CryptoChat: Implementing Secure Group Communication

Nya Haseley-Ayende

Brown University

nya_haseley-ayende@brown.edu

Amy Wang

Brown University

yanchi_wang@brown.edu

ACM Reference Format:

Nya Haseley-Ayende and Amy Wang. 2024. CryptoChat: Implementing Secure Group Communication. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Motivation

In contemporary digital discourse, group chats have emerged as an integral component of communication paradigms. These platforms are commonly used in both professional and personal settings, allowing people to collaborate, coordinate, and connect. However, despite their convenience, worries about data security are significant.

Consider the prevalence of major messaging applications like WhatsApp, which facilitate group chats on a massive scale. Despite assurances of encryption, the underlying anxiety persists: to what extent can servers access and potentially exploit user conversations? This situation highlights the need for secure communication channels where users can interact without compromising their privacy.

Our project thus endeavors to address this exigency by developing a messaging protocol tailored to support group chats while obfuscating the group's structural dynamics from server surveillance. By doing so, we aim to furnish users with a platform wherein they can converse with confidence, assured that their exchanges remain shielded from prying eyes. In effect, our endeavor stands as a pivotal stride towards fortifying digital privacy in an era fraught with data vulnerabilities.

1.2 Project Statement

Our final project implements a messaging protocol supporting **three**-person group chats, providing users with a platform where they can chat confidently, knowing that their conversations are protected from unwanted access and hiding the group structure from the server. This effort represents an important step towards enhancing digital privacy in an age where data security is increasingly precarious.

Conference'17, July 2017, Washington, DC, USA

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

2 PROJECT OVERVIEW

2.1 Program Flow Summary

This project will seek to modify the existing Auth project in CS1515. All users will still register and login to initialize communication with the server, almost exactly like we did in Auth previously. The server will save each of the client IDs locally in the database as they generate certificates upon registration.

However, upon all three users' request for a group chat, the server will initiate communication between the users to generate a shared group chat key amongst all parties currently connecting with them. Then, the server is constantly listening for messages from any user with a given index in their local variable for the list of currently active threads. Upon receiving any message, the server will send that messages to everyone except the user with that corresponding index. The users are constantly sending messages that are encrypted with their shared group chat key, allowing hidden messages from the Server and private communication between the users.

2.2 Generating a Shared Group Chat Key

To generate a shared secret key (R) among those 3 group chat members but hidden from the server, we followed the following protocol.

For Users:

Similar to auth, the 3 users will agree on some generator g and each create their local private keys a, b, c and public keys g^a, g^b, g^c.

(1) Each user generates their public key and sends it to the server.
   - User A shares their public key g^a
   - User B shares their public key g^b
   - User C shares their public key g^c

(2) They receive all other group chat members' public keys from the server.
   - User A receives public keys g^b, g^c
   - User B receives public keys g^a, g^c
   - User C receives public keys g^a, g^b

(3) Using their private key, each user generates the next level of public keys.
- User A generates public keys $g^{ab} = (g^b)^a$, $g^{ac} = (g^c)^a$
- User B generates public keys $g^{ab} = (g^a)^b$, $g^{bc} = (g^c)^b$
- User C generates public keys $g^{ac} = (g^a)^c$, $g^{bc} = (g^b)^c$

For Admin

(4) They generate a random integer sampled from 2^q, denoted as R.

(5) Using the next level of public keys, they encrypt R and send it to the server.

For Non-Admin User

(6) They receive encrypted messages containing R from the server and filter out for the ones involving them.
- User A receives $g_e^{ab}(R), g_e^{bc}(R), g_e^{ac}(R)$ and $g_e^{bc}(R)$. They keep $g_e^{ab}(R)$ and $g_e^{ac}(R)$.
- User B receives $g_e^{ab}(R), g_e^{ac}(R), g_e^{ac}(R)$ and $g_e^{bc}(R)$. They keep $g_e^{ab}(R)$ and $g_e^{bc}(R)$.
- User C receives $g_e^{ab}(R)$, $g_e^{ac}(R)$, $g_e^{ab}(R)$ and $g_e^{bc}(R)$. They keep $g_e^{ac}(R)$ and $g_e^{bc}(R)$.

  Note. e denotes an encryption of the key.

(7) Using their respective next level of public keys, they decrypt the messages to reveal R.

For the Server:

(1) The server receives public keys from users and distributes them as necessary.
- Server receives public key g^a from User A and sends it to User B and User C.
- Server receives public key g^b from User B and sends it to User A and User C.
- Server receives public key g^c from User C and sends it to User A and User B.

(2) If communicating with the admin, the server facilitates the exchange of encrypted R messages among users.
- Server receives $g_e^{ab}(R)$ and $g_e^{ac}(R)$ from User A and sends it to User B and User C.
- Server receives $g_e^{ab}(R)$ and $g_e^{bc}(R)$ from User B and sends it to User A and User C.
- Server receives $g_e^{ac}(R)$ and $g_e^{bc}(R)$ from User C and sends it to User A and User B.

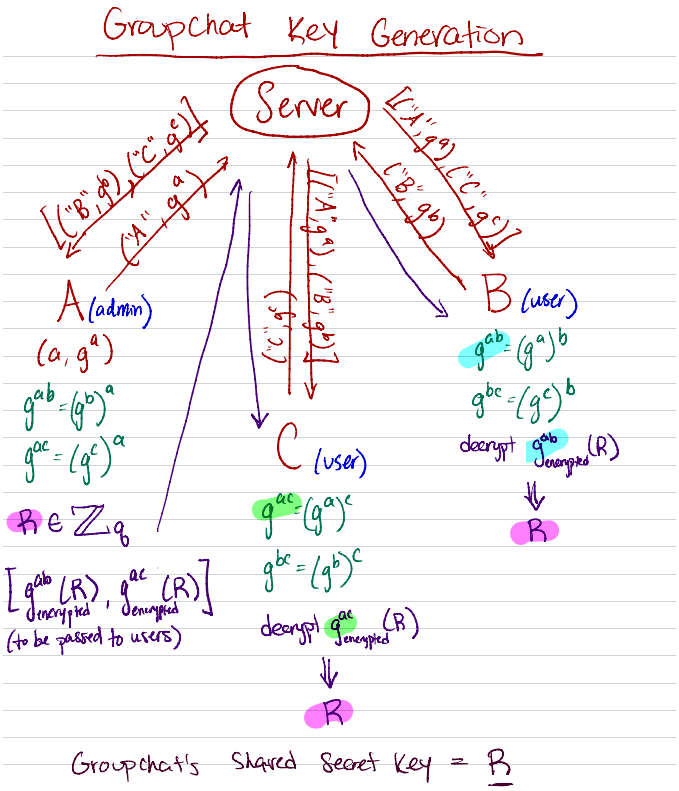
  Note. e denotes an encryption of the key.

Once the secret key R is shared and agreed upon, the user will simply send the message through the server, which will take the encrypted message and send that encrypted message to each of them along the three threads currently open, upon which each user will decrypt the message.

2.3 Process

Our program creates a groupchat for three people. The user will connect to the server in the same way as in Auth, input the groupchat command, and input three parameters. The three parameters should be the address, the port (both of which should be the same as the server), and last is "admin" or "user" depending on whether the user is an admin or not.

Upon connecting to the group chat server, the users and server will wait until there are at least three users connected to the server, one of which must be an admin. The server will maintain a list of these threads, one for each user, check its size and make sure there are three people connected. If more than one user attempts to log in as an admin, they will receive an error and the server will terminate the process. If there are no admins, the results will be the same as above. After, the users will generate a shared group chat key as described in section 2.2. When all users have successfully generated a shared key, any messages they send to the server will be broadcasted and sent to all other users in the group chat. To ensure security, the users must encrypt their messages with the secret key before sending them to the server.



2.4 Code

Our group chat functionality can be broken down into these specific methods. In addition to these functions, we also programmed our

own messages in `messages.cxx` for the data exchanged between the users, admin, and server in the key generation process, as well as their serializations and deserializations.

- `ServerClient::HandleGCConnection`
- `ServerClient::GenerateGCKey`
- `ServerClient::SendToAll`
- `ServerClient::GetKeys`
- `UserClient::HandleGCMessage`
- `UserClient::DoMessageGC`
- `UserClient::GenerateGCKey`

Every time the server is ran, it instantiates a private instance variable vector `threads`. While listening for connections and creating new threads, every time a new user connects, before it detaches the thread to run the `HandleGCConnection`, it saves the network driver of that specific connection in `threads`. Furthermore, `HandleGCConnection` takes in the network and crypto drivers, but also the index at which it is added to the `threads`. Thus, it passes this index as an input into `SendToAll` alongside the message it wants to send, and within `SendToAll`, the program loops through each of the network drivers bar the one at the given index to send to message.

```
void ServerClient::SendToAll(
    std::vector<unsigned char> data,
    int index){
  int thread_size = this->threads.size();
  for (int i = 0; i < thread_size; i++){
      if (i == index) continue;
      this->threads[i]->send(data);
    }
}
```

When three people connect to the server, the server calls `GetKeys` to get the public keys of each user and store them accordingly in a vector. The server then waits for all the keys to be populated, also noting the index of the thread for the user that is the admin. Once all the keys are received, the server finally calls `GenerateGCKey` to begin the server-side calls to generate the secret group chat key as noted in section 2.2.

After the key is generated, the server continuously listens to read messages from the users and send them to the other users using `SendToAll`.

On the user-side, after entering the corresponding repl command to enter the group chat function, `HandleGCMessage`, the computer parses the user's input, noting if the user is an admin or not, and then performs `DoMessageGC`. Within `DoMessageGC`, the user performs the key exchange to generate the secret group chat key.

Instead of using a DH-Ratchet, we modified the following code in Auth's `HandleUser` function wherein the code allows the user to freely send and read messages freely, and used it to set up the group chat functionality.

```
boost::thread msgListener =
  boost::thread(boost::bind(
    &UserClient::ReceiveGCThread,
    this,
    gc_keys));
this->SendGCThread(gc_keys);
msgListener.join();
```

Unlike in `HandleUser`, the user sends the messages to the server using the shared group chat key to encrypt the messages. Since the server does not know this key, it simply passes this message to the other users, who use their shared key to decrypt and read the message. From the user's perspectives, this visually does present like a group chat despite having only communicated directly with the server.

3 TESTING

The majority of our testing was completed locally by utilizing print statements for fields being passed through messages, checking lengths of fields for shared key vectors, and ensuring that the private shared group chat key generated by the users matched that origin generated by the admin.

We also made sure that the messages the server received for when the users are using the group chat to communicate with each other were not decryptable by the server by printing them out. As expected they were seemingly random. Using `decrypt_and_tag` would fail to reveal the original plaintext. Thus we made sure our code was secure against a malicious server and potential eavesdroppers. We did not design security around malicious users, as this protocol is open to allow all users to join.

4 CHALLENGES FACED

During the implementation of this project, we faced a couple challenges. Our biggest challenge was setting up a basic messaging scheme from one person to all other users in the group chat, utilizing the server as an intermediary. This was because it required some background on working with multi-thread programming and maintaining an array with all the threads so we know which users to send the corresponding messages to. In our original proposal, we were created an architecture design that essentially bypassed the role of the server entirely, only sending messages directly from one user to another. However, this was incredibly difficult as we would need a way to communicate from a given thread to another on the user side. We underestimated the complexity of this step, so we would have benefited from more thorough planning in the first week of the project to figure out how exactly we would send

messages to certain users in both the key generation and communication components. However, we were able to overcome this challenge by creating a basic `SendToAll` helper function that transmits messages to all threads managed by the Server, except for the thread identified by a specific index provided as input. This helper was then utilized in the `HandleGCConnection` and `GenerateGCKey` functions.

Furthermore, we struggled with the design of a three-way shared key generation algorithm. We originally planned to have the users rotate the key generation. We would have Alice sending her public key to Bob, Bob sending his public key to Charlie, and Charlie sending their public key to Alice. Then, they would raise the value they received to the power of their secret key, and send that result again to the person they previously sent to. At the end, everyone would have a shared key in the form of g^{abc}. However, this proved exceptionally difficult as the users would have to set up multiple channels with their sender, receiver, and the server. Even if we had the server handle all the interactions between the users, the complexity of reading and sending messages over and over increased drastically. In the end, we were recommended to look into pairwise shared key generation, which we liked and implemented in our project.

We also struggled to make sense of how the users would be remembered and kept track of within the server. Before settling on saving the network drivers within a simple vector, we wanted to set up a dictionary system of a vector of points so that we could also save other details of the thread, such as the server's keys with them, the crypto driver, and maybe even the id of the user. However, due to multiple threads trying to modify and update their fields on one shared resource, we ran into deadlock issues and ended up ditching that idea.

5 FUTURE GOALS

In addition to accommodating group chats with three participants, our project aims to extend its functionality to support larger groups with an arbitrary number of users. This expansion entails ensuring scalability and efficiency in handling communication among numerous members.

Furthermore, we intend to incorporate a private messaging feature within the group chat environment, affording individual members the capability to engage in discreet conversations akin to the initial Authentication implementation.

Moreover, we aspire to implement a seamless mechanism for adding new members to an existing group chat session, facilitating integration for individuals joining the same localhost port post-establishment. This could be relatively easy from the user side, as all that needs to be done is for the admin and the new user to establish a secure channel and a shared key, and then the admin can encrypt the group chat key with their shared key and send it over without leaking it to the server. However, for the server, it needs to be able to differentiate between existing and new users, as well as having the ability to receive a message in one thread and accurately find the target thread to send the message to.

Another feature we would want to add is the ability for the admin, when creating the group chat, be able to pick among the active connections and choose specific users to be added to the group chat. This would work well with Auth's user and id structure. However, this would require a lot of communication between threads and proves to be a larger network engineering problem.

6 ACKNOWLEDGEMENTS

There are no external libraries that we have used throughout this project. We offer immense gratitude to Brown University's CS1515 Auth Project and the course TA staff for their guidance and for creating the foundations for this project's development throughout the past couple of week. We would like to give special thanks to our project TA Nishchay Parashar, who took the time out of his day to meet with us and help us with the pairwise key-generation.