Robust Distributed Symmetric-key Encryption

Group: Alex Castro, Steve Chang, Garrett Christian, Rachel Litscher

Project Github url:
https://github.com/castroaj/CPP_ROBUST_DISTRIBUTED_SYMMETRIC-KEY_ENCRYPTION

Introduction

Our group has been working on analysing the communication pattern and attempting to find parallel improvements for Professor Wang's paper, *"robust distributed symmetric-key encryption scheme"*. We've created a proof of concept implementation of the scheme that allowed us to do performance benchmarking, find parallel improvements in selected parts of the scheme, and containerize the application to be run on different types of machines.

1 Background

Distributed symmetric-key encryption (DiSE) is an encryption/decryption scheme where a set of N distributed servers each share a partial set of keys, where any T threshold number of servers ($T \leq N$ servers) can use their keys to encrypt or decrypt a message. None of the distributed servers hold all the keys so no one server alone can reconstruct the entire secret key. In the original DiSE scheme, if you encrypt a message where one of the servers is corrupt and provides bad data, there is no way to know until you attempt to decrypt the message. Professor Wang's scheme is unique in that it provides robustness checking between the results returned from the partial keys so that you are made aware of a corrupt transaction earlier than in the usual DiSE scheme. There were two cases we examined for the project: a small case N 5 T 3 and a larger case N 24 T 16.

1.1 The Technology Stack

The demo we created is a QT application written in C++. It leverages OpenSSL Library for our cryptographic needs, the QT framework for networking and other utility classes, OpenMP and C++ threads for parallelization, and Docker to containerize and port the application to different environments. We used Ubuntu 20.04 virtual machines for development. We were also able to run the application on virtual machines with different operating systems, and on Google Cloud using Ubuntu 20.04.

1.2 Role Break Down

There are five roles in Professor Wang's scheme. The Dealer who supplies the omega table and partial keys to the DiSE Servers. The Client who requests either a decryption or encryption transaction. A DiSE Server who waits to be contacted by either the Dealer Client or Honest Initiator. The Honest Initiator, who is a random DiSE

Server chosen by the client to manage the transaction. Finally, the Participant who is contacted by the Honest Initiator to create the partial w's.

1.3 Dealer Scheme

Before transactions can occur the Dealer must be run to give each server their set of partial keys and the omega table. Our Dealer takes input information from a configuration file: a set of server addresses, which size scheme we're utilizing N 5 T 3 or N 24 T 16, as well as thread count for OpenMP parallelization. The Dealer will generate an omega table which is a map of each server id to the keys IDs that they will hold. The omega table has the property that between every T selected servers, they will have at least two redundant copies of each key. This property is required for the robust check between servers where each encrypted result is checked against another server's encrypted result to ensure that they are the same. After the omega table is generated the keys are created using OpenSSL's RAND_BYTES function. Once both the keys and omega table are generated, the Dealer will contact each DiSE Server and send them the full omega table as well as their set of partial keys buffered over QTcpSockets. This communication occurs within an omp parallel for loop to speed up the distribution. The servers save the omega table and partial keys in their environment for usage later.

Server	Assigned Keys					
A	k_1	k_2	k_3	k_4	k_5	k_6
B	k_1	k_2	k_3	k_7	k_8	k_9
C	k_1	k_4	k_5	k_7	k_8	k_{10}
D	k_2	k_4	k_6	k_7	k_9	k_{10}
E	k_3	k_5	k_6	k_8	k_9	k_{10}

Figure 1: Example of N 5 T 3 Omega Table with redundant partial keys color coded

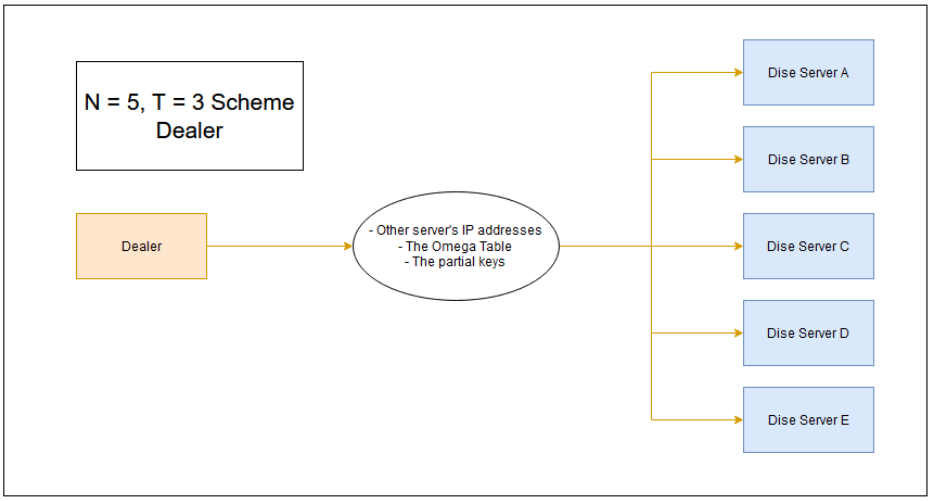

Figure 2: The Dealers Communication Pattern in N 5 T 3

1.4 Encryption Scheme

In an encryption transaction a client will send a message *M* to a randomly selected DiSE Server. The configuration file provides the possible server addresses to randomly choose from. The client will randomly select a server and that server will act as the Honest Initiator for the duration of this transaction. The client will send this server the encryption flag and the message to encrypt then wait for a response. The Honest Initiator upon receiving this information will create: *P* a long random number, *J* the Honest Initiator machine number, (*M*||*P*) message concatenated with *P*, *A* the SHA256 hash of (*M*||*P*), and (*A*||*J*) A concatenated with *J*. The Honest Initiator then will randomly select *T* servers as Participant servers in this transaction and decide what keys each will utilize, equally distributing the keys to use amongst the *T* randomly selected servers. The Honest Initiator will spawn *T* threads to each communicate with one of the *T* randomly selected Participant servers. The threads will each be sent (*A*||*J*) to be encrypted and the keys to use for each encryption. The Participant upon receiving this information will encrypt (*A*||*J*) using AES256 encryption function with each of the denoted keys. The Participant will send back each key id with the following encrypted result. The thread will then join the results together into a mutual exclusion protected list checking to see if all the results match. If they don't match the robust flag will be triggered and the transaction will be cancelled. While the threads are communicating with the other selected servers the Honest thread will utilize all the keys it posses to create its own list of partial *w*'s by encrypting (*A*||*J*) with each key. These partial *w*'s are all xor-ed together by the Honest Initiator into one final *w* then the Honest Initiator waits for the other threads to join. Once the threads have been joined the robust flag is checked. If it was not a robust transaction, meaning at least one pair of partial *w*'s returned from servers did not match, the transaction concludes early and the client is notified that a compromised server took part in the encryption. Otherwise all the robust partial *w*'s matched and the encryption continues. The Honest Initiator will xor the partial *w*'s into the final *w,* which is then used as the seed in a pseudo-random number generator, such as HMAC DRBG. Random bytes that are the size of the message and *P* are generated and xor-ed with the message to create the final ciphertext. The successful encryption robust flag, the final ciphertext, *A*, and *J* are all returned to the client. The client saves this information to a file and concludes the transaction.

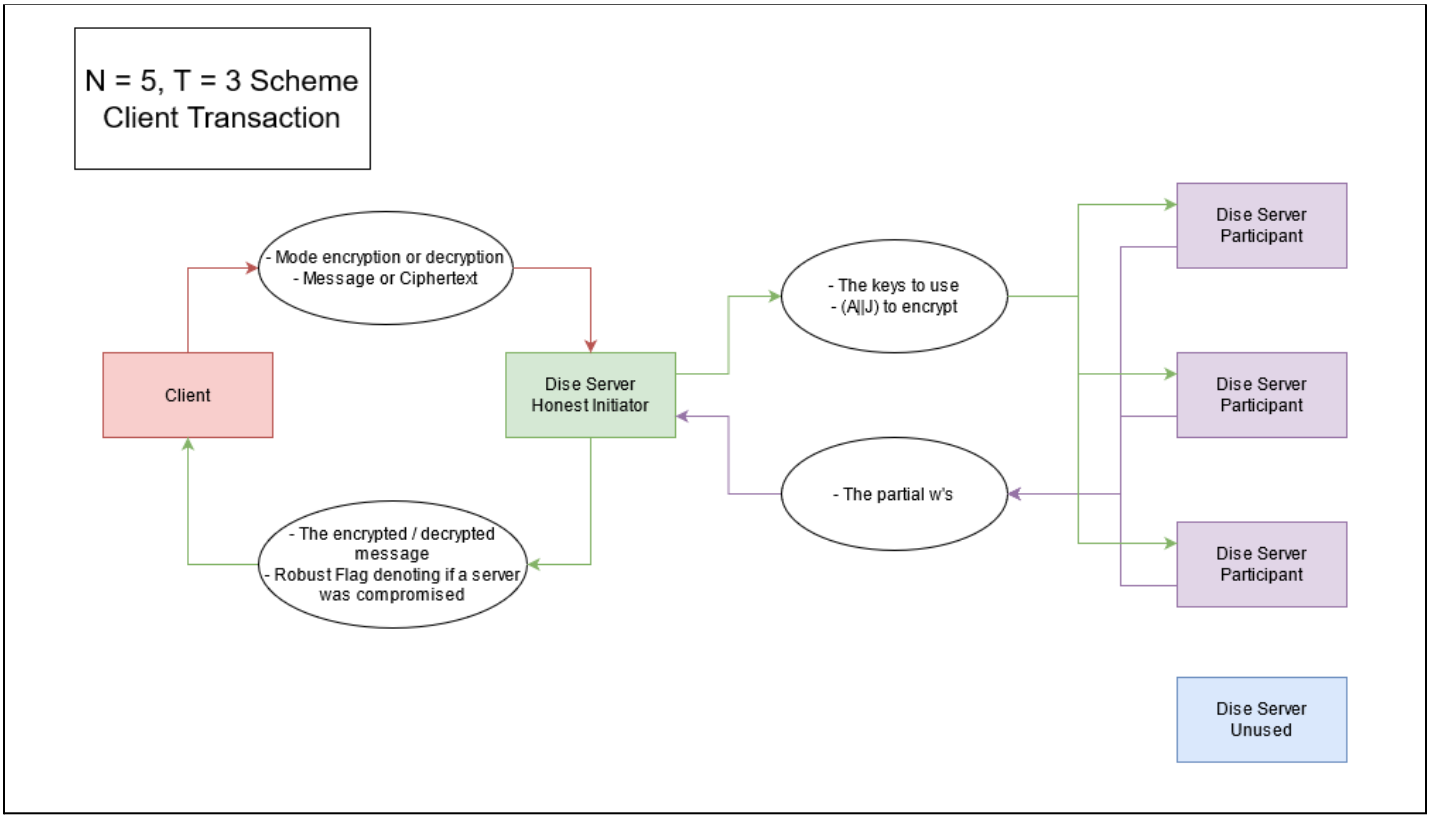

Figure 3: The Client Server High Level Communication Pattern in *N 5 T 3*

Server	*Role*	*Results Calculated*									
A	**Unused**	-	-	-	-	-	-	-	-	-	-
B	**Honest Initiator**	w_1	w_2	w_3	-	-	-	w_7	w_8	w_9	w_{10}
C	**Participating**	-	-	-	w_4	w_5	-	-	-	-	-
D	**Participating**	-	-	-	w_4	-	w_6	-	-	-	-
E	**Participating**	-	-	-	-	w_5	w_6	-	-	-	-

Figure 4: Example of possible *N 5 T 3* partial results

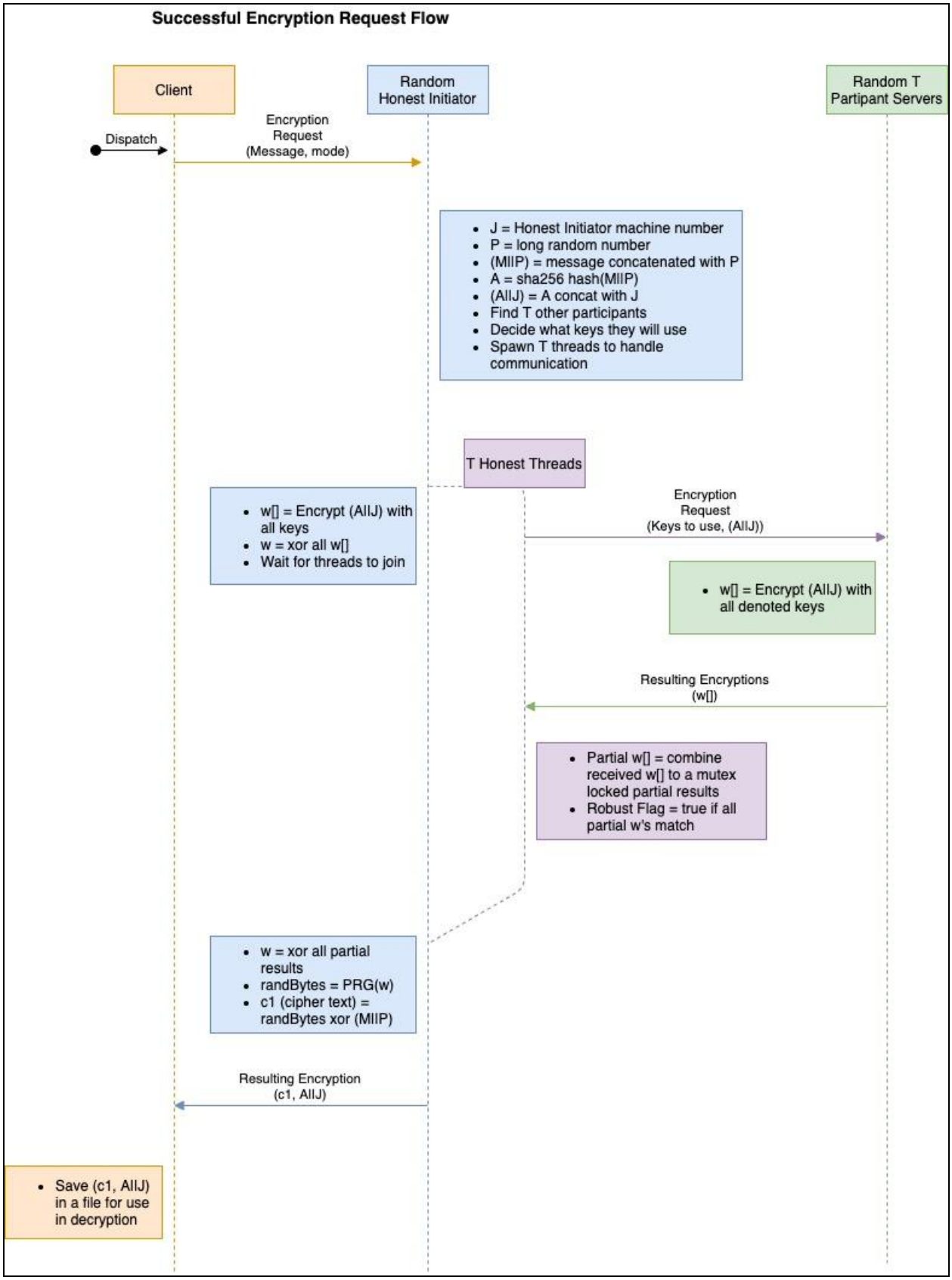

Figure 5: Example of encryption request flow

1.5 Decryption Scheme

In a decryption transaction a client will read cipher text and $(A||J)$ from a file then randomly select a DiSE server to act as the Honest Initiator for the transaction. The client will send that a decryption should be done, the ciphertext, and $(A||J)$ to the randomly chosen Honest Initiator. As in encryption, T Participant servers will be selected, T threads spawned, and the final w is generated with a robustness check performed. If the robustness check passes the resulting w is passed into the pseudo-random number generator where ciphertext-size bytes are generated and xor-ed with the ciphertext. The resulting text $(M||P)$ are passed into the SHA256 hash function to generate $A2$. If the new hash $A2$ doesn't match the original hash A then the transaction is rejected. If it does match, the decrypted message is returned to the client.

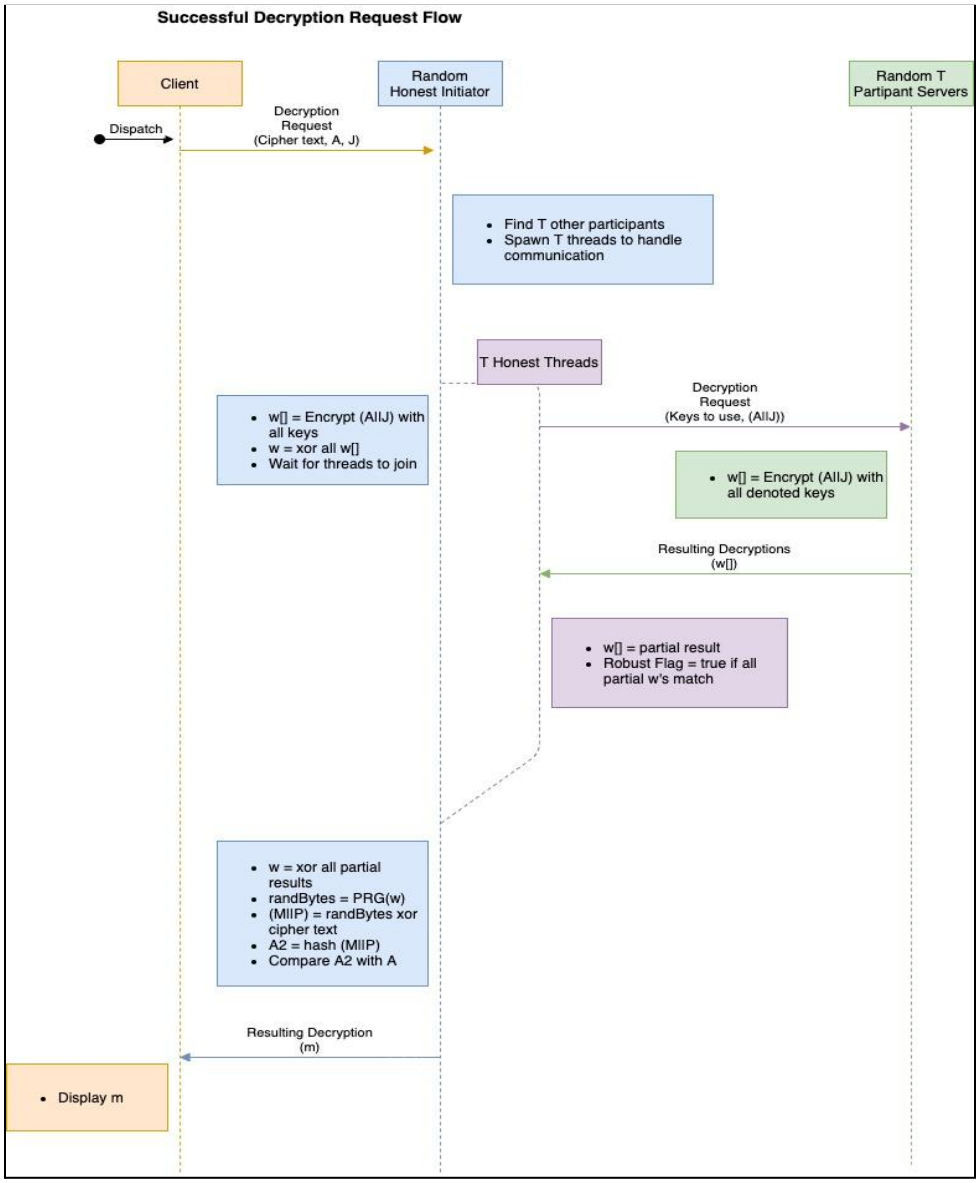

Figure 6: Example of decryption request flow

2 Methods

We began by creating a skeleton for the Dealer and the DiSE Server that could take input from both the command line and from configuration files. We got the Dealer generating the omega matrix and keys for both the small case of *N 5 T 3* and the larger case of *N 24 T 16*. While this development was occurring we were also doing research into how to use Google Cloud, Docker, and what framework to utilize for our networking.

After the mid project deliverable, we entered a period of rapid development. We connected the Dealer and DiSE Server using QTcpSockets, sending all necessary information and saving to an environment object. An issue we encountered from this communication interaction was the sheer size of the message in the larger *N 24 T 16* case. With 24 servers, the size of the omega table is 94,140,480 bytes and each server receives 490,314 keys at 32 bytes per key, for a total of 15,690,048 bytes for the keys. We solved this issue by buffering our reads and writes between servers, something we ended up repeating a lot since the messages for the large case are all extremely large. We were able to find parallel improvement here by using OpenMP on the for loop to connect to the distributed DiSE Servers.

Once we successfully implemented the dealing process, we replicated the interaction in the Client. We changed the parameters to fit the new class and switched on the message type in the DiSE server. This initial communication was relatively simple since we were only sending dummy data, however it began to get difficult working the actual scheme. We started by creating a function to randomly select *T* random participant servers out of the *N* total servers. Once we had that working we created a function to evenly distribute the keys to use among the *T* selected servers. This was a difficult function to implement since it required reworking many of our stored environment variables at the foundation of our project to different data types such as the omega table to a map of servers id's to a key id set instead of a key id list.

Once we had that function working correctly we started working on how to handle participants. We considered other approaches such as forking *T* participants, but we decided it was better to use threads spawned in the honest initiator process to communicate with the *T* Participant Servers. Threads use shared memory allowing us to mutex lock a shared participant's results map of key ids, results, and robust flag. We were worried initially about spawning 16 threads for the larger case along with encryption threads in the main thread, but those threads will be blocked most of the time waiting for the Participant Server to return results, therefore the large thread count did not end up making any performance impact.

After completing a simple demo of spawned threads that didn't communicate we worked on encryption and decryption. We used OpenSSL for our AES256 encrypt and decrypt functions. This ended up being a time consuming step solely due to the poor documentation in OpenSSL. We also realized here that we need an IV for each key and decided to hard-code it. At this point we believed we had all the pieces for the encryption so we created a small local example in our Honest Initiator function where

we encrypted the message provided with each key, xor-ed the encrypted message, then passed that to our decrypt function to decrypt and xor the results. This interaction did not work and we had to take another deep dive into Professor Wang's paper.

After some time spent analysing the scheme and how Professor Wang described his approach we realized that we had originally misunderstood the encryption and decryption flow and needed a large shift in our approach (the correct version illustrated above). At this point we went back and added in all the items we'd missed: the J machine number, P long random number, a SHA256 hash function using OpenSSL, and A the hash of $(M||P)$. We unsuccessfully put a large amount of time into trying to get a pseudo-random number generator to work. The OpenSSL and Crypto++ libraries both have HMAC-DRGB support that we tried to take advantage of, however both of them are farther in version than Ubuntu's apt repository would allow us to install. There are ways to install them from the source, but since we were building the project with docker we assumed that it would be a better use of our time to fake this function for the purpose of our demo and maintain the ability to dockerize the project easily.

After completing the new additions we implemented the encryption flow illustrated above only with the Honest Initiator's keys. Once we had the encryption and decryption functioning successfully with the revised scheme, we moved on to the threads and Honest Initiator communication. The threads send the data, check robustness in place, and add everything into the partial w results map. Once the participant communication was complete on both the Honest thread side and Participant side and still functioned correctly in place we began reworking the communication from the client and the DiSE Server. This shift was painful since we had initially had the overarching idea of the scheme incorrect and had to rework what we had to fit the new flows.

Once we had the small case working locally we began a large process of cleaning, commenting, testing, and used Valgrind to fix any memory issues we had. We added timing functionality and the ability to make a server "compromised" to trigger our robust flag.

3 Experiments & Results

After cleaning we were able to take advantage of the research performed in the first half of the project. Using docker we began running and recording the timing results of our project in various different environments.

Environment	Case	Dealer Time (Seconds)	Encryption Transaction Time (Seconds)	Decryption Transaction Time (Seconds)
1 virtual machine all on localhost	*N 5 T 3*	0.0045	4.0124	4.0133
1 virtual machine all on localhost	*N 24 T 16*	19.4463	23.7322	24.0301
Docker Container running on Virtual machines of: Ubuntu, Debian, and Kali	*N 5 T 3*	0.0068	2.0396	3.0088
Cluster of 6 machines on Google Cloud of type: E2, e2-medium 2 vCPU, 4 GB memory, Ubuntu 20.4 running our application in Docker containers	*N 5 T 3*	0.0051	2.5368	3.019

Figure 7: Timings of Dealer and both encryption and decryption transactions

One of our goals was to replicate the scheme on different operating systems. This is because it's more difficult for an attacker to compromise servers that exist spread across different platforms. We were glad to see that our demo worked in various different environments with the small case. Unfortunately we ran into some issues with Google Cloud. Originally, we wanted to use Google Cloud Run to host our Docker containers since that service provides an external ip address and runs your container, abstracting the base system away. However, we later learned that Google Cloud Run cannot support the persistent socket communication required by our project. We tried to boot up enough machines for the *N 24 T 16* case, but were capped at eight, both manually starting machines and booting them in a cluster setup. We believe there are ways around this cap to boot up enough instances to support our use case, but for our project we decided it was out of scope. For the 1 virtual machine all on localhost *N 24 T 16* case we did notice occasionally that the decryption would fail and some of our server applications would be killed by the operating system. We assumed this was either due to the massive memory usage required by this case overloading the 1 virtual machine or our buffered communication failing.

4 Future Work

Given more time there were some additional features we would have implemented to improve our demo of Professor Wang's scheme. One of which is saving the keys between runs. Ideally the Dealer class would only be run every few months to resupply the keys to the DiSE Servers. In a perfect solution, our implementation would have taken the environment we created and saved it to a priv file to be read in on server startup, so that servers could be shut down without having to redeal. A server would check for the file, and if it exists, it would use those keys. Otherwise it would block all other communication until the Dealer is run. For our demo runs, it was more convenient to be able to redeal out the keys every time so we knew it was a clean run and so that we

could reconfigure between the larger and smaller case without shutting down any servers. Another feature we would have liked to add, was switching the QTcpSockets we used out for QSslSockets with TLS. Currently the traffic between our distributed applications is unencrypted, so anyone who wanted to capture the traffic could easily retrieve the messages we're encrypting. Finally, in "DiSEServerUtil" randomNumberWithSeed function, we would have liked to used a cryptographic library for the pseudo-random number generator. We also did not generate new IV's to associate with each key which would be a necessity in a full implementation of the project. Finally we were only able to run the *N 24 T 16* case locally which is an area that could be expanded upon to debug if the errors were related to local hosting the project.

5 Discussion

We created from scratch an implementation of Professor Wang's robust distributed symmetric-key encryption scheme that only omits a few elements of the true scheme. We added parallel improvements in multiple places using OpenMP and C++'s threads. We also were able to dockerize the application and run it in multiple environments both locally using virtual machines and on Google Cloud.