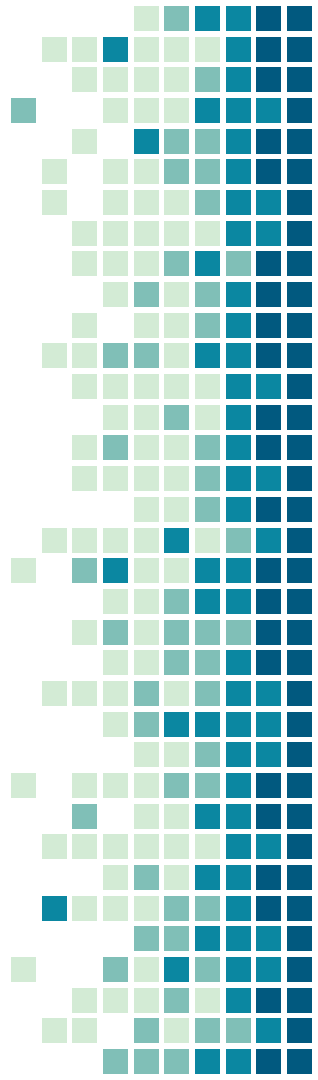


# Improvement of Diffie-Hellman Key Exchange



# Abstract

**This project is focused on the implementation of the Diffie-Hellman algorithm for secure key exchange between two parties. To achieve this, two large prime numbers must be selected, and the steps required for key exchange must be performed. This will involve generating secret keys, computing the shared secret key, and using it for secure communication. Knowledge of modular arithmetic and a programming language with support for large integers will be required. Upon successful implementation, this algorithm will provide a secure method for key exchange and contribute to the development of secure communication technologies and cryptography.**





# Introduction

Diffie-Hellman (DH) is a cryptographic protocol used for secure key exchange between two parties, invented by Whitfield Diffie and Martin Hellman in 1976. It enables two parties to establish a shared secret key over an insecure communication channel without any prior shared secrets. Before DH, symmetric-key cryptography was the dominant approach for securing communication channels, and the most well-known public-key encryption method was the RSA algorithm. Symmetric-key cryptography had the limitation of needing a secure way to distribute the secret key, and public-key cryptography had the limitation of a computational overhead. The Diffie-Hellman protocol solved both of these issues, and is still widely used today.



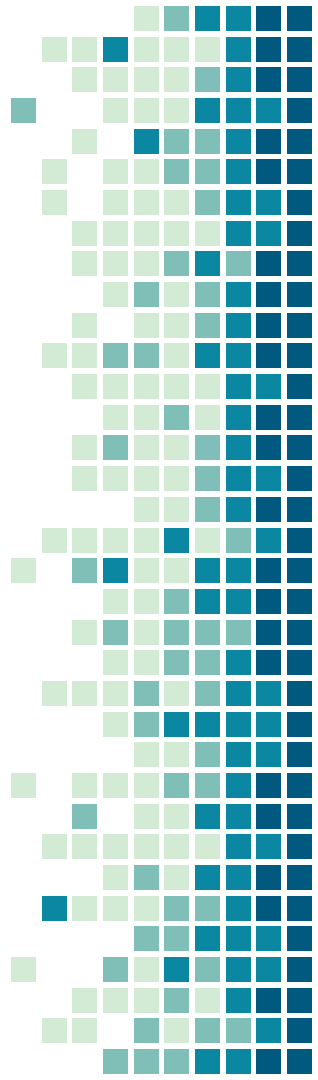
# Diffie-Hellman's Impact

The Diffie-Hellman protocol is an essential tool in modern cryptography, used in various applications such as TLS, VPNs, and messaging apps. It is vulnerable to certain attacks, however, and in 2016 the FBI attempted to compel Apple to provide a backdoor into an iPhone used by one of the San Bernardino shooters. This sparked a debate over the use of encryption and government access to private communication, and the Diffie-Hellman protocol was cited as an example of a secure encryption method that the FBI would not be able to easily break without a backdoor. As a result, there is ongoing research into improving and strengthening the Diffie-Hellman algorithm, leading to the development of more secure variants such as elliptic curve Diffie-Hellman (ECDH) key exchange.



# Problem Statement

**The Diffie-Hellman key exchange is a widely used cryptographic algorithm for secure communication. However, it is vulnerable to several attacks, such as the man-in-the-middle attack, which can compromise the security of the key exchange.**



To improve the Diffie-Hellman key exchange, there is a need to address these vulnerabilities and make it more secure against such attacks. The direction for improving the Diffie-Hellman key exchange is to develop a new variant or modification of the algorithm that can mitigate these vulnerabilities and provide stronger security guarantees, without compromising the efficiency and simplicity of the original algorithm. Additionally, the new variant or modification should be compatible with existing implementations and widely adopted by the cryptographic community.

## “ *Literature Survey*

One of the earliest applications of public key exchange in the context of cryptography is the Diffie-Hellman key exchange algorithm. The Diffie-Hellman algorithm is a cryptographic technique used to create a shared secret key that may be used to encrypt and decode messages between two parties in the future without sending the key via an insecure channel. With the diffie hellman algorithm, two parties can construct a shared secret key over an insecure channel without communicating the key itself, thereby protecting against sniffing attacks.

## 1: Prevention of the Man-in-the-Middle Attack on Diffie-Hellman Key Exchange Algorithm:

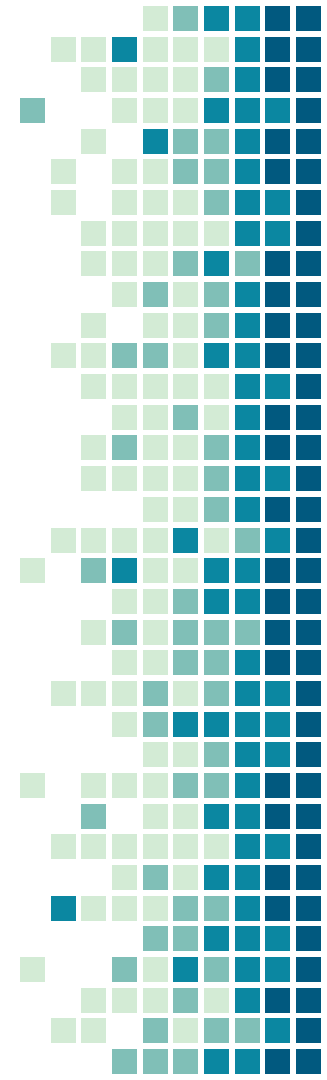
A Review The Man-In-The-Middle Attack is a fairly common issue with the Diffie-Hellman key exchange technique when keys are transmitted across a channel. Before calculating the secret and shared key, three tests were conducted to verify the created sequences and to determine the needed minimum length for which all the tests are satisfied. The Geffe generator is used to generate a binary sequence with a high degree of arbitrariness. The method here assures us that no such keys will be stored as hashes on the server and transmitted across the channel. This method performs better than the current algorithms and prevents the Man-In-The-Middle attack.

**Authors -Samrat Mitra, Samanwita Das, and Malay Kul**

## 2: Provably Secure Authenticated Group Diffie-Hellman Key Exchange - 2007

Dynamic group Diffie-Hellman protocols for Authenticated Key Exchange (AKE) are made to function in a situation where the membership of the multicast group is not known in advance and parties can join and depart at any time. Despite the fact that a number of schemes have been put up to address this situation, no formal solution to this cryptographic quandary has ever been put forth. The essential goal of Authenticated Key Exchange (AKE) in this work is "implicit" authentication, along with the entity-authentication goal.

**Authors - Emmanuel Bresson, Olivier Chevassut, and David Pointcheva**





### 3:Private DH: An Enhanced Diffie-Hellman Key-Exchange Protocol using RSA and AES Algorithms 2021

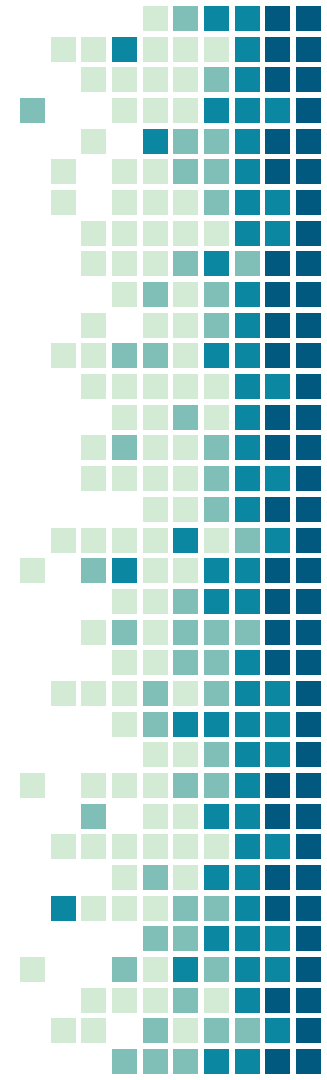
Diffie-Hellman is the most well-known key exchange protocol, although it is vulnerable to number discrete log algorithm attacks. Furthermore, new studies indicate that Diffie-Hellman may not be as secure as many people think. Another problem with the Diffie-Hellman algorithm is the logjam attack, which permits man-in-the-middle attacks. Integration of RSA, AES, and the Diffie-Hellman algorithm to prevent potential attacks on the private Diffie-Hellman key exchange protocol in order to overcome the difficulties such as the logjam attack. The main goal is to guarantee the Diffie-Hellman Algorithm's security.

**Author-Ripon Patgiri, Senior Member, IEEE**

### 4:A study on diffie-hellman key exchange protocols May 2017 :

The Diffie-Hellman protocol's goal is to make it possible for two parties to safely exchange a session key that can later be applied to symmetric message encryption. Diffie-Hellman does not, however, authenticate the entities that are communicating on their own. We examine the Diffie-Hellman Key exchange protocol in this essay. Whitfield Diffie et al. proposed the station-to-station (STS) protocol in 1992 as an effective authenticated key exchange protocol. The legitimate participants in a two-party authenticated key exchange can compute a secret key using Diffie-Hellman and then authenticate each other by exchanging digital signatures. Station to Station should therefore be protected against a man-in-the-middle attack.

**Authors-Manoj Ranjan Mishra and Jayaprakash Kar**



## 5:Enhanced diffie-hellman algorithm for reliable key exchange 2017

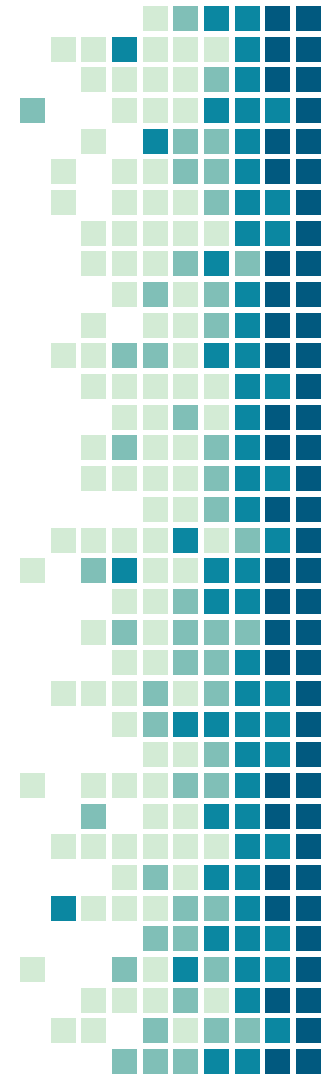
The Diffie-Hellman is one of the first public-key procedure and is a certain way of exchanging the cryptographic keys securely. This concept was introduced by Ralph Markel and it is named after Whitfield Diffie and Martin Hellman. In Public key cryptosystem, the sender has to trust while receiving the public key of the receiver and vice-versa. Man-in-the-middle attack is very much possible on the existing Diffie –Hellman algorithm. To reduce the possibility of attacks on Diffie-Hellman algorithm, we have enhanced the Diffie- Hellman algorithm to a next level. The second secret key will be generated by taking primitive root of the first secret key.

**Authors -Aryan, Chaithanya Kumar and Durai Raj Vincent P M**

## 6:The Elliptic Curve Diffie-Hellman (ECDH) 2015

A Diffie Hellman variant called the Elliptic Curve Diffie-Hellman (ECDH) enables two parties with no prior knowledge of one another to create a shared secret key via an insecure channel.ECDH has many advantages over the standard Diffie-Hellman such as a faster computation time,better scalability,an increase in efficiency.

**Author -Joanna Lang**



## 7: Integrating Diffie-Hellman Key Exchange into the Digital Signature Algorithm (DSA) 2004

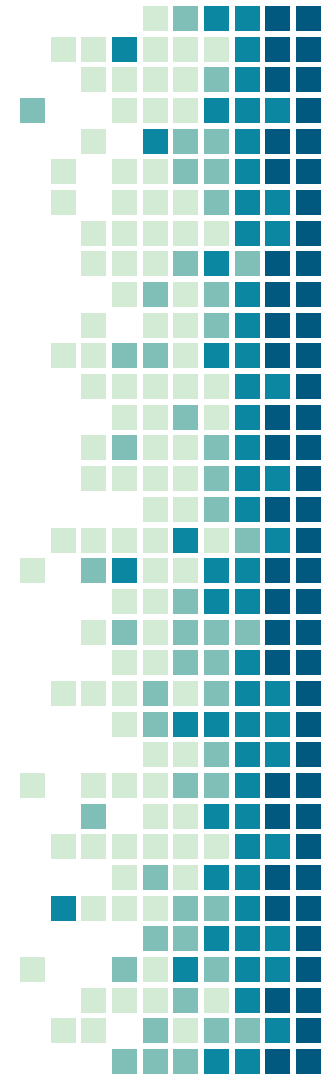
Under the Federal Information Processing Standard, the National Institute of Standards and Technology (NIST) has established a number of security standards (FIPS). Key agreement standards are still missing from the present standards. Arazi suggested including the Digital Signature Algorithm's (DSA) Diffie-Hellman (DH) key exchange (DSA). Nyberg et al., however, criticized the protocol. In order to securely integrate DH key exchange with DSA for authenticated key distribution, we provide three possible techniques.

**Authors -Lein Harn, Manish Mehta, Student Member, IEEE, and Wen-Jung Hsin**

## 8.A Conference Key Scheme Based on the Diffie-Hellman Key Exchange 2018

Secure group communication on the internet is becoming more and more crucial. Every group member must be able to create a shared secret key in order to ensure secure and dependable communication among participants over a public network. This particular public key distribution mechanism is what we refer to as a conference key distribution system (CKDS). Our protocol builds intermediate keys from each subgroup gradually until the whole conference key is achieved. It is based on the two-party Diffie-Hellman protocol. This technique helps a conference key distribution system operate more effectively.

**Authors- Li-Chin Huang and Min-Shiang Hwang**



## 9.Improvement of Diffie-Hellman Key Exchange Algorithm 2015

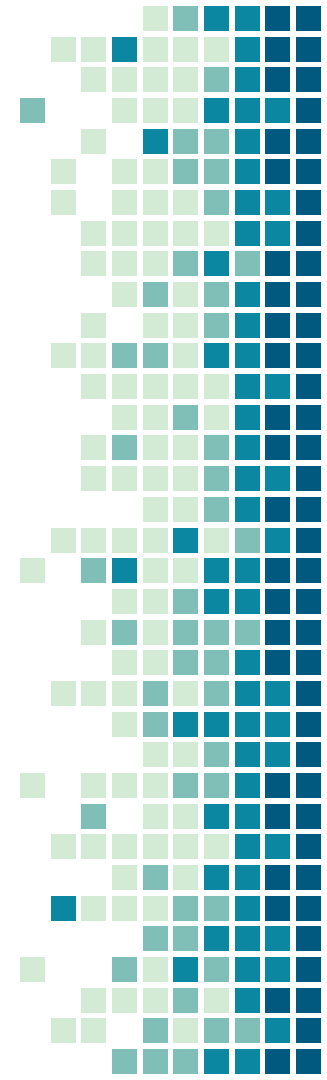
The purpose of this paper is to propose an algorithm which is an improvement over the Diffie-Hellman key exchange. The algorithm is based on using arithmetic and logarithmic calculations for transmission of the shared session keys which enable users to securely exchange keys which further can be used for later encryptions. The proposed algorithm has similar grounds with the Diffie-Hellman algorithm, and a new technique is used for sharing session keys which overcome the time complexity limitation of the Diffie-Hellman algorithm.

**Author-Gurshid Saini**

## 10. Research on Diffie-Hellman Key Exchange Protocol

The Diffie-Hellman protocol's goal is to make it safe for two users to exchange a secret key that will later be used to encrypt messages. The protocol itself is restricted to the key exchange. However, the Diffie-Hellman protocol is frequently vulnerable to man-in-the-middle and impersonation attacks because it lacks an entity authentication method. In this study, we evaluate the computational effectiveness of several authentication strategies. Finally, a hash function-based enhanced key exchange schema is presented, enhancing both the security and usability of the Diffie-Hellman protocol.

**Author -Nan Li**



## 11. Use of Digital Signature with Diffie Hellman Key Exchange and AES Encryption Algorithm to Enhance Data Security in Cloud Computing

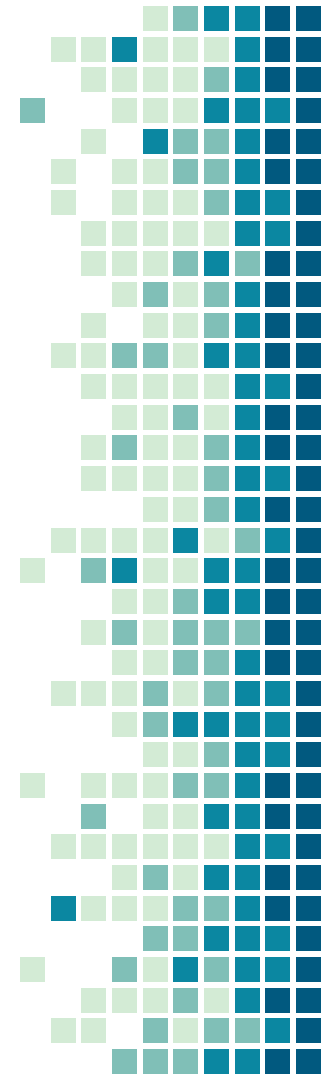
Using a combination of the Advanced Encryption Standard (AES) encryption algorithm, Diffie Hellman key exchange, and digital signature we can ensure the safety and privacy of data stored in the cloud. Even if the key in transmission is compromised, the Diffie-Hellman key exchange facility renders it useless because the key in transit is useless without the user's private key, which is only accessible by authorised users. The three-way technique makes it difficult for hackers to compromise the security system, securing cloud-based data.

**Author: Mr. Prashant Rewagad and Ms. Yogita Pawar**

## 12. Signed (Group) Diffie-Hellman Key Exchange with Tight Security

The paper proposes the first tight security proof for the ordinary two-message signed Diffie-Hellman key exchange protocol in the random oracle model. The proof is based on the strong computational Diffie-Hellman assumption and the multiuser security of a digital signature scheme. With the security proof, the signed DH protocol can be deployed with optimal parameters, independent of the number of users or sessions, without need to compensate any security loss

**Authors - Jiaxin Pan, Chen Qian and Magnus Ringerud**



### 13. Security Issues in the Diffie-Hellman Key Agreement Protocol

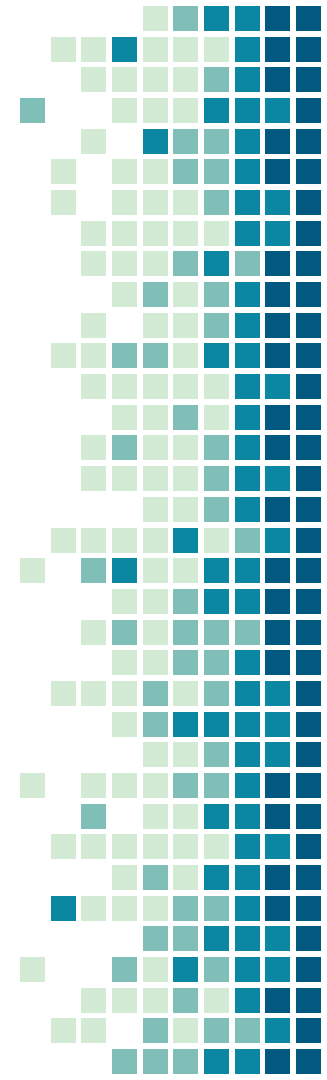
Implementations of the Diffie-Hellman key agreement mechanism have numerous critical security weaknesses. The attacks are frequently overlooked by protocol designers because they might be rather subtle. A reasonably thorough pseudo-code for the authenticated Diffie-Hellman protocol and for the half-certified Diffie-Hellman in addition to exposing the most significant attacks and concerns (a.k.a. Elgamal key agreement) is also provided.

**Authors: Jean-Francois Raymond and Anton Stiglic.**

### 14. A new approach of elliptic curve Diffie-Hellman key exchange

Elliptic Curve Cryptosystem (ECC) schemes are public-key mechanisms that provide encryption, digital signature and key exchange capabilities. The advantage of elliptic curves is that they ensure a level of security equivalent to that of existing public key systems but with shorter key lengths. In this paper, an interest is taken in public-key exchange of Diffie-Hellman. The paper proposes a new approach of elliptic curve Diffie-Hellman key exchange.

**Author- Nissa Mehibel, M'hamed Hamadouche**



### **15. Elliptic Curve Diffie-Hellman Key Exchange Algorithm for Securing Hypertext Information on Wide Area Network**

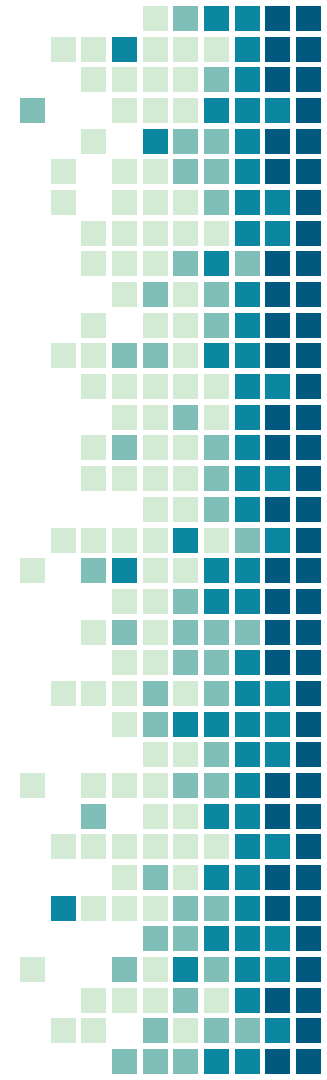
Elliptic curve cryptography (ECC) is an emerging favorite because it requires less computational power, communication bandwidth, and memory when compared to other cryptosystems. The paper presents Elliptic curve cryptography and Diffie–Hellman key agreement protocol, itself is an anonymous (non-authenticated) key-agreement protocol, it provides the basis for a variety of authenticated protocols, and is used to provide forward secrecy for web browsers application using HTTPS.

**Author-Ram Ratan Ahirwal, Manoj Ahke**

### **16. Secure Authentication Scheme Using Diffie-Hellman Key Agreement for Smart IoT Irrigation Systems**

A secure authentication scheme using Diffie-Hellman key agreement for smart IoT irrigation systems is proposed. This article discusses a secure authentication scheme using the Diffie-Hellman key agreement for smart IoT irrigation systems. A performance analysis demonstrated that the proposed scheme has a minimal cost in terms of storage size and also has a suitable level of intercommunication and running time costs, compared with other recently proposed authentication schemes. Finally, the proposed scheme is applicable for use in smart IoT irrigation systems in order to remotely monitor the actual amount of water required to irrigate the crops

**Author-Shadi Nashwan**



### 17. Mutual query data sharing protocol for public key encryption through chosen-ciphertext attack in cloud environment

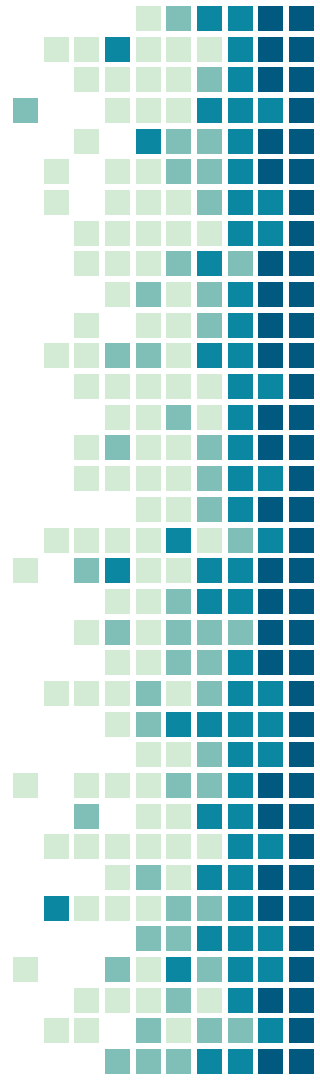
The proposed data sharing protocol is designed to resist the chosen-ciphertext attack (CCA) under the hardness solution for the query shared-strong diffie-hellman (SDH) problem. Encryption or decryption time limitations of exiting protocols like Boneh, rivest shamir adleman (RSA), Multi bit transposed ring learning parity with noise (TRLPN), ring learning parity with noise (Ring LPN) cryptosystem, key Ordered decisional learning parity with noise (kO DLPN), and KD\_CS protocol's. The evaluation of proposed work with the existing data sharing protocols in computational and communication overhead through their response time is evaluated. This paper proposes the Mutual Query Data Sharing Protocol (MQDS) to provide security for authenticated user data among distributed physical users and devices. The proposed data sharing protocol is designed to resist the chosen-ciphertext attack (CCA).

**Authors - Tarasvi Lakum and Barige Thirumala Rao.**

### 18. Authenticated Key Exchange Secure under the Computational Diffie-Hellman Assumption

The paper proposes a new authenticated key exchange (AKE) protocol and proves its security under the random oracle assumption and the computational Diffie-Hellman (CDH) assumption.

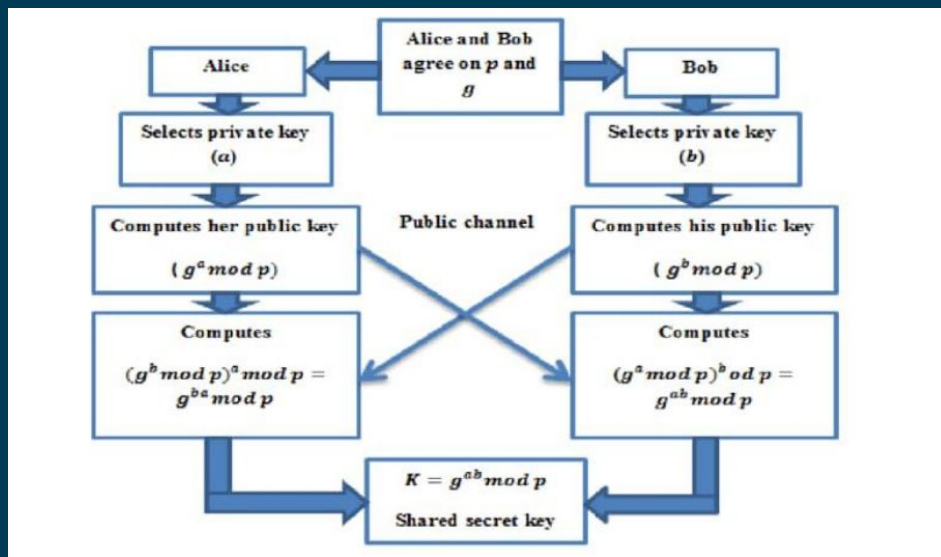
**Authors - Jooyoung Lee and Je Hong Park**





# Existing Methods

The Diffie-Hellman algorithm uses mathematical concepts such as modular arithmetic and discrete logarithm for making a common key for both sender and receiver using the communication channel where sender and receiver choose a common prime number  $p$  and  $g$  as it's primitive root



# Steps in Performing Diffie Hellman Algorithm for Secure Key Exchange

1. Sender and Receiver agree on a prime number  $p$ ,  $q$  as its primitive root.
2. Sender and Receiver choose private keys  $x$  and  $y$  respectively which are known to themselves only .
3. Sender generates Public\_key\_1 using the formula :  $((q^x) \bmod p)$
4. Receiver generates Public\_key\_2 using the formula :  $((q^y) \bmod p)$ .
5. Sender and Receiver exchange their respective public key. Now Sender has Public\_key\_1 and Receiver has Public\_key\_2.
6. Sender calculates Secret Key using the formula :  $((\text{Public\_key\_2} ^ x) \bmod p)$
7. Receiver calculates Secret Key using the formula :  $((\text{Public\_key\_1} ^ y) \bmod p)$
8. The Sender and Receiver both generate the same Secret key

```
import time

start = time.time()
P= 487
q = 3

print('The Value of Prime number P is :%d'%(P))
print('The Value of primitive root q is :%d'%(q))

# Sender will choose the private key "x"
x = 17
print('The Private Key for Sender is :%d'%(x))

#public key generated by sender
Public_key_1= int(pow(q,x,P))

print('The Public Key generated by the Sender is :%d'%(Public_key_1))

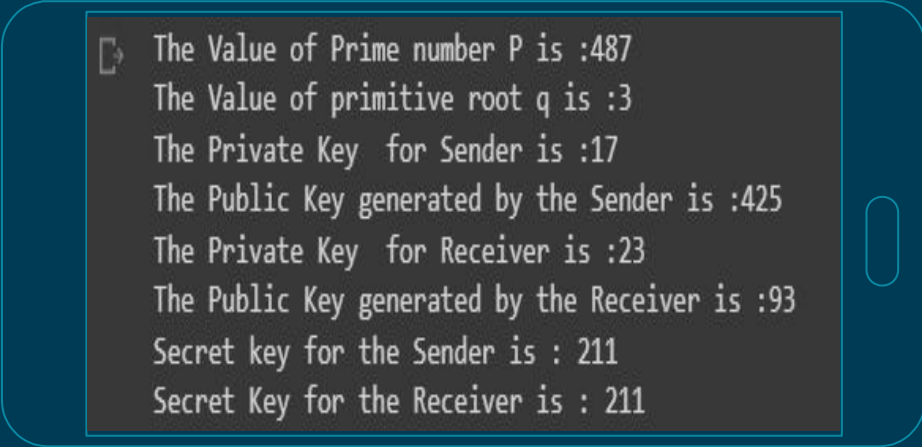
# Receiver will choose the private key "y"
y= 23
print('The Private Key for Receiver is :%d'%(y))

#public key generated by receiver
Public_key_2 = int(pow(q,y,P))
print('The Public Key generated by the Receiver is :%d'%(Public_key_2))

# Secret key for Sender
Secret_key_1 = int(pow(Public_key_2,x,P))

# Secret key for Receiver
Secret_key_2 = int(pow(Public_key_1,y,P))

print('Secret key for the Sender is : %d'%(Secret_key_1))
print('Secret Key for the Receiver is : %d'%(Secret_key_2))
end = time.time()
print(end - start)
```

A graphic of a smartphone with a dark screen. The screen displays a list of cryptographic parameters in a monospaced font. The parameters are: The Value of Prime number P is :487, The Value of primitive root q is :3, The Private Key for Sender is :17, The Public Key generated by the Sender is :425, The Private Key for Receiver is :23, The Public Key generated by the Receiver is :93, Secret key for the Sender is : 211, and Secret Key for the Receiver is : 211. The phone has a light blue outline and a small oval home button on the right side.

The Value of Prime number P is :487  
The Value of primitive root q is :3  
The Private Key for Sender is :17  
The Public Key generated by the Sender is :425  
The Private Key for Receiver is :23  
The Public Key generated by the Receiver is :93  
Secret key for the Sender is : 211  
Secret Key for the Receiver is : 211

A secret number (a private key) and a publicly known number called a base or generator. The public numbers are usually referred to as primitive roots. Here the primitive root is 3 and Prime number is 487. We can see that both sender and receiver share the same secret key and it can be inferred that a secure communication channel can be established between the two users.

# Disadvantages of Diffie-Hellman Key Exchange

Although the Diffie-Hellman key exchange algorithm has several advantages, there are also some disadvantages associated with its use:

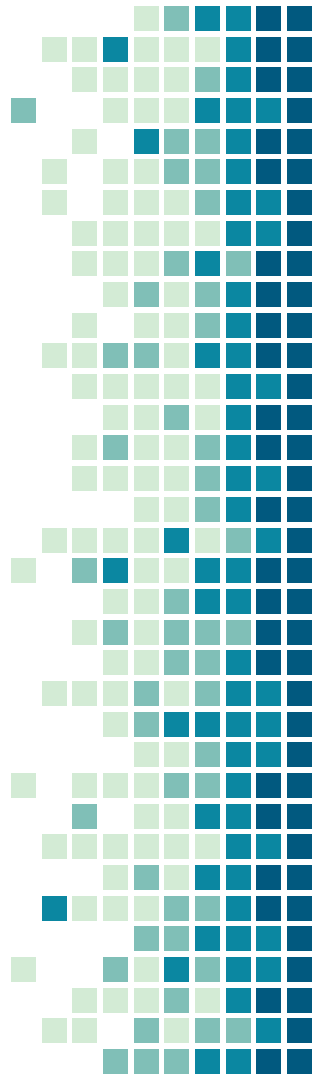
1. Man-in-the-middle attacks: The Diffie-Hellman algorithm does not provide authentication of the communicating parties. This means that an attacker could potentially intercept the communication and impersonate one of the parties, leading to a man-in-the-middle attack.
2. Key exchange is vulnerable to eavesdropping: Although the exchanged keys are secure, the key exchange itself is not encrypted, which means that an attacker could potentially intercept the exchange and determine the shared secret key.
3. Implementation vulnerabilities: The security of the Diffie-Hellman algorithm relies heavily on its implementation. If there are implementation vulnerabilities, an attacker could potentially exploit them to compromise the security of the algorithm.
4. Key management: Diffie-Hellman requires secure key management, which can be challenging in some environments. The keys need to be generated and managed securely to ensure their confidentiality and integrity.
5. Key reuse: If the same key is used for multiple sessions, the security of the system could be compromised if an attacker manages to determine the key.

# Methods to Improve Diffie-Hellman Key Exchange

1. Use larger prime numbers: Using larger prime numbers can increase the security of the Diffie-Hellman algorithm by making it more difficult for an attacker to compute the shared secret key. However, larger prime numbers can also increase the computational complexity of the algorithm.
2. Use elliptic curve cryptography: Elliptic curve cryptography (ECC) is a variant of the Diffie-Hellman algorithm that uses elliptic curves instead of prime numbers. ECC provides stronger security with smaller key sizes, making it a more efficient alternative to traditional Diffie-Hellman.
3. Use perfect forward secrecy: Perfect forward secrecy (PFS) is a property of cryptographic protocols that ensures that even if an attacker gains access to one set of keys, they will not be able to use them to decrypt past communications. PFS can be achieved in the Diffie-Hellman algorithm by generating a new set of keys for each session.
4. Use authenticated key exchange: Authenticated key exchange (AKE) is a variant of the Diffie-Hellman algorithm that includes authentication of the communicating parties to prevent man-in-the-middle attacks. AKE can be achieved through the use of digital signatures or other authentication mechanisms.
5. Use post-quantum cryptography: Post-quantum cryptography (PQC) is a type of cryptography that is resistant to attacks by quantum computers. PQC algorithms, such as the McEliece cryptosystem and the NTRU Encrypt system, can be used as alternatives to the Diffie-Hellman algorithm.

# Proposed Model

Using the concept of Elliptic Curve Cryptography we propose a variant of Diffie Hellman which is more secure and difficult to be compromised by attackers.



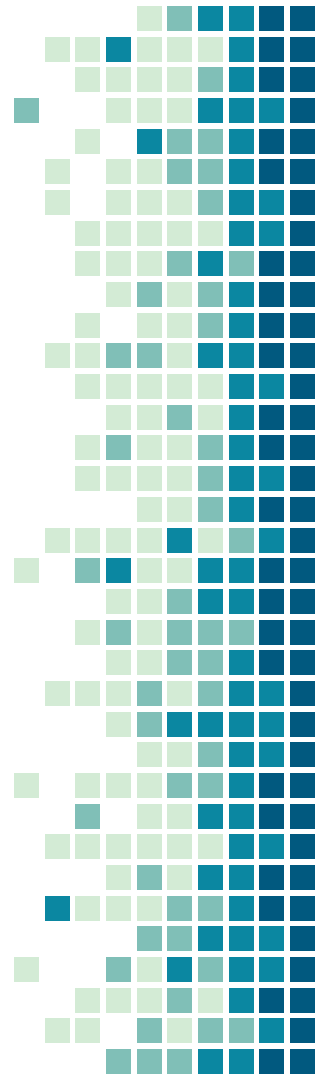
# Elliptic Curve Cryptography

Elliptical curve cryptography (ECC) is a public key encryption technique based on elliptic curve theory that can be used to create faster, smaller and more efficient cryptographic keys.

Public key cryptography systems, like ECC, use a mathematical process to merge two distinct keys and then use the output to encrypt and decrypt data.

One is a public key that is known to anyone, and the other is a private key that is only known by the sender and receiver of the data.

ECC generates keys through the properties of an elliptic curve equation instead of the traditional method of generation as the product of large prime numbers.

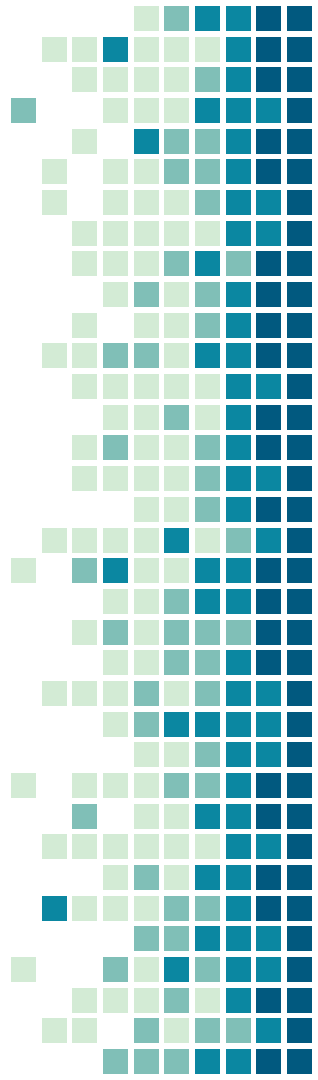




# Elliptic Encryption Cryptography

ECC is like most other public key encryption methods, such as the RSA algorithm and Diffie-Hellman. Each of these cryptography mechanisms uses the concept of a one-way, or trapdoor, function.

This means that a mathematical equation with a public and private key can be used to easily get from point A to point B. But, without knowing the private key and depending on the key size used, getting from B to A is difficult, if not impossible, to achieve.



# Private and Public Keys in ECC

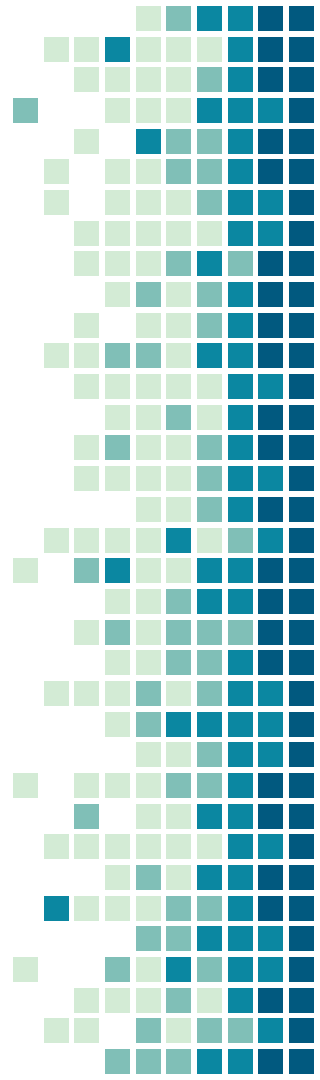
The private keys in the ECC are integers (in the range of the curve's field size, typically 256-bit integers). Example of 256-bit ECC private key (hex encoded, 32 bytes, 64 hex digits) is:

0x51897b64e85c3f714bba707e867914295a1377a7463a9dae8ea6a8b914246319.

The public keys in the ECC are EC points - pairs of integer coordinates  $\{x, y\}$ , laying on the curve. Due to their special properties, EC points can be compressed to just one coordinate + 1 bit (odd or even). Thus the compressed public key, corresponding to a 256-bit ECC private key, is a 257-bit integer

Example of an ECC public key in hex encoded format which comprises of 65 hex digits:

0x6c1c3fb01a3221e6ab2d2fd6e883ffb440d891921a59acf3763578c2f43866c40



# Elliptic Curves

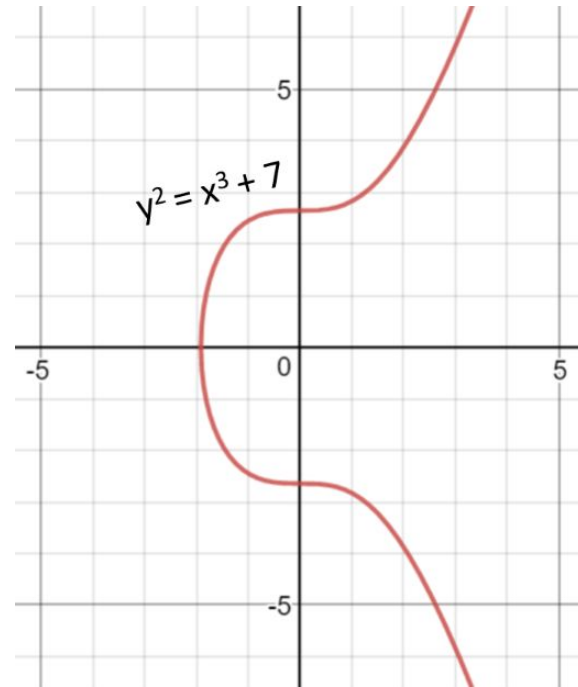
In mathematics elliptic curves are plane algebraic curves, consisting of all points  $\{x, y\}$ , described by the equation

Cryptography uses elliptic curves in a simplified form which is defined as:

$$y^2 = x^3 + ax + b \text{ (General Formula)}$$

Eg:

$$y^2 = x^3 + 7 \text{ (a is considered to be zero in this example)}$$



# Elliptic Curves over Finite Fields

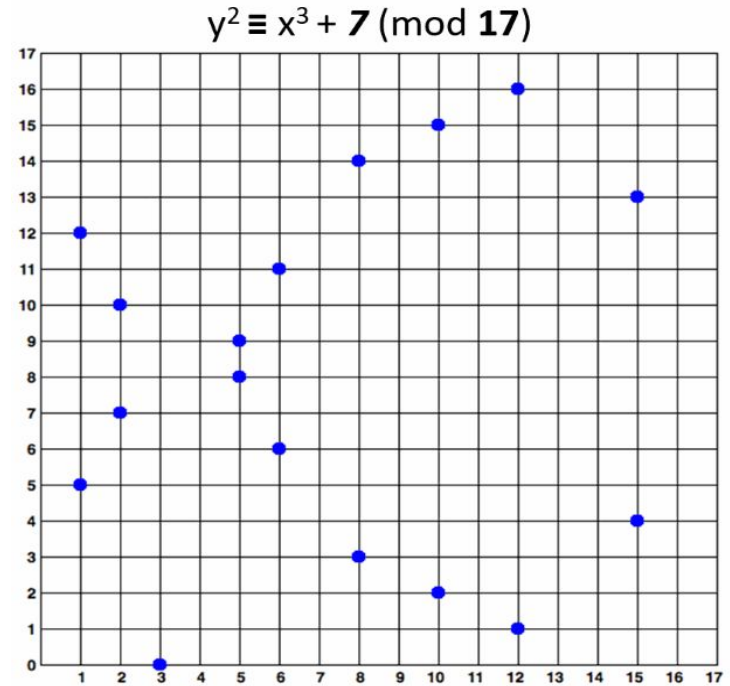
The elliptic curve cryptography (ECC) uses elliptic curves over the finite field  $F_p$  (where  $p$  is prime and  $p > 3$ )

The general form for the equation is given as:

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

$$\text{Eg : } y^2 \equiv x^3 + 7 \pmod{17}$$

The above example is only for visualizing purpose. It provides a very small key length such as 5 bits. In real world developers typically use curves of 256 bits or more



# Private Key, Public Key and the Generator Point in ECC

In the ECC, we multiply a fixed EC point  $G$  ( generator point) by a certain integer  $k$  ( private key), we obtain an EC point  $P$  (its corresponding public key).

$G \rightarrow$  Generator Point

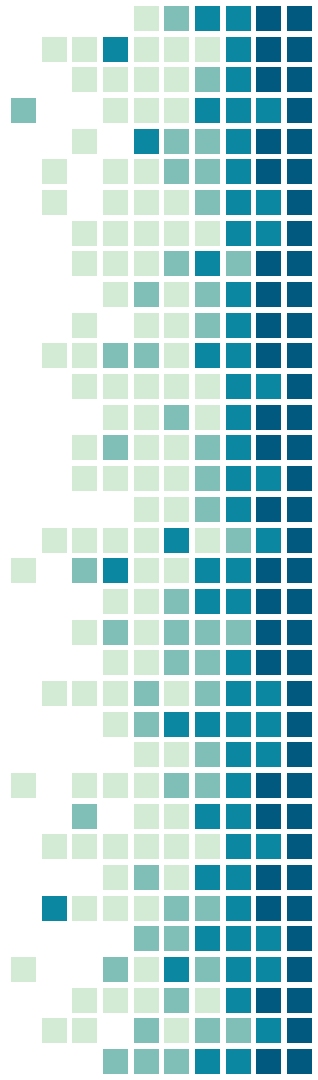
$k \rightarrow$  Private Key(integer)

$P \rightarrow$  Public key (point )

It is very fast to calculate  $P = k * G$ , using the well-known ECC multiplication algorithms in time  $\log_2(n)$ .

It is extremely slow (considered almost impossible for large  $k$ ) to calculate  $k = P / G$ .

This asymmetry (fast multiplication and infeasible slow opposite operation) is the basis of the security strength behind the ECC.



# BrainpoolP256r1

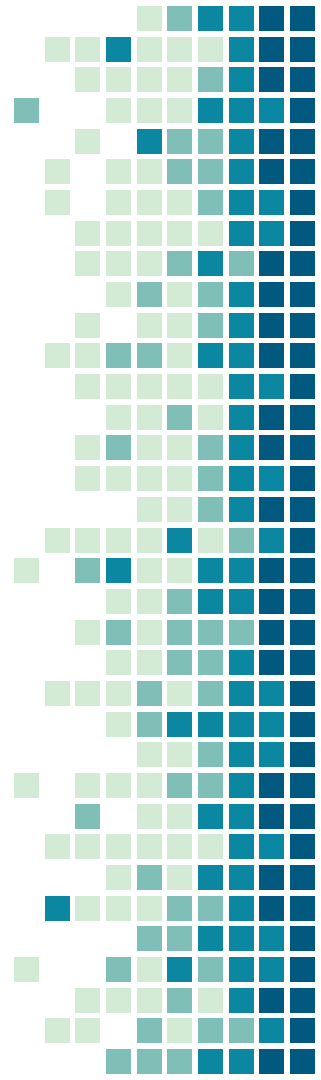
The BrainpoolP256r1 elliptic curve is a member of the Brainpool family of elliptic curves, which were developed for use in cryptographic applications.

The "P256r1" part of the "BrainpoolP256r1" name indicates:

- P Field type = Prime field
- 256 Key size = 256
- r Curve type = Regular curve
- 1 Cofactor = 1

The BrainpoolP256r1 curve is widely used in various cryptographic protocols, including Transport Layer Security (TLS), Internet Key Exchange (IKEv2), and Secure Shell (SSH).

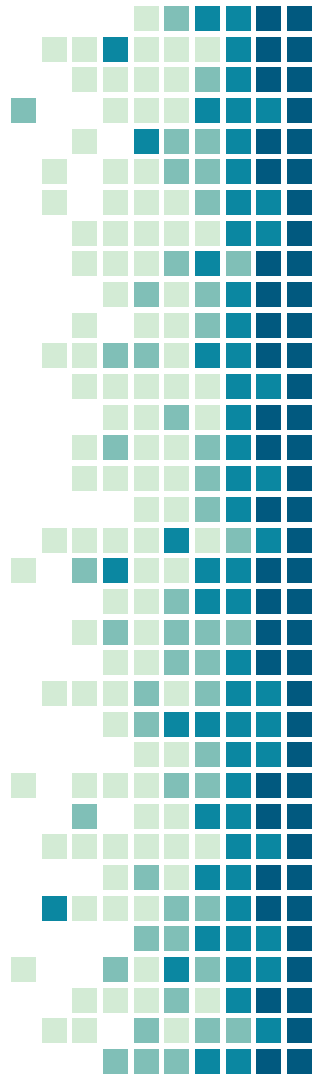
The BrainpoolP256r1 curve is considered a secure and reliable choice for information exchange.



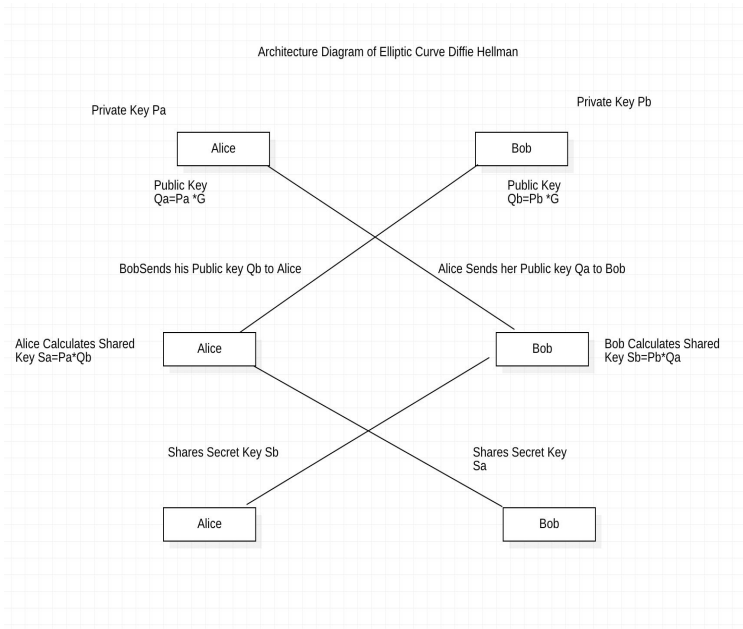
# Importing Modules

tinyec-A tiny library to perform arithmetic operations on elliptic curves in pure python.

secrets -The secrets module is used for generating cryptographically strong random numbers suitable for managing data such as passwords, account authentication, security tokens, and related secrets.



# Architecture Diagram of ECDH



This diagram represents the process of Elliptic Curve Diffie Hellman and the sharing of keys between the two respective users. Using Each other's respective private key and the other person's public key .

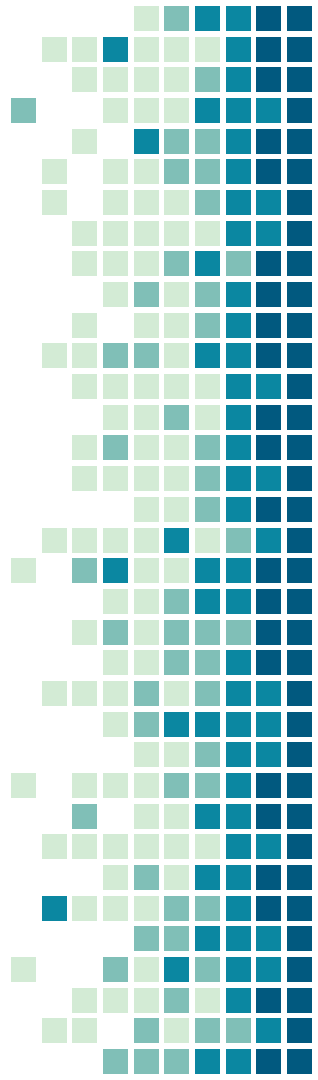
A shared key is calculated by both the users. If the obtained shared keys are equal it can be inferred that a secure communication channel has been established and the two users can share cryptographic keys securely.



# Elliptic Curve Diffie Hellman

The ECDH (Elliptic Curve Diffie–Hellman Key Exchange) is an anonymous key agreement scheme, which allows two parties, each having an elliptic-curve public–private key pair, to establish a shared secret over an insecure channel.

ECDH is very similar to the classical DHKE (Diffie–Hellman Key Exchange) algorithm, but it uses ECC point multiplication instead of modular exponentiations. ECDH is based on the following property of EC points which makes it more secure compared to regular Diffie–Hellman



# Working of ECDH

- Initially we create a function which will compress the value from the format  $y^2 \equiv x^3 + ax + b \pmod{p}$  to a hexadecimal format of 65 hex digits
- We select the desired curve using `registry.get_curve`. In our example the curve is 'brainpoolP256r1'
- Using `secrets.randbelow` function we find the private key for both the parties involved which will be in integer form.
- Then we find public key for both the parties involved by performing ECC point multiplication between the private key and generator point.
- The resultant public key will be in the form of  $y^2 \equiv x^3 + ax + b \pmod{p}$  so we will compress this public key using the compress function and we will be the public key values in hexadecimal format of 65 hex digits



# Working of ECDH

- The two parties exchange their respective Public keys to each other. Then the shared key is calculated by performing ECC point multiplication between one person's private key and another person's public key and vice versa.
- The resultant shared key will be in the form of  $y^2 \equiv x^3 + ax + b \pmod{p}$  so we will compress this public key using the compress function and we will be the public key values in hexadecimal format of 65 hex digits
- We would have two shared keys which will both be the same and this key can be used to later perform encryption and decryption.

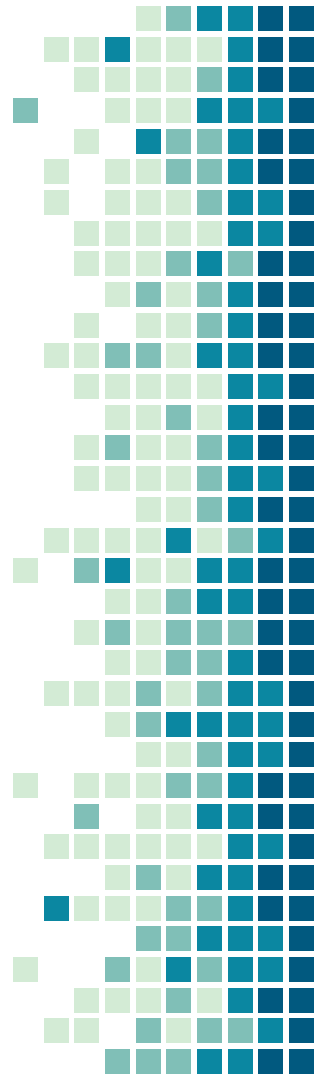


# Advantage of ECDH over diffie hellman

Both Elliptic Curve Diffie-Hellman (ECDH) and traditional Diffie-Hellman (DH) key exchange algorithms are used to establish a shared secret key between two parties over an insecure communication channel. However, ECDH offers some advantages over DH:

1. **Shorter key lengths:** ECDH uses shorter key lengths compared to traditional DH for the same level of security. This makes ECDH more efficient in terms of computation and storage requirements.
2. **Increased security:** ECDH is more resistant to attacks than DH, as it is based on the elliptic curve discrete logarithm problem, which is believed to be harder to solve than the traditional discrete logarithm problem used in DH.
3. **Faster key exchange:** ECDH provides faster key exchange compared to DH due to its shorter key lengths and efficient algorithms.
4. **Support for low-power devices:** ECDH is more suitable for low-power devices such as mobile phones, IoT devices, and embedded systems due to its lower computational requirements.

Overall, ECDH is a more efficient and secure key exchange algorithm compared to DH, and it is widely used in modern cryptographic protocols.



# Implementation

```
from tinyec import registry
import secrets
import time

def compress(pubKey):
    return hex(pubKey.x) + hex(pubKey.y % 2)[2:]

|

curve = registry.get_curve('brainpoolP256r1')

alicePrivKey = secrets.randbelow(curve.field.n)
print("Alice priv key in Integer form: " ,alicePrivKey)
alicePubKey = alicePrivKey * curve.g
print("\nAlice public key in hex encoded format:", compress(alicePubKey))

bobPrivKey = secrets.randbelow(curve.field.n)
print("\nBob priv key in Integer form : " ,bobPrivKey)
bobPubKey = bobPrivKey * curve.g
print("\nBob public key hex encoded format :", compress(bobPubKey))

print("\nExchanging the public keys")

aliceSharedKey = alicePrivKey * bobPubKey
print("\nAlice shared key hex encoded format :", compress(aliceSharedKey))

bobSharedKey = bobPrivKey * alicePubKey
print("\nBob shared key hex encoded format :", compress(bobSharedKey))

if(aliceSharedKey == bobSharedKey):
    print("Alice and Bob can communicate securely")
```



# Output:

```
Yuthishkumars-MacBook-Air:~ yuthishkumar$ /usr/local/bin/python3 /Users/yuthishkumar/Downloads/ecc.py
Alice priv key in Integer form: 42127454369299922530898835669497133389259759560584006766955501618650464905889

Alice public key in hex encoded format: 0xa82ab7d81cf968a24100f00511bb877bc0378b98b13ecfed92d9c049385073781

Bob priv key in Integer form : 34593893265841437955564906911849816109301705177486593635086296801761781268326

Bob public key hex encoded format : 0x288608ce7b6b4269eae1e1bb72976b8a38d3491e7f2c428f6f2a6e309c7a5130

Exchanging the public keys

Alice shared key hex encoded format : 0x173b8f4e43926784343867c1b8352bc0b54994745f39e64bd67c67a289d371230

Bob shared key hex encoded format : 0x173b8f4e43926784343867c1b8352bc0b54994745f39e64bd67c67a289d371230
Alice and Bob can communicate securely
Yuthishkumars-MacBook-Air:~ yuthishkumar$
```

Since the shared key which is derived from one person's public key and the other person's private key is the same. It can be concluded that the parties concerned have established a secure communication channel and can communicate securely.



# Application of ECDH in other encryption techniques.

To prove that the key found using Elliptic Curve Diffie Hellman is functional. We would use the same key to encrypt and decrypt messages between sender and receiver.

We are using the AES Encryption algorithm to implement encryption and decryption between the sender and receiver using the key which we found using ECDH.

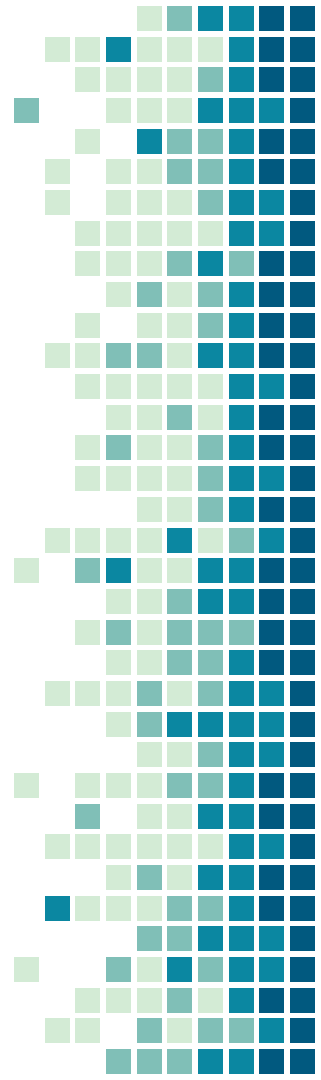


# AES [Advanced Encryption Standard]

AES (Advanced Encryption Standard) is a symmetric encryption algorithm that is widely used for securing sensitive data. It is considered one of the most secure encryption algorithms and is used in a variety of applications such as securing communication channels, securing data at rest, and securing data in transit.

The primary reason why AES is used is its security. AES has been designed to be secure against all known attacks, and its security has been extensively reviewed by the cryptographic community. AES uses a block cipher to encrypt data, where the block size can be 128, 192, or 256 bits. The key size is also configurable and can be 128, 192, or 256 bits.

In terms of how AES is used, the encryption and decryption process involves the use of a secret key, which is used to encrypt and decrypt the data. The key is only known to authorized parties and is used to encrypt and decrypt the data. AES uses symmetric encryption, which means that the same key is used for both encryption and decryption. This makes the process of encrypting and decrypting the data fast and efficient.





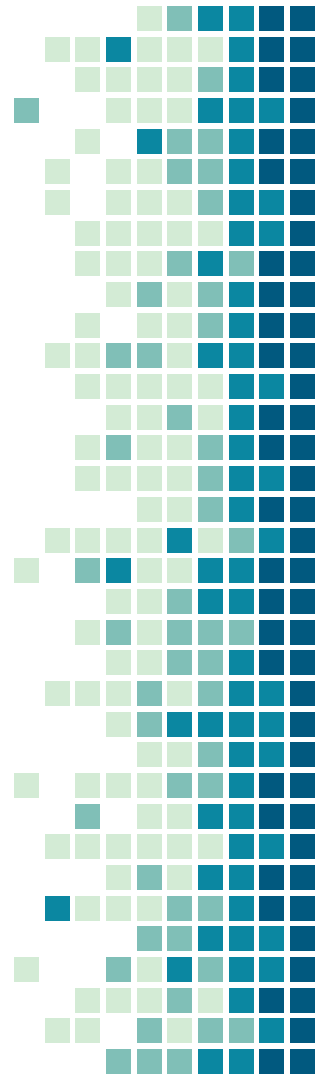
# Applications of AES

**Data encryption:** AES is used to encrypt sensitive data such as credit card information, medical records, and personal information. AES ensures that the data is protected against unauthorized access and that only authorized parties can access the data.

**Network security:** AES is used to secure communication channels such as VPNs (Virtual Private Networks) and SSL (Secure Socket Layer) connections. AES ensures that the data transmitted over the network is encrypted and cannot be intercepted by unauthorized parties.

**Disk encryption:** AES is used to encrypt data at rest such as on hard drives, USB drives, and other storage devices. AES ensures that the data is protected even if the storage device is stolen or lost.

**Database encryption:** AES is used to encrypt sensitive data stored in databases. AES ensures that the data is protected against unauthorized access and that only authorized parties can access the data.

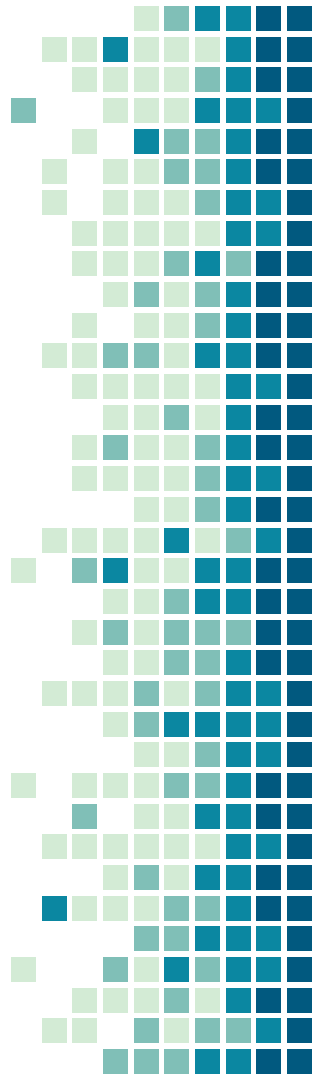


# Modules Imported

hashlib-The hashlib module provides a helper function for efficient hashing of a file or file-like object.

Pycryptodome-The Pycryptodome library offers implementations for techniques such as AES,RSA

base 64- The base64 encoding scheme is used to convert arbitrary binary data to plain text.

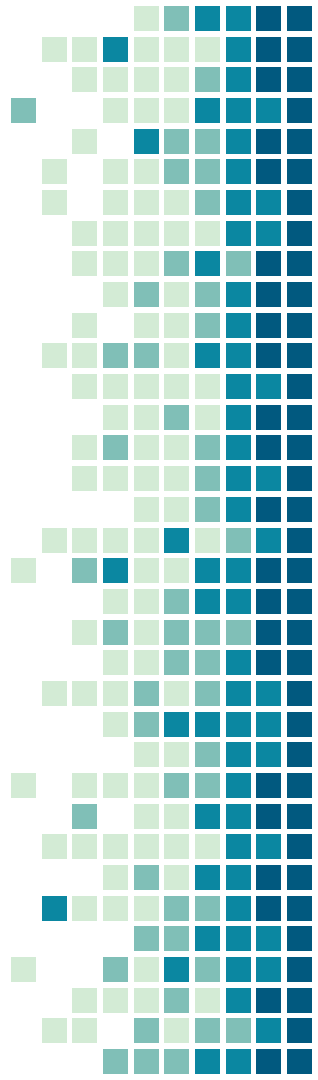


# Working of AES Algorithm:

Pad method:

The pad method receives the plain text to be encrypted and adds a number bytes for the text to be a multiple of 128 bits. This number is stored in the number of bytes to pad. Then in the ascii string we generate our padding character, and the padding str will contain that character times the number of bytes to pad. So we only have to add padding str at the end of our plain text so that it is now a multiple of 128 bits.

```
def pad(s):  
    return s.encode() + (AES.block_size - len(s.encode()) % AES.block_size) *  
    bytes([AES.block_size - len(s.encode()) % AES.block_size])
```



## Encrypt function:

The encrypt method receives the plain text to be encrypted. First we pad that plain text in order to be able to encrypt it. After we generate a new random IV(Initialization Vector) with the size of an AES block, 128 bits. We now create our AES cipher with AES.new with our key, in CBC (Cipher Block Chaining) mode and with our just generated IV. We now invoke the encrypt function of our cipher, passing it our plain text converted to bits. The encrypted output is then placed after our IV and converted back from bits to readable characters.

```
def encrypt(plaintext, key):  
    plaintext = pad(plaintext)  
    iv = os.urandom(AES.block_size)  
    cipher = AES.new(key, AES.MODE_CBC, iv)  
    return base64.b64encode(iv + cipher.encrypt(plaintext)).decode("utf-8")
```



## Decrypt function:

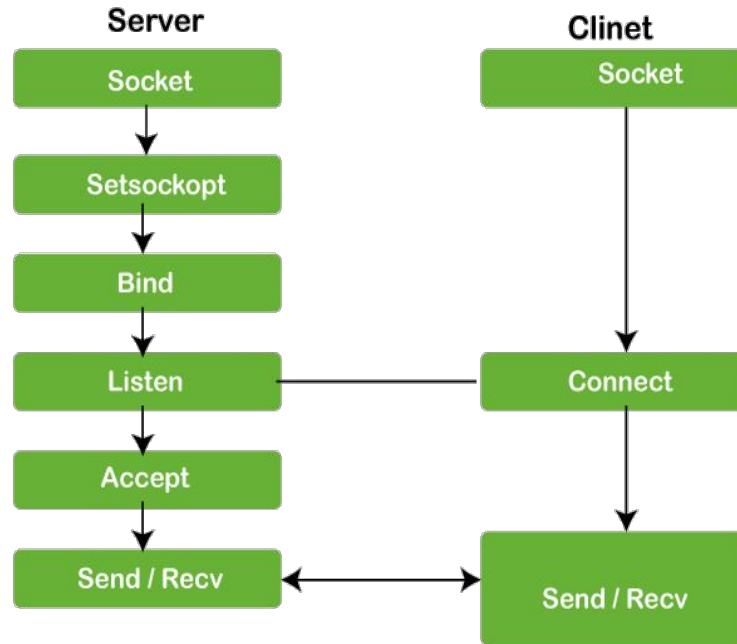
In order to decrypt, we must backtrack all the steps done in the encrypt method. First we convert our encrypted text to bits and extract the IV , which will be the first 128 bits of our encrypted text. Much like before, we now create a new AES cipher with our key, in mode CBC and with the extracted IV . We now invoke the decrypt method of our cipher and convert it to text from bits

```
def decrypt(ciphertext, key):  
    ciphertext = base64.b64decode(ciphertext)  
    iv = ciphertext[:AES.block_size]  
    cipher = AES.new(key, AES.MODE_CBC, iv)  
    plaintext = cipher.decrypt(ciphertext[AES.block_size:]).rstrip(bytes([AES.block_size]))  
    return plaintext.decode("utf-8")
```

# Socket Programming:

Socket programming is a way of connecting two nodes on a network to communicate with each other.

One socket listens on a particular port at an IP, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server.

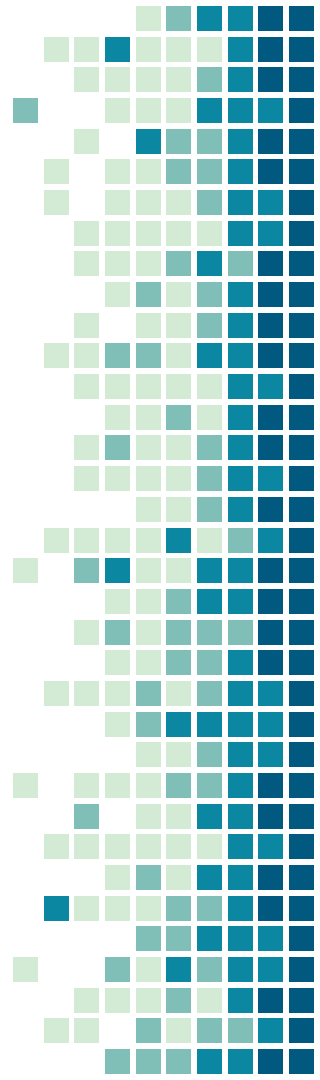


# Implementation of AES using Socket Programming:

Using Socket Programming and AES we could simulate how texts between two users are encrypted and decrypted respectively.

Key calculated from ECDH is used in the below program

Key-'0x173b8f4e43926784343867c1b8352bc0b54994745f39  
e64bd67c67a289d371230'



# Server Program

```
import socket
import base64
import hashlib
import os
from Crypto.Cipher import AES

def pad(s):
    return s.encode() + (AES.block_size - len(s.encode()) % AES.block_size) * bytes([AES.block_size - len(s.encode()) % AES.block_size])

def encrypt(plaintext, key):
    plaintext = pad(plaintext)
    iv = os.urandom(AES.block_size)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return base64.b64encode(iv + cipher.encrypt(plaintext)).decode("utf-8")

def decrypt(ciphertext, key):
    ciphertext = base64.b64decode(ciphertext)
    iv = ciphertext[:AES.block_size]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext[AES.block_size:]).rstrip(bytes([AES.block_size]))
    return plaintext.decode("utf-8")

new_socket = socket.socket()
host_name = '192.168.1.8'
s_ip = socket.gethostbyname(host_name)
port = 12345
new_socket.bind((host_name, port))
print("My IP: ", s_ip)
name = input('Enter Your name: ')
new_socket.listen(1)
conn, add = new_socket.accept()
client = (conn.recv(1024)).decode()
print(client + ' has connected.')
conn.send(name.encode())
```





```
conn.send(name.encode())

while True:

    message = input('Me : ')
    message = encrypt(message, hashlib.sha256('0x173b8f4e43926784343867c1b8352bc0b54994745f39e64bd67c67a289d371230'.encode()).digest())
    conn.send(message.encode())
    message = conn.recv(1024)
    print("Encrypted Message",message)
    message = decrypt(message, hashlib.sha256('0x173b8f4e43926784343867c1b8352bc0b54994745f39e64bd67c67a289d371230'.encode()).digest())
    print(client, ':', message)
```

## Output:

```
My IP: 192.168.1.8
Enter Your name: Alice
Bob has connected.
Me : Hello Bob
Encrypted Message b'Wa/Pj051iJnsRLWiBfrNIRSmwsvEzWEz65JnqjsyCmA='
Bob : Hello Alice
Me : How is the weather today?
Encrypted Message b'YudH/52AAF1Vvn7Z1B/vNaHB8Dcxpd/qZVRv+ZK9KXvNqUVy7s6GUpLV4eCyzPvh '
Bob : It is clear and sunny
```



# Client Program

```
import socket
import base64
import hashlib
import os
from Crypto.Cipher import AES

def pad(s):
    return s.encode() + (AES.block_size - len(s.encode()) % AES.block_size) * bytes([AES.block_size - len(s.encode()) % AES.block_size])

def encrypt(plaintext, key):
    plaintext = pad(plaintext)
    iv = os.urandom(AES.block_size)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return base64.b64encode(iv + cipher.encrypt(plaintext)).decode("utf-8")

def decrypt(ciphertext, key):
    ciphertext = base64.b64decode(ciphertext)
    iv = ciphertext[:AES.block_size]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext[AES.block_size:]).rstrip(bytes([AES.block_size]))
    return plaintext.decode("utf-8")

socket_server=socket.socket()
server_host="192.168.1.8"
ip=socket.gethostbyname([server_host])
port=12345
print('your ip address :',ip)
server_host=input('friend ip address:')
name=input('Your name : ')

socket_server.connect((server_host,port))

socket_server.send(name.encode())
server_name=socket_server.recv(1024)
server_name=server_name.decode()
print(server_name,'joined')
```

```
print(server_name,'joined')

while True:
    message=(socket_server.recv(1024)).decode()
    print("Encrypted Message",message)
    message = decrypt(message, hashlib.sha256('0x173b8f4e43926784343867c1b8352bc0b54994745f39e64bd67c67a289d371230'.encode()).digest())
    print(server_name,":",message)
    message=input("Me: ")
    message = encrypt(message, hashlib.sha256('0x173b8f4e43926784343867c1b8352bc0b54994745f39e64bd67c67a289d371230'.encode()).digest())
    socket_server.send([message.encode()])
```

## Output:

```
your ip address : 192.168.1.8
friend ip address:192.168.1.8
Your name : Bob
Alice joined
Encrypted Message pap1lsHm6nupEA2bIP5vTZoZCukvFI/0SopFsTZwsPk=
Alice : Hello Bob
Me: Hello Alice
Encrypted Message 03H2LRW5+BIY+3i+J56IjWKsIDLi+70NVWSrmvmBImtAsKyiW3GQAnXmVivn6tK
Alice : How is the weather today?
Me: It is clear and sunny
```



# Inferences:

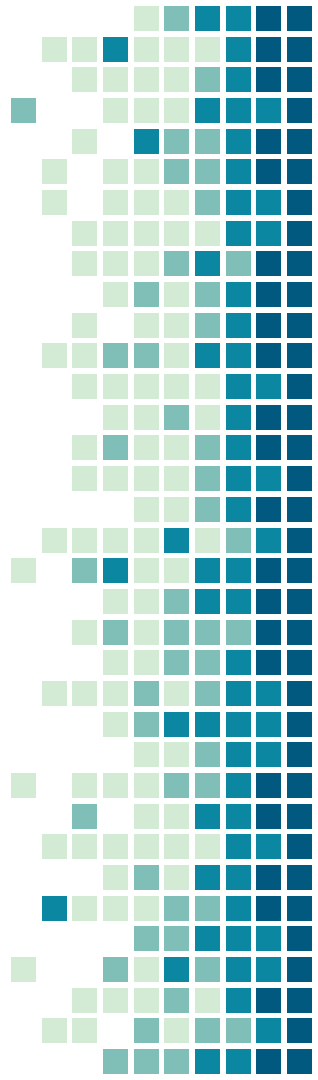
The server Program is executed first ,then the client Program

When Alice sends a message "Hello Bob" it is sent in an encrypted form using AES Encryption and Bob Decrypts the Encrypted Message to find the plain text "Hello Bob"

When Bob sends a message "Hello Alice" it is sent in an encrypted form using AES Encryption and Alice Decrypts the Encrypted Message to find the plain text "Hello Alice"

Many more messages can be sent like this to each other and AES encryption is performed to keep the text protected using the key which was obtained by ECDH and the message is encrypted using the key which was obtained by ECDH.

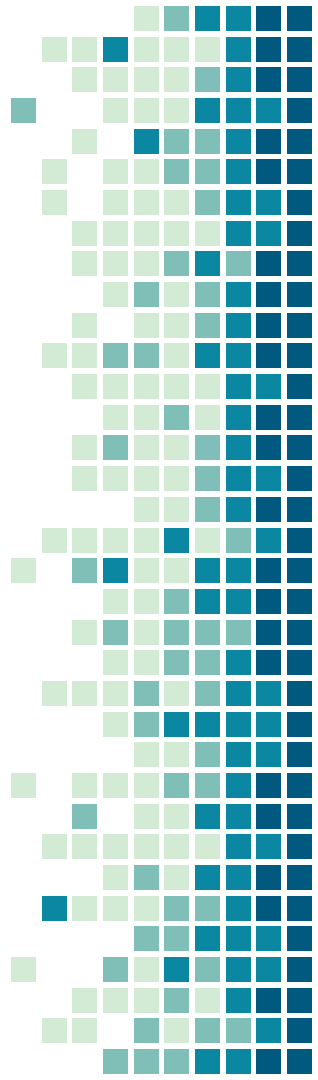
Therefore the key obtained from ECDH can be used in many encryption techniques such as AES,etc.



## Variant of Diffie Hellman using SHA -256 to detect Man in the Middle Attacks:

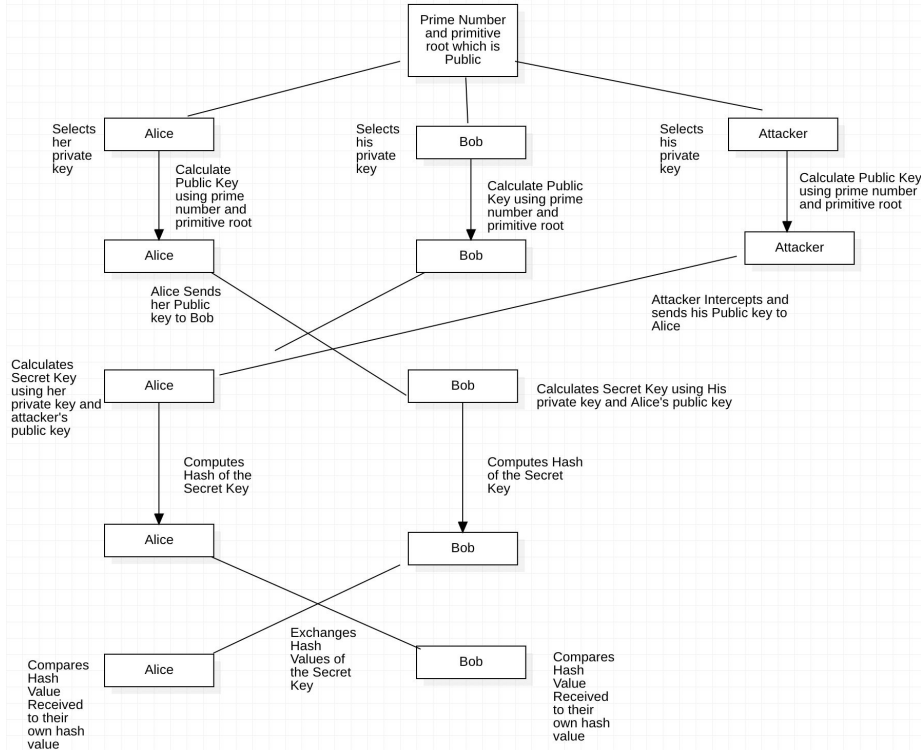
One of the major drawbacks of the Classic Diffie Hellman Algorithm is the inability to detect Man in the Middle Attack .

Here we propose an improved version of Diffie Hellman using SHA 256 to detect Man in the Middle Attack



# Architecture Diagram of DH using SHA 256

Architecture Diagram of Diffie Hellman Using SHA 256



This Diagram depicts the working of a variant Diffie Hellman using SHA 256 to detect Man-In-the-Middle Attacks.

If the transmission of public keys is compromised or there is any suspicion that it might be compromised. We could use this method to find whether an attack is performed or not.

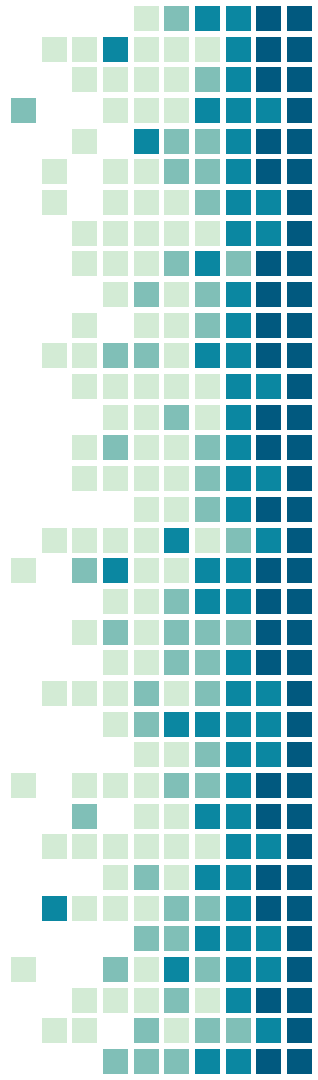
The Secret Keys Values are hashed and then compared by both the parties involved if there is any differences in the hash values. It can be concluded that the transmission is compromised and it is not safe to communicate anymore.

# SHA 256 [Secure Hash Algorithm]

SHA - 256 SHA-256 is a cryptographic hash function that is commonly used in various security applications, including digital signatures and message authentication. It takes an input message of any length and generates a fixed-size output of 256 bits (32 bytes) in length. This output is typically represented as a hexadecimal string of 64 characters.

SHA-256 is used in detecting Man-in-the-Middle (MitM) attacks by providing message integrity and authenticity. A MitM attack occurs when a third party intercepts and alters communications between two parties without either party knowing. To detect MitM attacks, SHA-256 is often used in conjunction with digital certificates.

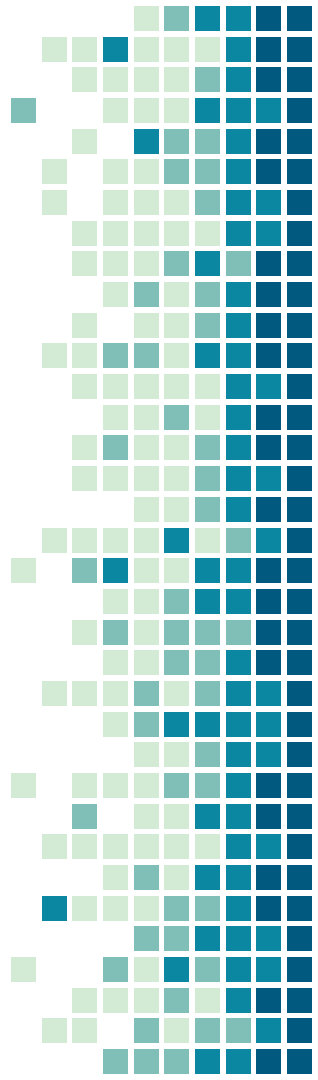
In secure communication, a digital certificate is used to verify the identity of the sender and receiver. The certificate is issued by a trusted Certificate Authority (CA) and contains the public key of the certificate holder. Before the communication begins, the sender and receiver exchange certificates and use the public key to encrypt and verify messages. The receiver can then use the private key to decrypt and verify the message.



SHA-256 is used to create a hash of the digital certificate. This hash is then signed using the private key of the certificate holder. The receiver can then use the sender's public key to verify the signature and the hash. If the hash and signature match, the receiver knows that the certificate has not been tampered with and can trust the sender's identity.

If a MitM attacker intercepts the communication and attempts to modify the certificate or message, the hash will not match, and the receiver will know that the communication has been compromised. This is because any modification made to the certificate or message will result in a different hash value, which will not match the original hash value that was signed by the certificate holder.

In summary, SHA-256 is used in detecting MitM attacks by providing message integrity and authenticity. By creating a hash of the digital certificate and signing it using the private key of the certificate holder, SHA-256 allows the receiver to verify the identity of the sender and detect any tampering with the communication.

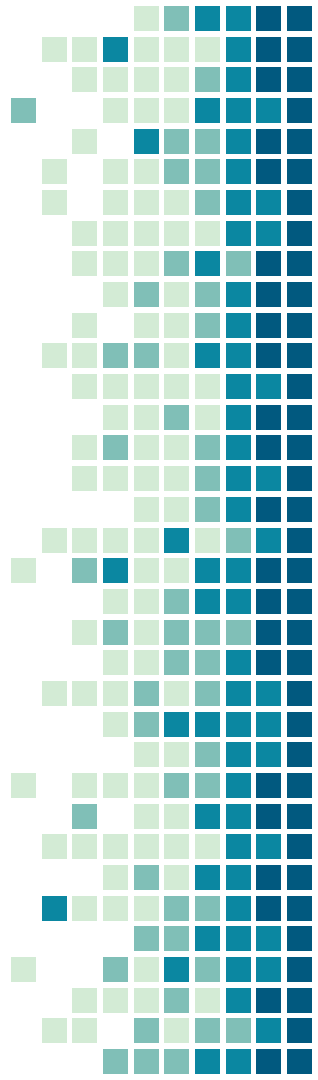




# Modules Imported

hashlib-The hashlib module provides a helper function for efficient hashing of a file or file-like object.

hash.hexdigest() returns a string object of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.



# Implementation

```
import random
import hashlib

# Define the prime number and primitive root
p = 487
g = 3

# Define Alice and Bob's private and public keys
a = 17 #alice private key
A = pow(g, a, p) # public key

b = 23 #bob private key
B = pow(g, b, p) #public key

# Man-in-the-Middle attacker intercepts Bob's public key and sends his own public key
M = 44 #Attacker Private Key
A_mitm = pow(g, M, p) #public key

# Alice computes the shared key and sends it to Bob

K_Alice = pow(B, a, p) #alice shared key using bob's public key
K_Alice_mitm = pow(A_mitm, a, p) #alice shared key using attacker's public key

# Bob computes the shared key and sends it to Alice
K_Bob = pow(A, b, p) #bob shared key using alice's public key

k_Mitm=pow(A,M,p) #attacker shared key using alice public key
# Compute the hash of the shared key
```



```
# Compute the hash of the shared key

hash_K_Alice = hashlib.sha256(str(K_Alice).encode()).hexdigest()
print("\noriginal alice secret key hash ",hash_K_Alice)

hash_K_Alice_mitm = hashlib.sha256(str(K_Alice_mitm).encode()).hexdigest()
print("\nalice secret key hash using attacker's public key",hash_K_Alice_mitm)
hash_K_Bob = hashlib.sha256(str(K_Bob).encode()).hexdigest()
print("\nbob secret key hash using alice's public key",hash_K_Bob)

hash_K_attack = hashlib.sha256(str(k_Mitm).encode()).hexdigest()
print(" \nhash of attacker secret key using alice's public key",hash_K_attack)

# Verify that the hash of the shared key matches between Alice and Bob
if (hash_K_Alice_mitm != hash_K_Bob): # checking the equality of alice secret key using attacker and bob secret key
    print("\nMan-in-the-Middle attack detected! Alice and Bob cannot communicate securely.")
else:
    print("\n Alice and Bob can communicate securely")
```

## Output:

```
Yuthishkumars-MacBook-Air:~ yuthishkumar$ /usr/local/bin/python3 /Users/yuthishkumar/Downloads/hashing.py
original alice secret key hash  093434a3ee9e0a010bb2c2aae06c2614dd24894062a1caf26718a01e175569b8

alice secret key hash using attacker's public key 7f2253d7e228b22a08bda1f09c516f6fead81df6536eb02fa991a34bb38d9be8

bob secret key hash using alice's public key 093434a3ee9e0a010bb2c2aae06c2614dd24894062a1caf26718a01e175569b8

hash of attacker secret key using alice's public key 7f2253d7e228b22a08bda1f09c516f6fead81df6536eb02fa991a34bb38d9be8

Man-in-the-Middle attack detected! Alice and Bob cannot communicate securely.
Yuthishkumars-MacBook-Air:~ yuthishkumar$ █
```



# Inference

In this Example Consider a third person who is an attacker trying to intercept the conversation between Alice and Bob.

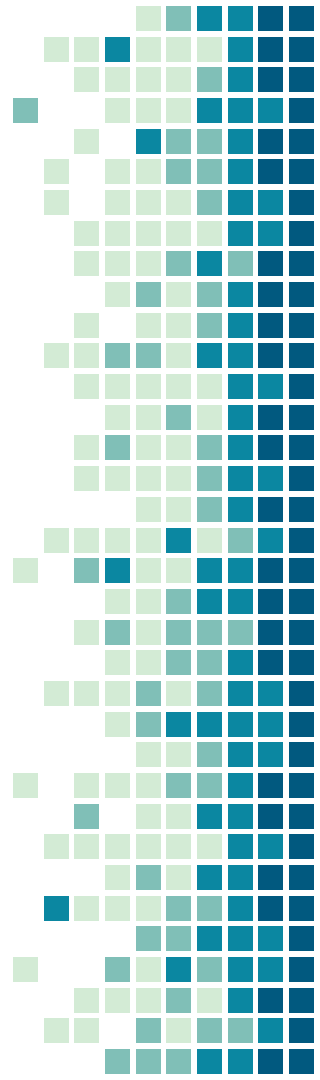
The attacker calculates his own Public Key using the values of the Prime number and Primitive Root which are made Public.

The Attacker intercepts Bob's public key and sends his own public key to Alice .Alice without knowing this computes a secret key.

The Secret keys are all hashed using SHA 256 Hashing technique.

In the above Code we compare the hashed value of Alice's Secret key using the attacker's public key and Bob's Secret key using Alice's Public Key

When Alice sends her hash value to Bob ,he will notice that it is not the same as his own hash value and it can be understood that their communication is compromised and they cannot communicate securely.



# Performance analysis

Based on Time Complexity:

The time complexity of the given code snippet is mainly determined by the number of scalar multiplication operations performed, which is equal to 2 in this case (one for Alice and one for Bob).

The time complexity can be approximated as  $O(\log n)$ , where  $n$  is the largest possible value of the scalar being multiplied (in this case,  $n$  is the order of the elliptic curve group, which is a large prime number). Whereas the overall time complexity of the diffie-Hellman is  $O(\log P)$ , where  $P$  is the prime number chosen.



## Elliptic Curve Diffie hellman

```
print("Equal shared keys:", alicesSharedKey == bobSharedKey)

end = time.time()

current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Current memory usage is {current / 10**6}MB; Peak was {peak / 10**6}MB")
print(f"Total time taken is {end - start} seconds")
```

```
➔ Alice priv key: 57465177845865985685302298237295124776834035058810400635620051691515523728057
Alice public key: 0x603ca4f3bcaef4301a25e45b6a27d77e03e1acc677c6bcb5925d5de00f76c1a1
Bob public key: 0x91f4a7c48748c662e2a7277a037267f8b593d9c196d9bb68bbdbcd5fcc38f48c1
Now exchange the public keys (e.g. through Internet)
Alice shared key: 0x2dc2c404f726ccad28807a3e509b2e082bc9db3e380a6186fec933c98ca44bc0
Bob shared key: 0x2dc2c404f726ccad28807a3e509b2e082bc9db3e380a6186fec933c98ca44bc0
Equal shared keys: True
Current memory usage is 0.124226MB; Peak was 0.377472MB
Total time taken is 3.602642059326172 seconds
```

## diffie hellman

```
# current = float((current / (10000)))
# print("Current memory usage is : ",(cuurent))
```

```
➔ The Value of Prime number P is :1571
The Value of primitive root q is :234
The Private Key for Sender is :17
The Public Key generated by the Sender is :304
The Private Key for Receiver is :23
The Public Key generated by the Receiver is :1132
Secret key for the Sender is : 348
Secret Key for the Receiver is : 348
time taken to for the program to execute : 3.01100492477417 Seconds
Current memory usage is 0.151201MB; Peak was 0.546757MB
Total time taken is 3.01100492477417 seconds
```

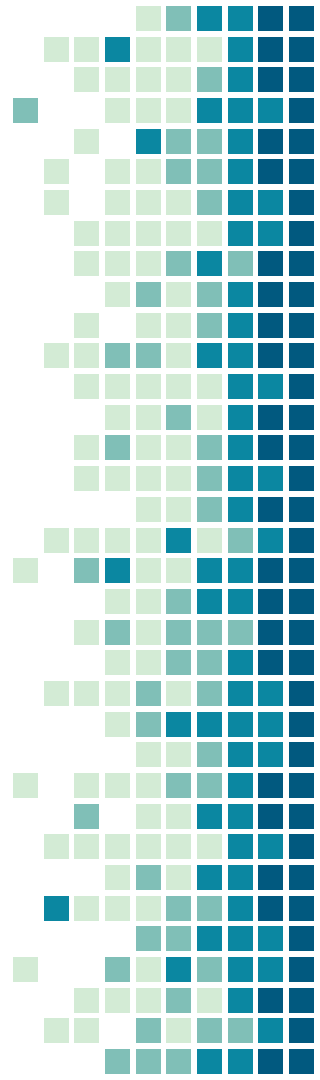
Based upon total time taken:

### DH:

```
The Value of Prime number P is :1571
The Value of primitive root q is :234
The Private Key for Sender is :17
The Public Key generated by the Sender is :304
The Private Key for Receiver is :23
The Public Key generated by the Receiver is :1132
Secret key for the Sender is : 348
Secret Key for the Receiver is : 348
Time taken for program to execute :3.011 seconds
```

### ECDH:

```
Alice public key: 0x4e5174463153a3d2593e48e4cf0835d70796d49dc4ce3fd86508ace517f9a78e1
Bob public key: 0x5dbe3b78fc136dc7077ffa4694ee1d6225fa3c3cb1686eb7f0fd3fee03b0dbf00
Exchanging Public Keys
Alice shared key: 0x5a38d724b41a95862c10fa5eb70d0375e590cc8812162f3b22ad49fb94b1320
Bob shared key: 0x5a38d724b41a95862c10fa5eb70d0375e590cc8812162f3b22ad49fb94b1320
Equal shared keys: True
Time taken for program to execute 0.1631 seconds
```



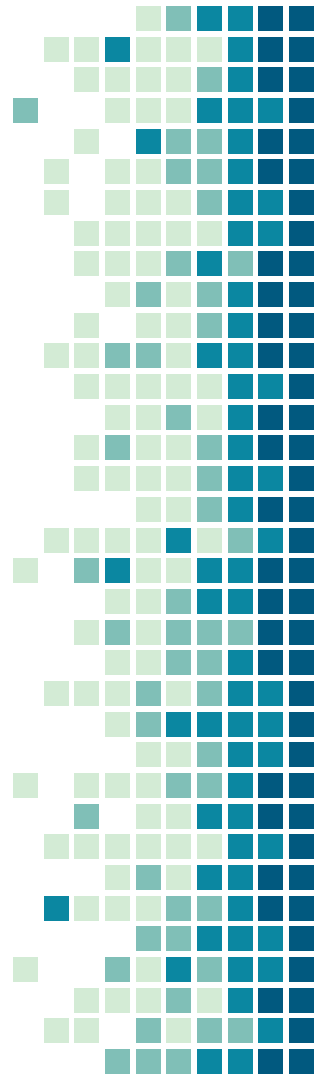
Bases on memory consumption:

### DH:

```
The Value of Prime number P is :1571
The Value of primitive root q is :234
The Private Key for Sender is :17
The Public Key generated by the Sender is :304
The Private Key for Receiver is :23
The Public Key generated by the Receiver is :1132
Secret key for the Sender is : 348
Secret Key for the Receiver is : 348
Current Memory Usage is 0.1521 MB ; Peak was 0.5467 MB
```

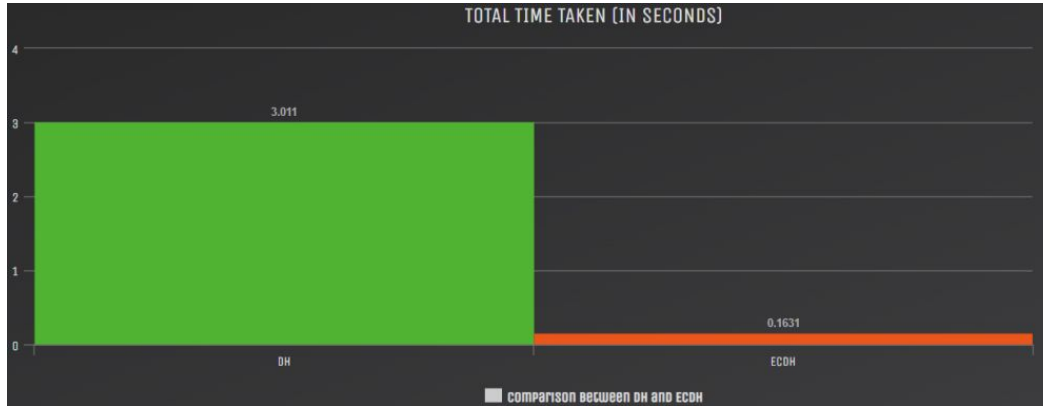
### ECDH:

```
Alice public key: 0xdf54d8e8092f56be020b9045323d4521d8a5e1d169022ce6d8d6eb4c96c63b1
Bob public key: 0x3198db1cfb4173a5d66aacbb7acdbbf9fbad75b67fe387024cb34bc25739148e1
Exchanging Public Keys
Alice shared key: 0x2bda52f3dfb123d6e5e62315392469bbcf7c90313fae9b9053a9dbf059af343c0
Bob shared key: 0x2bda52f3dfb123d6e5e62315392469bbcf7c90313fae9b9053a9dbf059af343c0
Equal shared keys: True
Current Memory Usage is 0.0269 MB ; Peak was 0.0377 MB
```

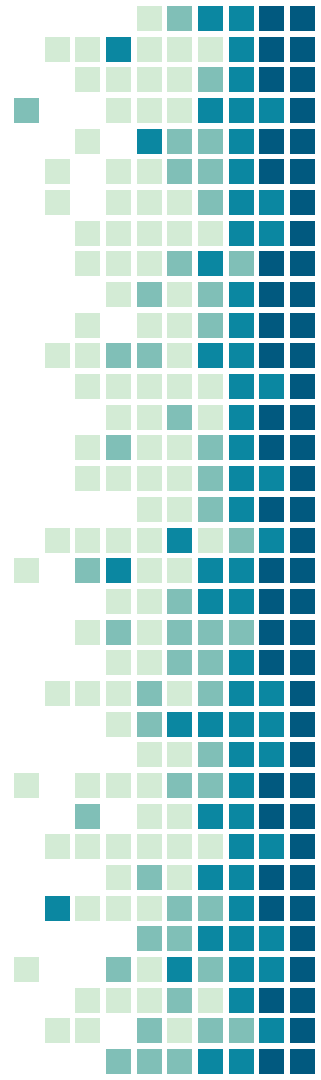
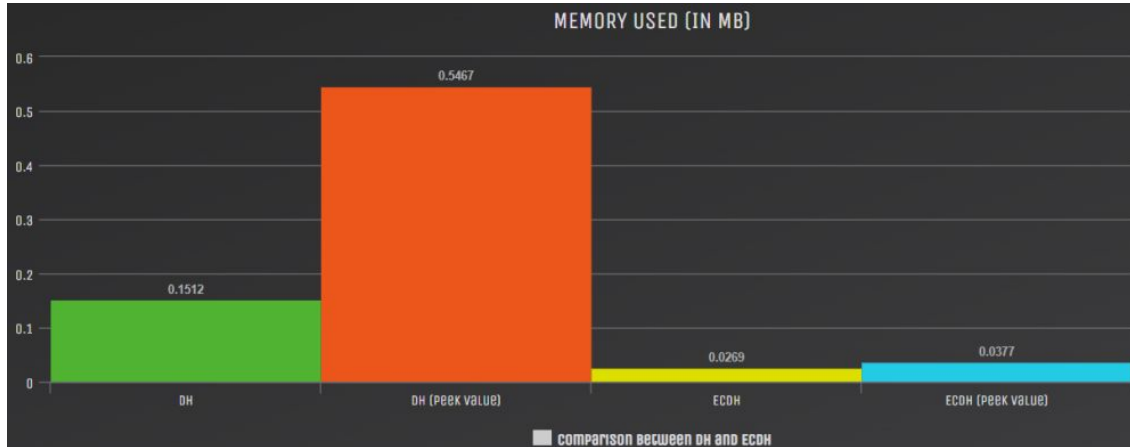




Based upon execution time:



Bases on memory consumption:



## Inference from the performance analysis:

- ❖ ECDH key exchange is generally faster than DH key exchange due to the smaller key sizes and more efficient mathematical operations, the total time taken for key exchange depends on a variety of factors and may vary widely depending on the specific application and implementation details
- ❖ ECDH key exchange is based on elliptic curve cryptography, which involves performing mathematical operations on points on an elliptic curve. The key size for ECDH is typically smaller than DH, with 256-bit and 384-bit curves being commonly used. This means that ECDH requires less memory than DH to store the keys and perform the mathematical operations.
- ❖ The performance analysis of DH and ECDH suggests that both algorithms provide secure key exchange, but ECDH is generally considered more efficient due to its use of elliptic curves. This efficiency results from the fewer arithmetic operations required in ECDH compared to DH. However, the performance of both algorithms can also be influenced by parameter choices such as the size of prime modulus in DH and the choice of elliptic curve in ECDH.
- ❖ Overall, the selection of the appropriate algorithm and parameters depends on the specific security requirements and computational resources of the application. For applications that require higher levels of security, larger modulus sizes or curve sizes may be necessary, which can increase computational costs. For applications with limited computational resources, ECDH may be a better choice due to its smaller computational cost.



# Differences between classical DH and DH with SHA 256

## Diffie Hellman

```
The Value of Prime number P is :487
The Value of primitive root q is :3
The Private Key for Sender is :17
The Public Key generated by the Sender is :425
The Private Key for Receiver is :23
The Public Key generated by the Receiver is :93
Secret key for the Sender is : 211
Secret Key for the Receiver is : 211
time taken to for the program to execute : 8.416175842285156e-05 Seconds
Current memory usage in Kb is : 0.062
```

## Diffie Hellman with SHA 256

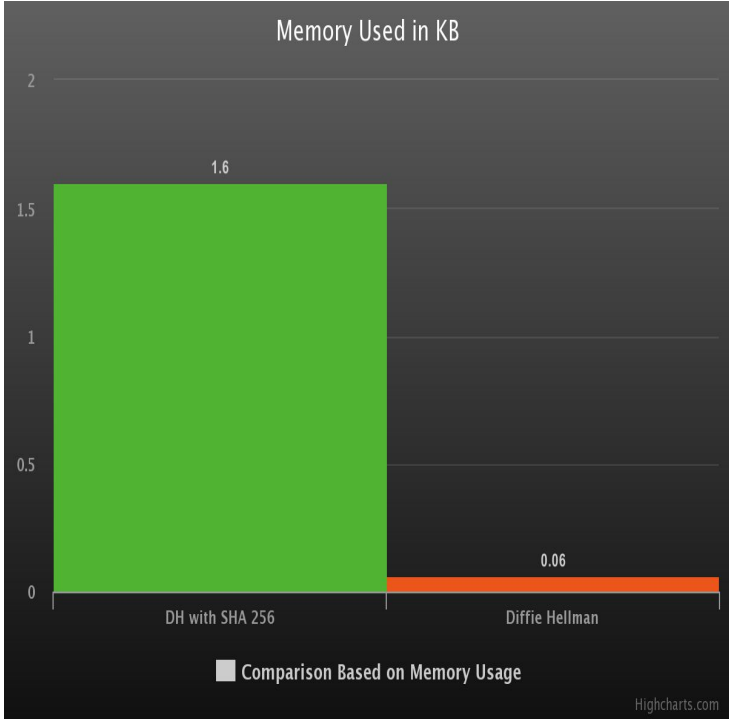
```
original alice secret key hash 093434a3ee9e0a010bb2c2aae06c2614dd24894062a1caf26718a01e175569b8
alice secret key hash using attacker's public key 7f2253d7e228b22a08bda1f09c516f6fead81df6536eb02fa991a34bb38d9be8
bob secret key hash using alice's public key 093434a3ee9e0a010bb2c2aae06c2614dd24894062a1caf26718a01e175569b8
hash of attacker secret key using alice's public key 7f2253d7e228b22a08bda1f09c516f6fead81df6536eb02fa991a34bb38d9be8
Man-in-the-Middle attack detected! Alice and Bob cannot communicate securely.
time taken to for the program to execute : 7.677078247070312e-05 Seconds
Current memory usage in KB is : 1.654
```



# Based on Execution Time



# Based on Memory Usage



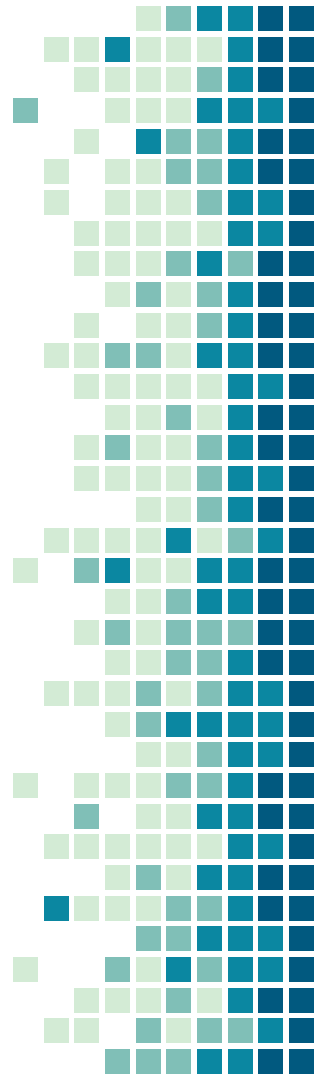
# Inference

When we compare the above two outputs it can be inferred that the time taken to run the program is similar for the normal DH and DH with SHA 256.

But when we compare the memory used by each program the normal DH uses less memory compared to DH with SHA 256 which is expected because the Diffie Hellman variant also performs hashing unlike the first one.



# General overview of security between DH and ECDH



# DH Key Exchange

DH key exchange is based on modular exponentiation, which is a well-studied mathematical operation. However, DH key exchange is vulnerable to certain attacks, such as the man-in-the-middle (MITM) attack, which can allow an attacker to intercept and modify the exchanged keys. DH key exchange can also be vulnerable to brute-force attacks if the key size is too small or if the prime number used is not large enough.



## ECDH Key Exchange

ECDH key exchange, on the other hand, is based on elliptic curve cryptography, which provides stronger security for the same key sizes as compared to modular exponentiation used in DH key exchange. ECDH key exchange is not vulnerable to the same types of attacks as DH key exchange, such as the small subgroup confinement attack and the invalid curve attack. However, ECDH key exchange is vulnerable to certain types of attacks, such as side-channel attacks that exploit implementation weaknesses, and quantum attacks that could break the underlying elliptic curve cryptography.





## Comparison

In terms of key sizes, ECDH key exchange can use smaller key sizes than DH key exchange for equivalent security. For example, a 256-bit ECDH key provides the same level of security as a 3072-bit DH key. This means that ECDH key exchange can provide equivalent security with less computational overhead and smaller key sizes.

ECDH key exchange has some advantages over DH key exchange in terms of security, key sizes, and computational overhead. However, the choice of key exchange protocol depends on the specific application requirements and the threat model, and both protocols have their own strengths and weaknesses that should be carefully evaluated.



# Conclusion:

- Performance: DH and ECDH are both asymmetric key agreement protocols used for key exchange. In terms of performance, ECDH is generally faster than DH, especially for longer key sizes, because it uses elliptic curve cryptography, which requires fewer computations than traditional DH.
- Security: Both DH and ECDH are considered secure key exchange protocols, as long as they are implemented correctly and used with sufficiently large key sizes. However, ECDH is generally considered to be more secure than DH, because it offers equivalent security with shorter key sizes, which makes it less vulnerable to attacks such as the discrete logarithm problem.

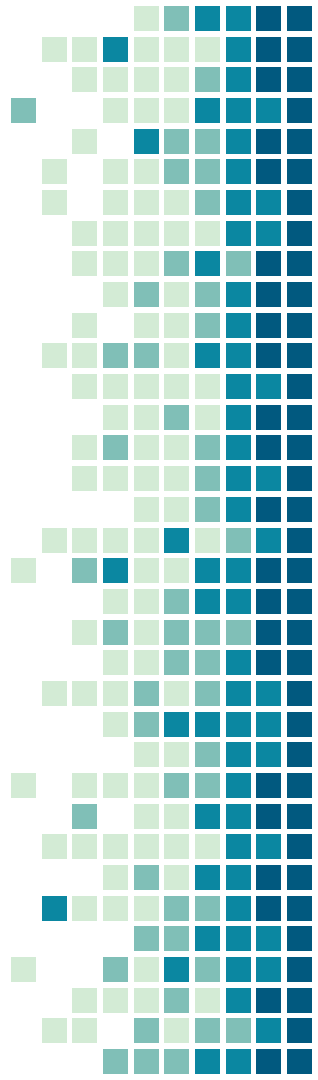


# Conclusion

ECDH offers better performance and security than DH, especially for modern applications where efficiency and security are both important factors to consider

However, it's crucial to use appropriate key sizes and to implement the protocols correctly to ensure their security.

DH with SHA 256 can be used to detect Man in the Middle attacks by comparing the hashed values of secret keys, which the normal Diffie Hellman protocol cannot do. In terms of execution time, both protocols are similar.



# REFERENCES

1. Mitra, S., Das, S., Kule, M. (2021). Prevention of the Man-in-the-Middle Attack on Diffie-Hellman Key Exchange Algorithm: A Review. In: Bhattacharjee, D., Kole, D.K., Dey, N., Basu, S., Plewczynski, D. (eds) Proceedings of International Conference on Frontiers in Computing and Systems. Advances in Intelligent Systems and Computing, vol 1255. Springer, Singapore.  
[https://doi.org/10.1007/978-981-15-7834-2\\_58](https://doi.org/10.1007/978-981-15-7834-2_58)
2. Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. 2007. Provably secure authenticated group Diffie-Hellman key exchange. ACM Trans. Inf. Syst. Secur. 10, 3 (July 2007), 10-es.  
<https://doi.org/10.1145/1266977.1266979>
3. Ripon Patgiri. (2021). privateDH: An Enhanced Diffie-Hellman Key-Exchange Protocol using RSA and AES Algorithm.
4. Mishra, Manoj & Kar, Jayaprakash. (2017). A study on diffie-hellman key exchange protocols. International Journal of Pure and Applied Mathematics.  
<http://www.ijpam.eu/contents/2017-114-2/2/>
5. Aryan et al 2017 IOP Conf. Ser.: Mater. Sci. Eng. 263 042015

6. Lang, Joanna. "The Elliptic Curve Diffie-Hellman (ECDH)." (2015).
7. L. Harn, M. Mehta and Wen-Jung Hsin, "Integrating Diffie-Hellman key exchange into the digital signature algorithm (DSA)," in IEEE Communications Letters, vol. 8, no. 3, pp. 198-200, March 2004, doi: 10.1109/LCOMM.2004.825705.  
<https://ieeexplore.ieee.org/document/1278320>
8. International Journal of Network Security, Vol.20, No.6, PP.1221-1226, Nov. 2018 (DOI: 10.6633/IJNS.20181120(6).23)
9. International Journal of Science and Research (IJSR) ISSN (Online): 2319-7064 Volume 6 Issue 6, June 2017
10. Nan Li, "Research on Diffie-Hellman key exchange protocol," 2010 2nd International Conference on Computer Engineering and Technology, Chengdu, China, 2010, pp. V4-634-V4-637, doi: 10.1109/ICCET.2010.5485276  
<https://ieeexplore.ieee.org/document/5485276>
11. Rewagad and Y. Pawar, "Use of Digital Signature with Diffie Hellman Key Exchange and AES Encryption Algorithm to Enhance Data Security in Cloud Computing," 2013 International Conference on Communication Systems and Network Technologies, Gwalior, India, 2013, pp. 437-439, doi: 10.1109/CSNT.2013.97.  
<https://dl.acm.org/doi/10.1109/CSNT.2013.97>

12. Pan, J., Qian, C. & Ringerud, M. Signed (Group) Diffie-Hellman Key Exchange with Tight Security. J Cryptol 35, 26 (2022).  
<https://doi.org/10.1007/s00145-022-09438-y>
13. Raymond, Jean-francois & Stiglic, Anton. (2002). Security Issues in the Diffie-Hellman Key Agreement Protocol. IEEE Transactions on Information Theory. 22.
14. N. Mehibel and M. Hamadouche, "A new approach of elliptic curve Diffie-Hellman key exchange," 2017 5th International Conference on Electrical Engineering - Boumerdes (ICEE-B), Boumerdes, Algeria, 2017, pp. 1-6, doi: 10.1109/ICEE-B.2017.8192159  
<https://ieeexplore.ieee.org/document/8192159>
15. Ram Ratan Ahirwal et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 4 (2) , 2013, 363 - 36
16. Nashwan, Shadi. (2022). Secure Authentication Scheme Using Diffie-Hellman Key Agreement for Smart IoT Irrigation Systems. Electronics. 11. 188. 10.3390/electronics11020188.  
<https://www.mdpi.com/2079-9292/11/2/188>

17. International Journal of Electrical and Computer Engineering (IJECE)  
Vol. 12, No. 1, February 2022, pp. 853-858 ISSN: 2088-8708, DOI:  
10.11591/ijece.v12i1.pp853-858  
<https://ijece.iaescore.com/index.php/IJECE/article/view/24794>

18. Jooyoung Lee, & Je Hong Park. (2008). Authenticated Key Exchange Secure  
under the Computational Diffie-Hellman Assumption.

# TEAM PRESENTATION



**Nitin hariharan**

21MIA1141

Team Leader



**Yuthish**

21MIA1023

Teammate #1



**Goutham**

21MIA1014

Teammate #2



**Visva R**

21MIA1064

Teammate #3





*THANK YOU*