same group are added or a point in one group is multiplied by a scalar value, the resulting point is a point that lies in the same group and on the same curve as the input points. Depending on the values of the coefficients of the curve's equation, the points lying on the curve may form a single cyclic group or they may form several non-overlapping cyclic subgroups. In case of the latter, each of these subgroups will contain a subset of the elliptic curve's points.

Consider the second case. Let the number of cyclic subgroups or partitions be given by h and let the number of points in each subgroup be denoted by r. n which represents the total number of points lying on the curve in the finite field is called the order of the curve while h is called the cofactor of the curve and r is the order of each subgroup.

The cofactor is typically expressed as

$$h = n/r$$

where h is the cofactor, n is the order of the curve and r is the order of the subgroups.

3) Generator Point: Another important attribute of the curve is the generator point. This point is also sometimes called the "base point". The generator point of a subgroup is defined as that point which can generate any other point in the subgroup when multiplied by a scalar k $\in [1 \ldots r]$.

Since the points in a subgroup form a cyclic group, we have

$$rG = 0G = infinity$$

When the number of subgroups or the cofactor of the finite field curve equals 1, the generator point results in the generation of each and every single point in the finite field when multiplied by a scalar k $\in [1 \ldots n]$.

The order of a subgroup which is obtained from the generator point plays a significant role in cryptography as this number determines the number of all possible private keys for the curve in use (more on this in the following subsections).

In order to ensure that the private key space is large enough for a certain amount of security strength, cryptographers have to carefully select the parameters of the elliptic curve — curve equation and coefficients, generator point, number of cofactors, etc. Different generator points for the same curve can lead to the generation of subgroups having different orders. This means that some subgroups might have a smaller order than other subgroups. If the subgroup used for the private key space is small, the overall security strength of the cryptosystem will be low and this will make it susceptible to *small-subgroup attacks*.

4) Elliptic Curves in the Wild: Several well known elliptic curves exist, many of which play an important role in Elliptic Curve Cryptography.

For example, the curve $secp256k1$ is used for ECC in Bitcoin. The number 256 in the name is because it is a 256-bit curve with p, the modulus, and n, the order, being 256-bit numbers. As we will see later, this curve provides 128-bit security strength.

F. Elliptic Curve Cryptography

In Elliptic Curve Cryptography, the private key is randomly generated, while the elliptic curve used, the curve's parameters such as the generator point, the cofactor of the curve, the order of the curve, etc. are fixed. We know from the previous section that multiplying a point on the curve by a scalar results in another point that lies on the same curve. The generation of the public key is based on this principle. The public key is generated by multiplying the generator point with the private key (which itself is just a large number).

$$P = kG$$

where P is a point on the elliptic curve (this point acts as the public key), k is the private key and G is the fixed generator point/base point of the curve.

The reason ECC is secure is because the above equation acts as a trapdoor function. Calculating $P = kG$ is very fast and takes very little time even for 256-bit elliptic curves. However, it is extremely slow and in most cases infeasible to calculate $k = P/G$ when given the values of P and G.

This asymmetry in computational speed is the basis of the security strength of Elliptic Curve Cryptography and the problem is very similar to the discrete log problem.

1) Elliptic Curve Discrete Logarithm Problem: The discrete logarithm problem is defined as follows [12] – given a prime modulus p, and a generator g chosen from a group $\mathbb{Z}p$, it is easy to calculate the value of

$$y \equiv g^x \; (mod\, p)$$

for some value of x. However, it is infeasible to calculate the value of x when the values of y, g, and p are known.

Similarly, in the Elliptic Curve Discrete Logarithm Problem (ECDLP), when given an elliptic curve over the finite field $\mathbb{F}p$ and the generator point G and the public key P, computing the value of k is infeasible. The multiplication of points on the curve in the group $\mathbb{F}p$ is similar to the exponentiation of integers in the group $\mathbb{Z}p$ in the discrete log problem.

2) Security Strength: Caimu Tang's paper on ECDKG shows that to solve the elliptic curve discrete log problem with key of size k, at least $\sqrt{k}$ steps are required. Thus, to achieve k-bit security strength, at least a $2 * k$-bit elliptic curve is required. This means that the field size given by p must be a $2 * k$-bit number.

For example, the elliptic curve *secp256k1* (having the modulus p of size 256 bits) provides around 128-bit security strength.

3) Public Key and Point Compression: In the above subsections, we saw that the public key is calculated as $P = kG$. However, P is a point with coordinates (x_P, y_P) lying on the elliptic curve. This needs to be converted to a number just as the private key is a number. This is done by a method called *point compression*.

Consider the elliptic curve equation in its Weierstrass form over a finite field $\mathbb{F}p$. For a given value of x, there are at most two values of y that satisfy the equation, i.e there are at most two points that lie on the curve with the same x coordinate. One point will have an odd value of y while the other point will have an even value of y. This allows us to compress a point $P(x, y)$ which yields $C(x, odd/even)$. Only one bit is required to represent whether the y coordinate is even or odd. Since, 256 bits are required for the x coordinate, we need only 257 bits for the public key.

In Ethereum, the format of compressing and representing the public key is as such — the x coordinate of the point P is encoded in hexadecimal and prepended with one byte (02 if y is even and 03 if y is odd).

G. Shamir's Secret Sharing

Before we can take a look at the design of our distributed key generation algorithm over the P2P network, we need to understand Shamir's Secret Sharing. This will be used for storing the private key over the n nodes in such a way that none of the nodes will have knowledge of the value of the private key and only t ($t < n$) nodes will be required to recover the key.

In Shamir's Secret Sharing, a secret is broken up into n parts wherein each part is stored in a participating node. However, in order to recover the original secret, we only require a threshold t number of shares ($t < n$). This threshold is the minimum number of shares required to retrieve the original secret and this is achieved using poynomial interpolation.

1) Generating the shares: In order to uniquely identify a $t-1$ degree polynomial, we need at least t points. This concept forms the basis of polynomial interpolation which is used in Shamir's Secret Sharing.

To split a secret z into n shares with a threshold of t shares, we generate a polynomial of degree $t-1$ having random coefficients. The polynomial f is such that

$$f(0) = secret$$

We then select n random points having x coordinates $X_1, X_2, \dots, X_n$. The y-coordinates of these points are the shares $z_1, z_2, \dots, z_n$ of the secret that will ultimately be distributed between the participants.

2) Lagrange's Interpolation: One popular polynomial interpolation method used in Shamir's Secret Sharing is Lagrange's Interpolation method. Suppose, of the n shares that were generated in the previous step, we have t shares $t_1, t_2, \dots, t_n$ to retrieve the secret z. Lagrange's Interpolation method works as follows:

1) We define t polynomial curves. Each curve $f_t(x)$ is of degree $t-1$ and hence will have $t-1$ roots.
2) The ith curve $f_{t,i}(x)$ will have roots $x_j\ \forall j \in [1,t]\ and\ j \neq i$. Thus we can write $f_{t,i}(x)$ as

$$f_{t,i}(x) = A \prod_{j=1, j\neq i}^{t} (x - x_j)$$

where A is some constant.

3) We also want the value of $f_{t,i}(x_i)$ to be equal to the value of t_i.

$$f_{t,i}(x_i) = t_i$$

4) This can be done by defining $f_{t,i}(x)$ as:

$$f_{t,i}(x) = (\frac{\prod_{j=1,j\neq i}^{t}(x - x_j)}{\prod_{j=1,j\neq i}^{t}(x_i - x_j)})t_i$$

5) Adding all the polynomials $f_{t,i}(x)\ \forall i \in [1,t]$ gives us the original function $f(x)$ that we had used to generate the shares. This is because the t shares uniquely identify the original $t-1$ degree function.
6) Finally, putting the value of 0 in the sum of functions f_t gives us the value of the original secret z.

H. Generating the Private Key over the P2P Network

We know that in Elliptic Key Cryptography, the private key is randomly generated. However, in distributed key generation using ECC, our aim is for each node to contribute to this randomness. In our design, each node generates a random secret z_i that none of the other nodes will ever have knowledge of. The private key will then be the sum of these private shares but it will never be explicitly calculated when generating the public key.

$$k = \sum_{i=1}^{n} z_i$$

I. Generating the Public Key over the P2P Network

As explained in the section on *Elliptic Curves*, the scalar multiplication operation is distributive over addition. Therefore, each node i can calculate a part of the final public key from its own secret as

$$P_i = z_i G$$

Once all the nodes calculate their respective values of P_i, every node can send its P_i to every other node. Each node can then simply add up the list of public key shares $P_j\ \forall j \in [1,n]$ it receives. In this way, every node will know what the public key is.

J. Retrieving the Private Key over the P2P Network

After each node generates its own secret z_i, each node will behave as a dealer and use Shamir's Secret Sharing to split its own secret into n shares $z_{i,j}\ \forall j \in [1,n]$. The same set of x-coordinates $x_1, x_2, \dots, x_n$ will be used by all the nodes for generating the shares.

The shares generated in each node are then distributed to all the other nodes such that at the end of the entire procedure, each node i will have a share $z_{j,i}\ \forall j \in [1, n]$. Each node can then sum these values up to get its own unique brand new share of the private key.

$$k_i = \sum_{j=1}^{n} z_{j,i}$$

To retrieve the final private key, we need a minimum of t (the threshold) shares. This can be done by randomly sampling t shares from the set of n nodes. Once the network decides which t nodes the shares should be procured from, Lagrange's interpolation can be used to retrieve the final private key.

$$f(x) = \sum_{i=1}^{t}((\frac{\prod_{j=1,j\neq i}^{t}(x - x_j)}{\prod_{j=1,j\neq i}^{t}(x_i - x_j)})k_i)$$

The value of the function f at $x = 0$ gives us the private key.

$$f(0) = \sum_{i=1}^{t}((\frac{\prod_{j=1,j\neq i}^{t} x_j}{\prod_{j=1,j\neq i}^{t}(x_j - x_i)})k_i) = k$$

K. Proof of equality of generated and retrieved private keys

In this section, we present a mathematical proof of the equality of the generated private key ($z_{gen} = \sum_{i=1}^{n} z_i$) and the retrieved private key ($z_{ret} = f(0)$).

We have already seen in the earlier sections that in each node, z_i is split up into n shares $z_{i,1}, z_{i,2}, \ldots, z_{i,n}$. z_i can be retrieved from t of these shares using Lagrange's interpolation. As mentioned earlier, the set $x_1, x_2, \ldots, x_n$ used to generate the shares $z_{i,j}$ in each node is the same.

After the distribution of shares between the nodes, each node i will have shares $z_{1,i}, z_{2,i}, \ldots, z_{n,i}$. These shares are summed up in each node i to get

$$k_i = \sum_{j=1}^{n} z_{j,i}$$

Applying Lagrange's interpolation method on t of the k_i shares gives us

$$f(x) = z_{ret} = \sum_{i=1}^{t}((\frac{\prod_{s=1,s\neq i}^{t}(x - x_s)}{\prod_{s=1,s\neq i}^{t}(x_i - x_s)})k_i)$$

$$= \sum_{i=1}^{t}((\frac{\prod_{s=1,s\neq i}^{t}(x - x_s)}{\prod_{s=1,s\neq i}^{t}(x_i - x_s)})(\sum_{j=1}^{n} z_{j,i}))$$

$$= \sum_{j=1}^{n}(\sum_{i=1}^{t}((\frac{\prod_{s=1,s\neq i}^{t}(x - x_s)}{\prod_{s=1,s\neq i}^{t}(x_i - x_s)})z_{j,i}))$$

The inner summation represents the application of Lagrange's interpolation to retrieve the value of k_i from the t values of $z_{i,j}$ for node i. Therefore, this gives

$$f(x) = \sum_{j=1}^{n} k_j = z_{gen}$$

Thus, we have

$$z_{gen} = z_{ret}$$

V. Implementation

A. Twisted

The project is written completely in Python. *Twisted* has been used for implementing the P2P network. *Twisted* is an event-driven network programming library. It abstracts away all of the underlying implementation required for network programming, thereby allowing the programmer to rapidly build and develop powerful networking applications. It comes with implementations of clients and servers for all of its protocols and allows for the easy configuration and deployment of network-driven applications.

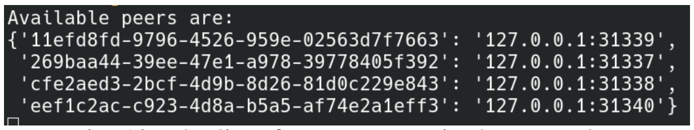
The main constructs of the *Twisted* library used for this project are the *Protocol*, *Factory*, *TCP4ServerEndpoint* and *TCP4ClientEndpoint* classes.

1) Protocol: This class is responsible for handling message communication between all the nodes in the network. Each connection is associated with a protocol and a $Protocol$ instance is instantiated in each of the two end nodes. The protocol instance associated with the connection gets garbage-collected as soon as the connection ends.

2) Factory: This class is responsible for storing data and information important to each node. This is because the factory offers persistent storage of data that might be useful to the instantiated protocols. Each node is associated with one $Factory$ instance. All the protocol instances that are instantiated in a node can access the data stored in a factory.

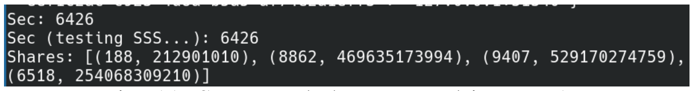
3) TCP4ServerEndpoint: This class is used to create a listening endpoint at the server. It listens for connections and accepts new connections that are initiated by clients.

4) TCP4ClientEndpoint: This class is used to instantiate a client endpoint that can be used to connect to a server endpoint listening for connections.



```
Available peers are:
{'11efd8fd-9796-4526-959e-02563d7f7663': '127.0.0.1:31339',
 '269baa44-39ee-47e1-a978-39778405f392': '127.0.0.1:31337',
 'cfe2aed3-2bcf-4d9b-8d26-81d0c229e843': '127.0.0.1:31338',
 'eef1c2ac-c923-4d8a-b5a5-af74e2a1eff3': '127.0.0.1:31340'}
```

Fig. 10: The list of peers present in the network.



```
Sec: 6426
Sec (testing SSS...): 6426
Shares: [(188, 212901010), (8862, 469635173994), (9407, 529170274759),
(6518, 254068309210)]
```

Fig. 11: Secret and shares stored in peer 1.

B. tinyec

Tinyec is a python library to perform arithmetic operations on elliptic curves. It allows the programmer to construct their own curves by feeding it parameters such as the generator point, the order of the curve and the cofactor. It also has a registry of known named curves including $secp256k1$.

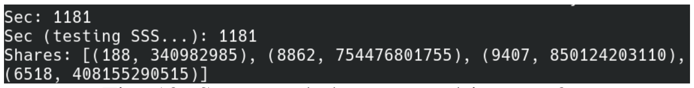
VI. Results

The application is capable of building and maintaining a P2P network. The nodes are able to randomly generate secrets, the sum of which is the private key. However, the private key is neither computed nor stored in a single place. The public key is constructed in a distributed manner and then stored in every node in the network. Shamir's Secret Sharing is used for distributing and storing the private shares amongst the nodes and Lagrange's Interpolation formula is used to retrieve the private key from the shares stored in t nodes. Figures 10 through 16 show the results obtained during a particular test run.

VII. Conclusions

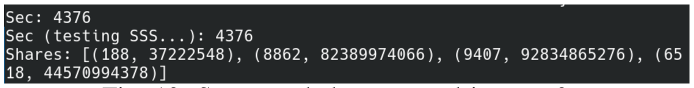
This project demonstrates the implementation of a distributed key generation algorithm using Elliptic Curve Cryptography over a P2P network. The network has a mesh topology which is not suitable for practical purposes when there are a significantly large number of nodes. This is because the number of links in the network increases as the square of the number of nodes present in the network.

Each node in the P2P network acts as a dealer and uses Shamir's Secret Sharing to split its own secret



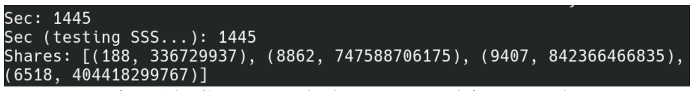
```
Sec: 1181
Sec (testing SSS...): 1181
Shares: [(188, 340982985), (8862, 754476801755), (9407, 850124203110),
(6518, 408155290515)]
```

Fig. 12: Secret and shares stored in peer 2.



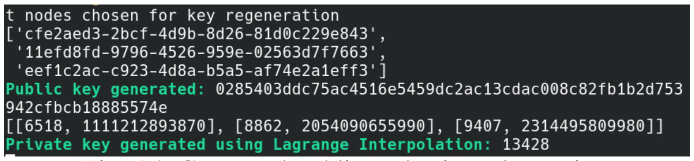
```
Sec: 4376
Sec (testing SSS...): 4376
Shares: [(188, 37222548), (8862, 82389974066), (9407, 92834865276), (65
18, 44570994378)]
```

Fig. 13: Secret and shares stored in peer 3.



```
Sec: 1445
Sec (testing SSS...): 1445
Shares: [(188, 336729937), (8862, 747588706175), (9407, 842366466835),
(6518, 404418299767)]
```

Fig. 14: Secret and shares stored in peer 4.



```
t nodes chosen for key regeneration
['cfe2aed3-2bcf-4d9b-8d26-81d0c229e843',
 '11efd8fd-9796-4526-959e-02563d7f7663',
 'eef1c2ac-c923-4d8a-b5a5-af74e2a1eff3']
Public key generated: 0285403ddc75ac4516e5459dc2ac13cdac008c82fb1b2d753
942cfbcb18885574e
[[6518, 1111212893870], [8862, 2054090655990], [9407, 2314495809980]]
Private key generated using Lagrange Interpolation: 13428
```

Fig. 15: Generated public and private key pair.

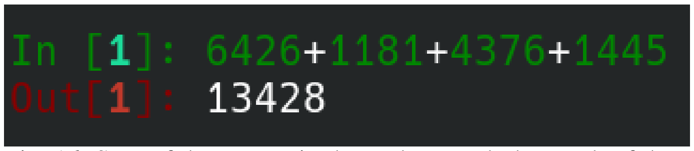
and distribute the shares amongst each other. Each node calculates a part of the public key. Taking advantage of the properties of point addition and scalar multiplication on elliptic curves, these parts are shared with all the nodes and the final public key is reconstructed in each node.

The project also allows for the private key to be retrieved from the shares of at least t nodes using Lagrange's interpolation formula. While the elliptic curve that is used in this project is $secp256k1$ which is also used in bitcoin, the private key that is generated is quite small (less than 32 bits) . This is because of implementation issues in Python with regards to floating point arithmetic involving very large numbers (at least of the order of 10^2 bits)

The plan for the future is to make the application capable of making valid digital signatures and verifying digital signatures using the *Elliptic Curve Digital Signature Algorithm* in a distributed manner wherein the private key will not have to be reconstructed in any single node. A secondary objective is to also figure out a way to handle arithmetic involving much larger floating point numbers. A third objective is to allow for the application to handle situations when new nodes join the network or old nodes leave the network after the computation of the public and private key pair.

References

[1] T. Fadai, S. Schrittwieser, P. Kieseberg, and M. Mulazzani, "Trust me, i'm a root ca! analyzing ssl root cas in modern browsers and operating systems," 08 2015, pp. 174–179.



```
In [1]: 6426+1181+4376+1445
Out[1]: 13428
```

Fig. 16: Sum of the secrets in the nodes match the result of the private key obtained after applying Lagrange's interpolation formula.

[2] H. Adkins. (2011) An update on attempted man-in-the-middle attacks. [Online]. Available: https://security.googleblog.com/2011/08/update-on-attempted-man-in-middle.html

[3] D. Bogdanov, "Foundations and properties of shamir's secret sharing scheme research seminar in cryptography," 01 2007.

[4] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," *Journal of Cryptology*, vol. 20, pp. 51–83, 05 2007.

[5] H. Lee, H. Shen, and B. W. May, "Implementation and discussion of threshold rsa," 2016.

[6] O. Althobaiti and H. Aboalsamh, "An enhanced elliptic curve cryptography for biometric," 01 2012, pp. 1048–1055.

[7] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology — CRYPTO '91*, J. Feigenbaum, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 129–140.

[8] S. K. Langford, "Weakness in some threshold cryptosystems," in *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1109. Springer, 1996, pp. 74–82.

[9] P. Jesus, C. Baquero, and P. S. Almeida, "Id generation in mobile environments," 2006.

[10] H. L. Nguyen, "Rsa threshold cryptography," 2005.

[11] Y. E. Housni, "Introduction to the mathematical foundations of elliptic curve cryptography," 2018.

[12] J. R. Vacca, *Cyber Security and IT Infrastructure Protection*. Waltham, MA : Syngress, 2014.