

# Private Group Communication in Blockchain Based on Diffie-Hellman Key Exchange

Zachary Laney<sup>1</sup> and Yoohwan Kim<sup>2</sup>

Department of Computer Science  
University of Nevada, Las Vegas, Nevada, 89154

<sup>1</sup>laneyz1@unlv.nevada.edu

<sup>2</sup>yoohwan.kim@unlv.edu

**Abstract**—One of the most important aspects of the distributed blockchain is that it allows the trade of information without having to depend on a single trusted third party to facilitate the interaction by providing authentication and validation. Decentralized Ledger Technology allows all parties to agree on the rules in advance and therefore also agree on the expected results. Instead of all parties having to depend on a single trusted mediator they can depend on the deterministic output of the rules governing the Decentralized Ledger Technology and the data not being manipulated by the protection of immutability. The upside to the public visibility of information is that it enables the neutral transfer of information and public verifiability with only a cryptocurrency wallet and a Public Key. The downside is that secret information like passwords cannot be transferred to other parties and still maintain its secrecy due to the data being publicly accessible to anyone with the motive to retrieve it. The solution to this problem is establishing a private encryption channel so that users may trade hidden information over the open blockchain. There are many proven techniques available on establishing a private encryption channel between two users like Elliptic Curve Cryptography and Diffie-Hellman. However, there are only a few protocols that investigate how to establish a private encryption channel with a group of more than two users and none of them are able to establish a private group communication channel on the open distributed blockchain. Our proposed solution involves a web interface that first requires users to publish pieces of information to a mediatory smart contract in order to establish a private channel with each user inside a group and then utilize these channels to share an encrypted group key over email. Our research, while only requiring access to the internet and an email address on a home computer, successfully achieved private group communication on the open distributed blockchain for a group size of 500 users in 0.852 seconds of time with a highly secure 3027 bit Diffie-Hellman Public Key.

**Index Terms**—Diffie-Hellman, Ethereum Smart Contract, Group Key, Encryption, Private Channel

## I. INTRODUCTION

The advancement of decentralized applications (dApps) in Ethereum have presented new opportunities as well as new challenges that need decentralized solutions that complement the benefits of the blockchain. Anonymity, privacy, and security are core aspects to a dApp that is built on the Ethereum Blockchain. There are many beneficial applications for establishing a private channel with a group on the blockchain including: decentralized private key infrastructure (PKI), private group chat rooms, group encrypted file sharing with IPFS, or

just the ability to pseudonymously establish a private group communication channel behind the identity of a Public Key and a cryptocurrency wallet.

## II. DISTRIBUTED TECHNOLOGY

### A. Distributed Blockchain

A distributed blockchain is similar to a regular database but the primary difference is that the blockchain is distributed among many unique computers whereas a regular database is only stored on a single computer. The official term for a distributed blockchain is known as Distributed Ledger Technology (DLT). The reason that the ledger is distributed among many computers is for the creation of a single data state which is referred to as the world state. The DLT software is constantly validating all of the ledger copies to maintain an accurate world state and will respond if a discrepancy appears thereby ensuring immutability.

### B. Ethereum Smart Contracts

The structure of DLT provides a way for distributed computer programs to be created, distributed, and executed consensually on all nodes. However, Bitcoin's architectural design never implemented the ability to store these distributed computer programs. Ethereum enables distributed computer programs known as smart contracts to be stored on its ledger and executed platform independently using the Ethereum Virtual Machine (EVM). A smart contract is a distributed Turing machine that is similar to a world computer and it trades software performance for its code to achieve immutability and transparency.

### C. Interplanetary File System (IPFS)

IPFS is a content-addressed peer-to-peer distributed file system. Any user connected to the internet can upload files to IPFS which are then stored on and downloaded from other peers. In order for a user to upload a file an IPFS access point is required. IPFS stores and downloads resources with the use of hashes. The IPFS client will ask the entire distributed network if it contains a peer with a resource that has a unique hash value. If the resource is located by its hash value it will be downloaded by the requestor.

### III. PRIVACY ON ETHEREUM BLOCKCHAIN

#### A. Diffie-Hellman (DH) Key Exchange Algorithm

DH key exchange is a protocol that allows two parties that have no prior knowledge of each other to establish a shared key over an unsecure communication channel. This shared key is then used to encrypt communications using a symmetric key. DH is currently used in many protocols including Secure Socket Layer (SSL), Transport Layer Security (TLS), Secure Shell (SSH), Internet Protocol Security (IPSec), and Public Key Infrastructure (PKI). But all of these protocols are conducted in a centralized communication architecture. Centralized communication systems suffer from certain characteristics like Distributed Denial of Service (DDoS) attacks, Man In The Middle (MITM) attacks, and service outages that decentralized communication architectures aren't as vulnerable to [9]. The reason why DH is vulnerable to MITM attack is because the protocol doesn't provide authentication on its own. The DH protocol by itself is only a mathematical formula to generate shared keys and doesn't verify the identities therefore it requires additional steps in order to provide a form of authentication as a defense.

#### B. Creating a pairwise key with DH

Diffie-Hellman is dependent upon the difficulty to compute discrete logarithms. The process begins by defining a primitive root of a number called  $m$  as one the one that generates all of the possible integers from  $1$  to  $m - 1$ . If  $g$  is a primitive root of the prime number  $m$  then the numbers:

$$g \bmod (m), g^2 \bmod (m), \dots, g^{m-1} \bmod (m)$$

are unique and consist of all possible integers from  $1$  through  $m - 1$ . For any possible integer  $i$  less than  $m$  and a primitive root  $g$  of prime number  $m$  it is possible to find an exponent  $t$  [4] such that:

$$i = g^t \bmod (m) \text{ where } 0 \leq t \leq (m - 1)$$

There are two publicly known numbers: a prime number  $p$  and an integer  $\lambda$  that is a primitive root of  $p$ . In the following example there are two users, User A and User B, that need to create a shared key in order to open a private channel on the blockchain. User A and User B choose a random number  $x$  and  $y$  respectively such that:

$$x < p$$

$$y < p$$

User A calculates their Public Key  $X$  such that:

$$X = \lambda^x \bmod (p)$$

User B calculates their Public Key  $Y$  such that:

$$Y = \lambda^y \bmod (p)$$

User A keeps  $x$  secret and publishes  $X$  to the smart contract while User B keeps  $y$  secret and publishes  $Y$  to the smart contract. User A and User B will then create a shared key

by retrieving  $X, Y$  values from the smart contract. User A calculates the shared key with:

$$S_A = Y^x \bmod (p)$$

User B calculates the shared key with:

$$S_B = X^y \bmod (p)$$

Resulting in User A and User B reaching the same result with:

$$S_A \equiv S_B$$

This process is illustrated in **Figure 1** below:

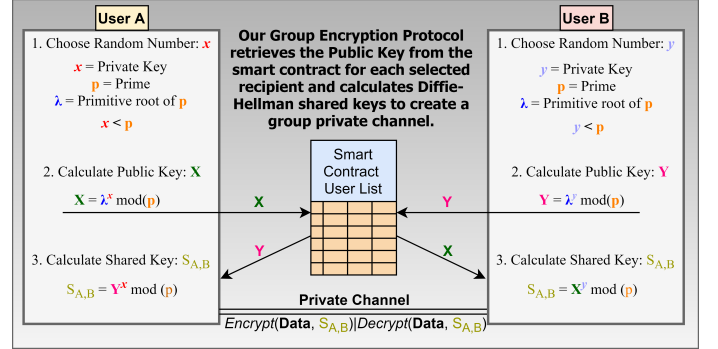


Fig. 1: The Group Leader (User B) calculates a pairwise Diffie-Hellman shared key for encrypting Group Key and sending it to  $g$  recipients (User A).

#### C. Current Research Status

Diffie-Hellman is currently used in many protocols including, Secure Socket Layer (SSL), Transport Layer Security (TLS), Secure Shell (SSH), Internet Protocol Security (IPSec), and Public Key Infrastructure (PKI) [6]. However, there has been minimal research on establishing a shared key with a group of users through the public data stored in a smart contract. Current research with IPFS and smart contracts only focus on the most optimal way to store and retrieve encrypted files uploaded in IPFS by indexing the returned hash strings inside of smart contracts [1]. Other research discusses similar attempts to establish group keys that support the adding and removal of users but requires a complicated communication architecture like binary trees and ring communication along with persistent data transfer [2] [3]. Our method proposes a simpler way of securely distributing a Group Key with a non-persistent communication to any amount of recipients by utilizing a JSON file, IPFS, and the recipient email addresses.

#### D. Challenges in Research and Practice

There are three primary challenges that private group communication over Ethereum Smart Contracts currently faces that our protocol attempts to solve:

- 1) The first challenge are the public discovery of Public Keys. The centralized Public Key discovery method require groups of users to aggregate on centralized communication channels like privately owned websites or in

person to trade information thereby exposing themselves to security vulnerabilities and privacy issues. Smart Contracts are accessible to the entire internet which simplifies the Public Key discovery process for establishing a private encryption channel. The decentralized nature of the smart contracts enables users to trade Public Keys behind the identity of only an Ethereum wallet address and a Public Key while benefiting from the security protections of the blockchain.

- 2) The second challenge involves maintaining perfect forward secrecy due to the fact that every transaction is permanently stored in the blockchain which allows another user to decrypt all of the private communications encrypted with the group key should the key be stolen in the future. Currently, the best way to maintain forward secrecy with our protocol is to continually publish a new Public Key and repeat the shared key process which can be expensive and inefficient. Additional research is needed to find the best solution to this problem.
- 3) The third challenge is MITM attacks and Impersonation attacks. Our protocol is protected by utilizing the immutable and openly verifiable nature of smart contracts as a trust anchor which minimizes the threat of these attacks more than other Group Key Encryption Protocols that require a centralized trusted third party.

#### IV. PROPOSED ARCHITECTURE FOR PRIVATE GROUP COMMUNICATION

##### A. Our Approach

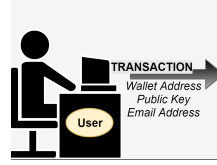
Our approach enables a user to access a web interface to publish their Public Key, Wallet Address, and Email Address on the distributed blockchain. A smart contract is utilized as a neutral mediator for DLT interaction. Any user of this group may self-elect to distribute a Group Key thereby becoming the Group Leader. The Group Leader selects recipients from the list of users in the smart contract to establish a group private communication channel by sending a Group Key encrypted with the Diffie-Hellman (DH) process to the email addresses published by the selected recipients.

##### B. Operation Procedure

Our approach utilizes email as a method to distribute the group key once it has been created by the Group Leader to lessen the total amount of transactions. However, this is not the only method that can be used for distribution. Another possible distribution method is by transacting the encrypted group key to the smart contract so that it can be directly retrieved from the blockchain by the recipients. This method would provide stronger anonymity and strengthen the non-persistent communication structure by removing all reliance on internet and email connectivity. For simplicity, our research will only explore how the encrypted group key may be distributed to the recipients by email.

**Figure 2** shows the table structure as it exists within the smart contract. The User Index is the numeric identifier of a user within the total number of users in the smart contract

list. The User Wallet Address is the cryptocurrency wallet address, in our case an Ethereum Wallet, that paid to publish the information to the smart contract. The User Email Address is the email address that a user has chosen to receive a notification when they are selected as a recipient for a group key.



User Index	User Wallet Address	User Public Key	User Email Address
1	Wallet Address	Public Key	Email Address
2	Wallet Address	Public Key	Email Address
...	...	...	...
n	Wallet Address	Public Key	Email Address

Fig. 2: Table of the users list as it exists within the smart contract.

**Figure 3** shows the process of a user publishing their information to the smart contract so that it may be publicly discovered so a group leader may send them a group key. The user first accesses the web interface and JavaScript uses web3.js to access MetaMask and extract their currently selected cryptocurrency wallet address. The user utilizes the web interface to enter an email address and generate a Private Key used to generate their Public Key. The private key should be stored somewhere secure as it is required to decrypt the private channel. The user clicks a join button that calls a smart contract function to publish their Public Key and Email Address. The user's Wallet Address, Public Key, and Email Address are transacted to the smart contract and saved in the blockchain's world state inside a user mapping. The gas cost of this transaction will depend on the size of the Public Key and Email Address used.

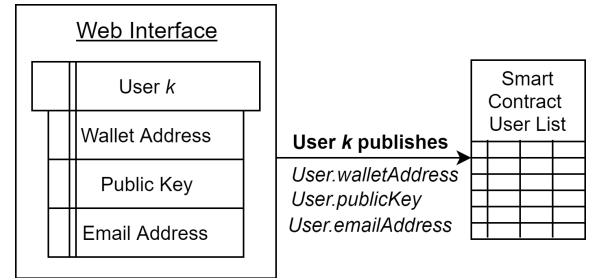


Fig. 3: A user utilizes a web interface to publish their Public Key, Wallet Address, and Email Address to the smart contract.

**Figure 4** shows the process of a Group Leader  $k$  sending a Group Key to  $g$  recipients within the Smart Contract User List.

The total amount of users within a group is referred to as  $n$ . The index for the Group Leader of a group is referred to as  $k$ . The number of recipients to receive a group key excluding the Group Leader is referred to as  $g$ . Any user  $n$  may choose to send a Group Key to  $g$  recipients that have already joined the Smart Contract User List by choosing to become a Group Leader  $k$ .

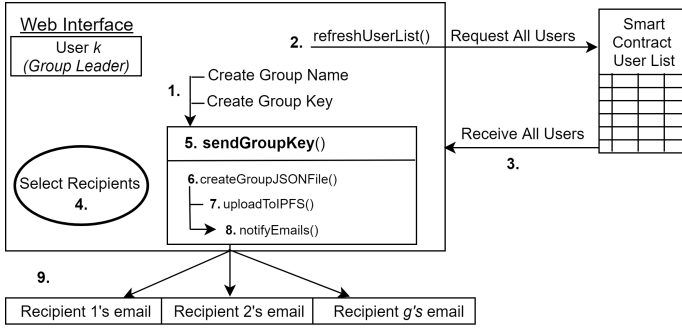


Fig. 4: The group leader performs all the steps required to send an encrypted Group Key to  $g$  recipients.

- 1) The Group Leader creates a *Group Name* and a *Group Key* on the web interface.
- 2) The Group Leader clicks the refresh button that calls a smart contract function to request the Smart Contract User List.
- 3) The web interface receives the Smart Contract User List and populates a selection list with  $n$  Wallet Addresses, Public Keys, and Email Addresses. The selection list excludes the Group Leader  $k$  from visibility so that only  $g$  users can be selected as a recipient.
- 4) The Group Leader  $k$  chooses  $g$  users from the populated selection list to become a recipient for the Group Key and clicks the send button to start the sending process.
- 5) The selected recipients stored in local memory are retrieved by the internal function `sendGroupKey()` which subsequently calls the helper functions `createGroupJSONFile()` in Part #6, `uploadToIPFS()` in Part #7, and `notifyEmails()` in Part #8.
- 6) The helper function `createGroupJSONFile()` creates a JSON file referred to as *GroupJSONFile* that contains the required data to successfully share the encrypted Group Key to  $g$  Recipients. **Figure 5** shows a visual representation of the *GroupJSONFile* structural layout. The fields in **Figure 5** are explained below in detail.

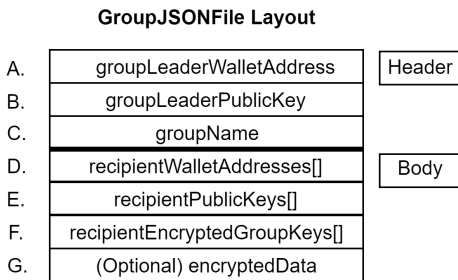


Fig. 5: A visual representation of the structure inside the GroupJSONFile.

- A. The Group Leader's Ethereum Wallet Address.
- B. The Group Leader's Public Key.
- C. The Group Name of the Group Key recipients.
- D. An array of  $g$  amount recipient Wallet Addresses.

- E. An array of  $g$  amount of recipient Public Keys.
  - F. An array of  $g$  amount of recipient Group Keys encrypted by the shared key between the Group Leader's Public Key and the recipient's Public Key.
  - G. The optional data included by the Group Leader that has been encrypted by the Group Key.
- 7) The helper function `uploadToIPFS()` converts the *GroupJSONFile* to a string representation first then buffers and uploads that to IPFS. Upon a successful upload an IPFS hash is returned that will be used to later download the file again.
  - 8) The helper function `notifyEmails()` sends an email containing a web link to a JSON extractor web interface along with the IPFS hash to the GroupJSONFile to  $g$  recipient email addresses. This feature wasn't implemented but was included for demonstration purposes only as it is beyond the scope of our research.
  - 9) Each recipient's email address receives the information sent in Step #8. The *GroupJSONFile* may be manually extracted but an automatic extraction process is optimal for speed. Every  $g$  recipient receives the email, extracts the contents of the *GroupJSONFile*, calculates the shared key with the Group Leader, decrypts their Group Key, and optionally uses the Group Key to decrypt the *encryptedData* if it was included. The Group Key has been successfully sent through the Ethereum private channel to  $g$  recipients thereby concluding Group Key distribution process.

**Figure 6** shows the Concept of Operations (COP) for the overall process with two visual steps. Step 1 shows **Figure 3** from a different perspective and Step 2 shows **Figure 4** from different perspective.

## V. SECURITY ANALYSIS

### A. Key Strength and Complexity

If an attacker wants to create a shared key using the public information  $\lambda$ ,  $i$ ,  $p$  the discrete logarithm must solve for the exponent  $t$  in a finite field  $GF(p)$  with:

$$i = \lambda^t \mod (p)$$

In order to reverse the DH process a discrete logarithm must be solved [5] [16] and a popular method for this is the Number Field Sieve (NFS). The formula for estimating the number of simple arithmetic operations needed to calculate the NFS for a prime number  $n$ :

$$L(n) = k \times e^{((1.92+o(1)) \times \sqrt{\ln(n) \times \ln(\ln(n))^2})}$$

There are three sets of the values  $\lambda$  and  $p$  that have been prechosen by Internet Engineering Task Force (IETF) with optimal computational complexity referred to as RFC 5114 [7] and are currently being used as the standard for a variety of security protocols (e.g. TLS, SSH, IKE, SMIME) [6]. In addition, there are also several more Diffie-Hellman (DH) groups available like RFC 3526 [8]. If the optimal values are chosen for the initial DH process it is too computationally

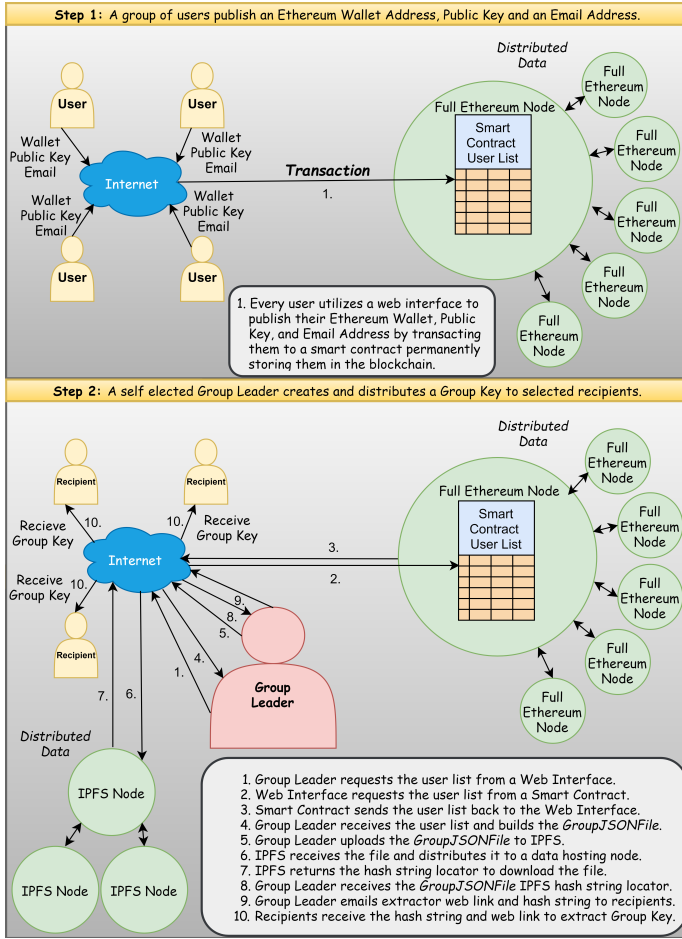


Fig. 6: A visual two step operation for the entire Group Key distribution process.

difficult to solve the equation that reverses the process which is the basis for the security of DH. DH key size is proportional to the computational effort required to reverse the encryption process using the public variables. Larger key sizes result in the improved security of the protocol. **Table I** below shows a comparison of several security protocols with key sizes in bits that are nearly the same in security strength.

TABLE I: Key bit size comparison to other encryption protocols that present the same level of security.

ECC (bits)	AES (bits)	DH/RSA (bits)
160	80	1024
224	112	2048
256	128	3072
384	192	7680
521	256	15360

Diffie-Hellman (DH) and Rivest-Shamir-Adleman (RSA) keys are similar in terms of security. DH requires solving a complicated discrete logarithm while RSA requires factoring large integers and both are similar in terms of the computational effort required to reverse the process therefore they are included in the same category. AES and Elliptic Curve Cryptography (ECC) are in different categories because the creation and reversal process are different. For example, a 128 bit symmetric AES key requires around the same computational effort to reverse as a 3072 bit asymmetric DH key thereby presenting the same level of security. However, the larger the key size the more computational effort required to produce the encryption which can be problematic for encrypting large amounts of data. The current encryption standard for DH is 2048 bits up to the year 2030. According to the National Security Agency (NSA) the advancement of supercomputers will require the new DH standard to become a 3072 bit key size beyond the year 2030 to maintain the security against reversing the process [10].

**B. Attacks on DH**

1) **Man in the Middle (MITM) attack:** A MITM attack occurs when UserA attempts to create a shared key with UserB but actually creates one with an imposter referred to as UserC [13]. UserC acts as a middle man between the two parties by calculating the shared key for everyone and keeping it to decrypt all of the encrypted communications [14]. UserA and UserB don't realize that UserC is involved. The following is an example in which the described scenario takes place:

### B. Attacks on DH

- 1) **Man in the Middle (MITM) attack:** A MITM attack occurs when UserA attempts to create a shared key with UserB but actually creates one with an imposter referred to as UserC [13]. UserC acts as a middle man between the two parties by calculating the shared key for everyone and keeping it to decrypt all of the encrypted communications [14]. UserA and UserB don't realize that UserC is involved. The following is an example in which the described scenario takes place:

$$\begin{aligned}
 & UserA \rightarrow UserC(UserB) : \lambda^{t_A} \mod(p) \\
 & \hookrightarrow UserC(UserA) \rightarrow UserB : \lambda^{t_C} \mod(p) \\
 & \hookrightarrow UserB \rightarrow UserC(UserA) : \lambda^{t_B} \mod(p) \\
 & \hookrightarrow UserC(UserB) \rightarrow UserA : \lambda^{t_C} \mod(p)
 \end{aligned}$$

The reason this attack is possible is because there is no authentication between UserA and UserB [12]. The threat of this attack is greatly minimized due to the distributed immutability of the blockchain.

- 2) **Impersonation Attack:** An impersonation attack occurs when UserA sends a Public Key and ID to UserB and it is intercepted by an imposter referred to as UserC. UserC switches the UserA's Public Key and ID to UserC's Public Key and ID before sending the message to its original recipient UserB. UserA intended to create a shared key with UserB but created one with UserC instead without realizing. Every message that UserA sends to UserB can be decrypted by UserC. The following is an example in which the described scenario takes place:

$$\begin{aligned}
 & UserA \rightarrow UserC(UserB) : \lambda^{t_A} \mod(p) \\
 & \hookrightarrow UserC(UserB) \rightarrow UserA : \lambda^{t_C} \mod(p)
 \end{aligned}$$

There is no threat of this attack due to the distributed immutability of the blockchain. When a user joins the smart contract they are expected to ensure that their information has successfully been included a block of data on the longest chain.



### C. Attacks on Smart Contracts

- 1) **Spam Attacks:** A spam attack occurs when a user creates faulty transactions with the objective of slowing down the mining operation of the decentralized blockchain. This can result in large transaction time delays, the loss of gas money, and wasted mining computation power. This could also decrease the number of reachable full nodes and in a worst case scenario result in a complete stop to new transactions [15].
- 2) **Malicious Contracts:** A malicious contract attack can occur by presenting itself as the authentic contract in order to deceive a user into a MITM attack. Another malicious contract attack can occur because smart contracts cannot handle code errors. This may be exploited by a knowledgeable attacker to postpone or create inaccurate output results for smart contract transactions. In addition, it could also result in a user losing all of their gas funds. This problem is preventable with the proper smart contract coding practices [14].

## VI. TESTBED IMPLEMENTATION

### A. Components and Interactions among them

The sequence diagram of the actions of a user are illustrated below in **Figure 7**.

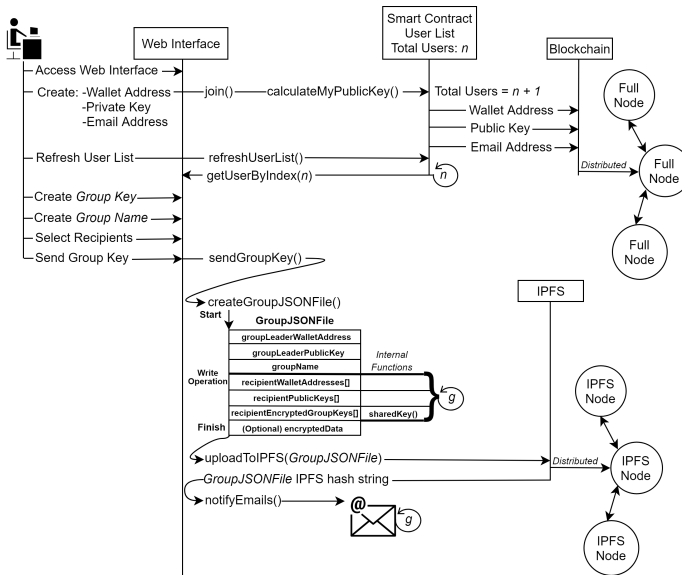


Fig. 7: A visual two step operation for the entire Group Key distribution process.

### B. Group File Creation

**Figure 8** shows a web interface we created in order to demonstrate the functionality of our Group Key Encryption Protocol creation and distribution.

- Select a data file to be encrypted by the Group Key.
- Create a Private Key or automatically generate one.
- Join the Smart Contract User List with a Public Key and a Notification Email Address.
- Retrieve the Smart Contract User List.

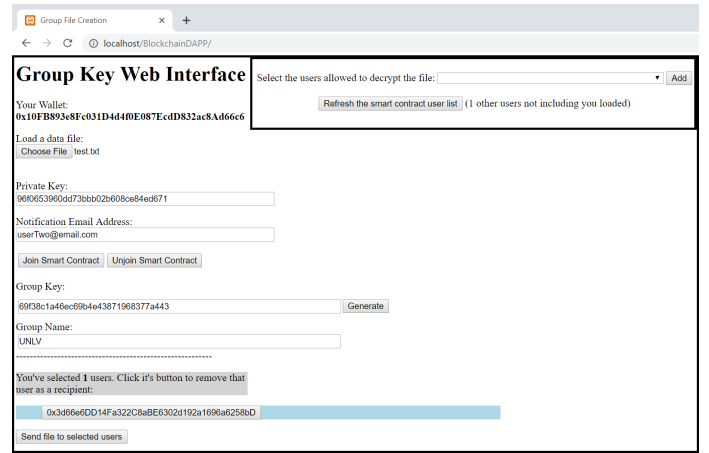


Fig. 8: The web interface used for a Group Leader to create and distribute a Group Key to users in the Smart Contract User List.

- Create a Group Key or automatically generate one.
- Securely send a Group Key and/or encrypted data to selected recipients.

### C. Group File Extractor

**Figure 9** shows the web interface created to illustrate one possible way that the *GroupJSONFile* may be extracted when it is received through email.

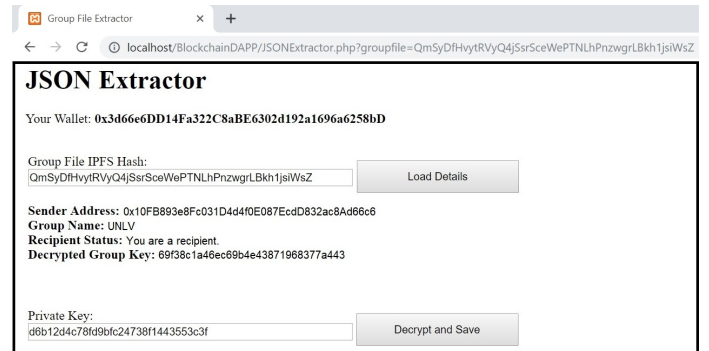


Fig. 9: The web interface for extracting the *GroupJSONFile* is included with the IPFS hash emailed by the Group Leader to the recipients.

### D. Code Segments

**Listing 1** is the function called when a user clicks the button "Send file to selected users" which subsequently calls **Listing 2** that builds the *GroupJSONFile* and **Listing 3** that uploads it to IPFS and captures the hash string locator. **Figure 10** measures the performance of these three functions being used together in different scenarios. It should be noted that the performance only measures the *GroupJSONFile* creation and not the amount of time it takes for the email to arrive at the recipient.

```
function sendGroupKey() {
  var GroupJSONFile = createGroupJSONFile(); //
  Creates GroupJSONFile
  uploadToIPFS(JSON.stringify(GroupJSONFile)); //
  Uploads to IPFS
  notifyEmails(); // Emails GroupJSONFile hash string
  to selected recipients. Implementation is
  beyond the scope of our research.
}
```

Listing 1: Function: sendGroupKey()

```
function createGroupJSONFile() { // Creates the
  container for holding all of the encryption data
  var Recipients = selectedUserList; // Get
  Recipients from the selectedUserList in the web
  interface
  var myPrivateKey = $("#privateKey").val(); // Get
  the Leader's Public Key from the web interface
  var GroupName = $("#groupName").val(); // Get the
  Group Name from the web interface
  var GroupKey = $("#groupKey").val(); // Get the
  Group Key from the web interface
  var recipientPublicKeys = []; // Create an empty
  array to hold all of the recipient Public Keys
  var sharedKeys = []; // Empty array to hold the
  shared keys
  var GroupJSONFile = new Object(); // Create the
  GroupJSONFile Object
  GroupJSONFile.groupLeaderWalletAddress =
  currentWallet; // Add the Leader's currently
  selected MetaMask wallet to Leader.
  walletAddress
  GroupJSONFile.groupLeaderPublicKey = myPublicKey;
  // Add the Leader's current Public Key to
  Leader.publicKey
  GroupJSONFile.groupName = GroupName; // Add the
  created Group Name to GroupName
  GroupJSONFile.recipientWalletAddresses = []; // A
  blank array for Recipient.walletAddresses
  GroupJSONFile.recipientPublicKeys = []; // A blank
  array for the Recipient Public Keys
  GroupJSONFile.recipientEncryptedGroupKeys = []; //
  A blank array for each Group Key encrypted by
  Recipient shared key
  if (fileBuffer) {
  var encryptedData = CryptoJS.AES.encrypt(fileBuffer
  , GroupKey); // (Optional) Encrypt the Data
  File if it was included on the Group Control
  Panel
  GroupJSONFile.encryptedData = encryptedData.
  toString(); // Convert the data to string and
  add to encryptedData
  }
  Recipients.forEach(function(user) { // Iterate
  through each Recipient
  var recipientWalletAddress = user.get("Address");
  // Get the Recipient Wallet Address
  if (recipientWalletAddress != currentWallet) { //
  Skip your Wallet Address
  GroupJSONFile.recipientWalletAddresses.push(
  recipientWalletAddress); // Add user Wallet
  Address to recipientWalletAddresses array
  var recipientPublicKey = bigInt(user.get("
  PublicKey")); // Get the Recipient Public Key
  GroupJSONFile.recipientPublicKeys.push(
  recipientPublicKey); // Add user Wallet
  Address to recipientWalletAddresses array
  SharedKey = bigInt(sharedKey(myPrivateKey,
  recipientPublicKey)); // Create a Shared Key
  between the Private Key and Recipient Public
  Key
  encryptedGroupKey = encrypt(GroupKey, SharedKey.
  toString()); // Encrypt the Group Key with
```

```
that Shared Key
  GroupJSONFile.recipientEncryptedGroupKeys.push(
  encryptedGroupKey); // Store the Recipient
  Encrypted Group Key in
  recipientEncryptedGroupKeys
  });
  return GroupJSONFile; // The GroupJSONFile has been
  built. Return the file.
}
```

Listing 2: Function: createGroupJSONFile()

```
function uploadToIPFS(data) {
  window.ipfs.add(ipfs.types.Buffer.from(data),
  function(err, result) { // Convert the data to
  a Buffer and then add to IPFS
  if (result) { // Successful upload
  var fileHash = result[0].hash; // Retrieve IPFS
  hash from the upload
  sent to each Recipient
  } else if (err) { // File had an error uploading
  console.log("Error reading file. Error: " + err);
  } else { // File didn't upload at all
  console.log("IPFS hash retrieval unsuccessful.")
  });
  return fileHash;
}
```

Listing 3: Function: uploadToIPFS()

## VII. PERFORMANCE ANALYSIS

Group Key Encryption Protocols are varied when it comes to Big O notation. Our algorithm maintains a competitive performance at  $O(n)$  to prepare a GroupKey to be sent in email to selected recipients. In **Figure 10** below the performance test measures how long it takes to create the *GroupJSONFile* with different amounts of recipients, key sizes, and the time required to upload the *GroupJSONFile* to IPFS and retrieve the hash string. The performance will be drastically increased with stronger computer performance specifications and optimized algorithms.

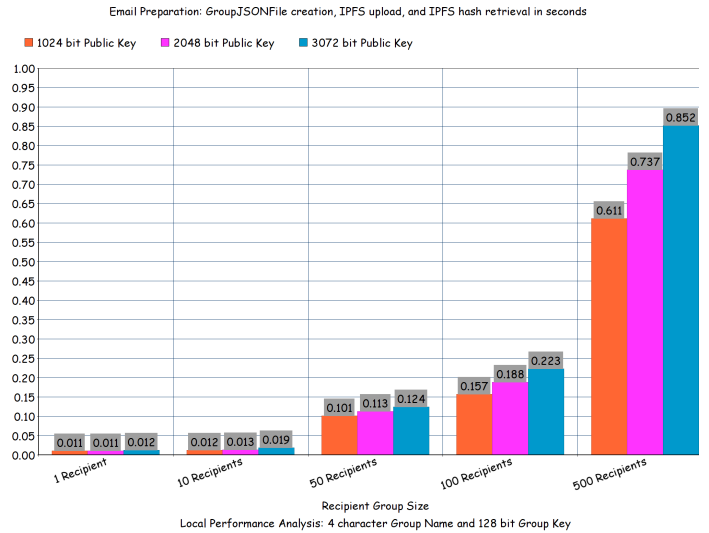


Fig. 10: Alternative Group Key Encryption protocol communication and computation costs.

## VIII. DISCUSSIONS

### A. Suggested Usage

**Main Ethereum Blockchain:** The best option for the discovery of Public Keys with the highest transaction speed and data immutability. However, it costs real money therefore the smart contract and web interface need to be optimized for gas usage and smart contract code vulnerabilities. Any information that is published on this blockchain like a Public Key or a smart contract will exist forever.

**Ropsten Blockchain:** The best option for our prototype because of the open blockchain access with free transactions. The Ropsten is still immutable as there are many data nodes but the data is wiped every year. This option allows our prototype to be implemented with the help of Infura and MetaMask in real applications.

**Private Blockchain:** The best option for further development, access control, unlimited gas, and low transaction time. However, this option exchanges security for more efficiency as it suffers from the lack of data immutability and Public Key visibility.

### B. Elliptic Curve Cryptography (ECC)

Smart Contracts are limited by the high cost of storing information on the blockchain [11]. However, Elliptic Curve Cryptography (ECC) offers the same level of security as DH with a smaller Public Key size visually explained in **Table 1**. If ECC was implemented it would increase algorithm performance time as well as a lower the cost for transactions. We recommend ECC instead of DH as it offers many benefits over DH. ECC also has several JavaScript browser implementations available that would be simple to integrate into any web interface.

## IX. FUTURE WORK AND CONCLUSIONS

Our Group Key Encryption Protocol has many areas that can be improved with additional research. Alternative Group Key Encryption Protocols currently have the ability to dynamically add or remove users but ours does not. However, the *GroupJSONFile* architecture allows additional layers of information to be added to enable this feature in the future. While the immutability of DLT means little authentication is required to prevent MITM and impersonation attacks contrary to centralized alternatives, new layers of information may be added in future research to further increase security by adding more authentication procedures. Alternative Group Key Encryption Protocols are built upon a centralized architecture as a trusted third party for exchanging of Public Keys. Centralized architectures trade security for better performance and as a result are vulnerable to MITM attacks, DDoS attacks, and service outage. The alternatives also require a persistent connection and active participation on behalf of other participants which can create unfavorable conditions if the connection is suddenly terminated. Our group key encryption protocol uses the decentralized open architecture as the trusted third party which trades performance for better security and transparency. It is non-interactive because our

protocol only requires access to information that already exists on the open permanent distributed blockchain and therefore only requires a non-persistent connection. The data stored on the blockchain will in theory always have a data node to respond to information requests thereby removing the risk of service outage entirely. According to **Figure 10**, after a group publishes their Public Keys, Wallet Addresses, and Email Addresses to the smart contract a Group Leader may establish private group communication for 500 users in less than 1 second using the 3072 bit Public Key IETF standard. At the time of writing this paper no other blockchain solution existed for this specific problem. Therefore, we conclude that our Group Encryption Protocol performs better than alternative protocols for a variety of use case scenarios and is a viable solution for establishing private group communication on the open distributed blockchain.

## REFERENCES

- [1] Shangping Wang, Yinglong Zhang, Yaling Zhang. "A Blockchain-Based Framework for Data Sharing With Fine-Grained Access Control in Decentralized Storage Systems" IEEE, 2018
- [2] Lavanya R, Dr. S V Sathyanarayana. "Group Diffie Hellman key Exchange Algorithm Based Secure Group Communication", International Conference on Applied and Theoretical Computing and Communication Technology, IEEE 2017
- [3] Yang Guang-ming, Lu Ya-feng, MA Da-ming. "An Efficient Improved Group Key Agreement Protocol Based on Diffie-Hellman Key Exchange." IEEE, 2017
- [4] Ian F. Blake and Theo Garefalakis, "On the complexity of the Discrete Logarithm and Diffie-Hellman problems", Journal of Complexity 20 (2004), 148–170
- [5] H. Orman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys," IETF, April 2004. [Online]. Available: <https://tools.ietf.org/html/rfc3766>. [Accessed 24 March 2019].
- [6] K. Suganya, K. Ramya. "Performance study on Diffie Hellman Key Exchange Algorithm." International Journal for Research in Applied Science and Engineering Technology (IJRASET), Vol. 2 Issue III, March 2014.
- [7] Lepinski, M. and S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards", RFC 5114, pages 3-5. January 2008.
- [8] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", RFC 3526, pages 3-7. May 2003.
- [9] Patsonakis, Chris & Samari, Katerina & Roussopoulos, Mema & Kiyayas, Aggelos. (2018). "Towards a Smart Contract-Based, Decentralized, Public-Key Infrastructure", 16th International Conference, CANS 2017, Hong Kong, China, November 30, December 2, 2017.
- [10] Commercial National Security Algorithm Suite and Quantum Computing FAQ U.S. National Security Agency, January 2016
- [11] T. Chen, X. Li, X. Luo and X. Zhang, "Under-optimized smart contracts devour your money," 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, 2017, pp. 442-446.
- [12] Steven Galbraith and Victor Rotger, Easy decision Diffie-Hellman groups, LMS Journal of Computation and Mathematics 7 (2004), 201-218.
- [13] Nan Li, "Research on Diffie-Hellman key exchange protocol," 2010 2nd International Conference on Computer Engineering and Technology, Chengdu, 2010, pp. V4-634-V4-637.
- [14] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in International Conference on Principles of Security and Trust. Springer, 2017, pp. 164-186.
- [15] L. Parker, "Bitcoin Under Attack!," securitycommunity.tcs.com, 4 November 2017. [Online]. Available: <https://securitycommunity.tcs.com/infosecsoapbox/articles/2017/11/04/bitcoin-under-attack>. [Accessed 20 March 2019].
- [16] D. Gordon, "Discrete logarithms in GF(p) using the number field sieve," SIAM Journal on Discrete Mathematics, pp. 124-138, 1993.