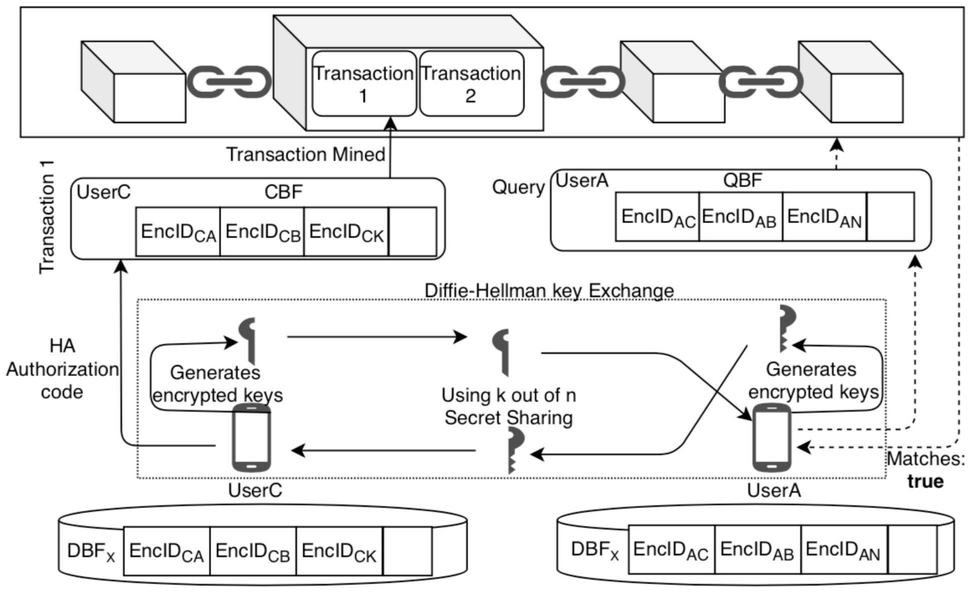
COMP4337

Assignment Report



Transaction 1
Transaction 2
Transaction Mined
Transaction 1
UserC
CBF
EncID$_{CA}$ EncID$_{CB}$ EncID$_{CK}$
Query
UserA
QBF
EncID$_{AC}$ EncID$_{AB}$ EncID$_{AN}$
Diffie-Hellman key Exchange
HA Authorization code
Generates encrypted keys
Using k out of n Secret Sharing
Generates encrypted keys
Matches: true
UserC
UserA
DBF$_X$ EncID$_{CA}$ EncID$_{CB}$ EncID$_{CK}$
DBF$_X$ EncID$_{AC}$ EncID$_{AB}$ EncID$_{AN}$

Group #27

Michael Gysel z5251938

William Yin z5017279

Executive Summary

Group #27's Did I Meet You protocol implementation generates identifiers, advertises and receives identifiers, stores encounter information, allows users to upload encounter identifiers to both the blockchain and a centralised server, and queries both the blockchain and centralised server for contact verification.

When the program begins, each user generates an Ephemeral ID using Elliptic Curve Diffie-Hellman, calculates the hash of this Ephemeral ID, and generates six shares of this Ephemeral ID using the k-out-of-n Shamir Secret Sharing mechanism, where k is 3 and n is 6. Every minute, this process repeats, generating a new Ephemeral ID, and calculating its corresponding hash and shares.

After these shares are initially calculated, the program broadcasts each share along with the hash of the Ephemeral ID over UDP, at the rate of one share every 10 seconds. Thus, when a new Ephemeral ID is created every 60 seconds, the device broadcasts these new shares. Furthermore, the program also listens to UDP broadcasts from other users, storing every share it receives. When a user has received at least 3 shares with the same hash, it uses Elliptic Curve Diffie Hellman to calculate the shared secret between itself and that sender, known as the Encounter ID. This Encounter ID represents the encounter between these two devices.

After an Encounter ID is calculated, our implementation stores this encounter information in a Daily Bloom Filter (DBF) and deletes the Encounter ID. When the program starts, a new DBF is created; moreover, every 10 minutes thereafter a new DBF is created, with each device storing a maximum of 6 DBF's. More specifically, the device encodes this Encounter ID using the murmur3 hash function and stores it in the user's DBF, built using a bitarray.

Our implementation allows the user to store this encounter information on both the blockchain and a centralised server in the event that they test positive for COVID-19. When a user prompts the program to either *uploadCBF* or *uploadCBFCentralised*, the device combines all of its DBF's into one Contact Bloom Filter (CBF), and then sends the CBF over TCP to the blockchain or centralised server where they are stored. When a CBF is sent to the blockchain and/or centralised server, its device stops generating Query Bloom Filters (QBF).

Finally, the user's device checks if they have been in contact with a person who has tested positive for COVID-19 every 60 minutes. Every 60 minutes, the user's device combines its DBF's into one QBF. Once the QBF is created, it queries both the blockchain and the centralised server to determine if the user's QBF matches any of the stored CBF's. The blockchain and centralised server then reply, informing the user of the results.

DIMY Protocol Design Implementation

Group #27's DIMY protocol successfully implements all 11 tasks specified in the Assignment Specification. The DIMY implementation can best be explained by first describing the series of threads used throughout the program.

Ephemeral ID Generation Thread (ephID_thread): The ephID_thread starts at the beginning of the program and runs for the duration of the program. Every 60 seconds a new 16 Byte Ephemeral ID is generated and its corresponding hash and Shamir secret shares are calculated. The Ephemeral ID is calculated using Elliptic Curve Diffie-Hellman, using the SEP128r1 curve from the python-ecdsa library. These are stored in the global variables ephID, hash_ephID, and send_shares, respectively. When a new Ephemeral ID is generated, ephID, hash_ephID, and send_shares are overwritten with their new values.

Send Broadcast Thread (send_broadcast_thread): The send_broadcast_thread starts after the ephID_thread is called and runs for the duration of the program, broadcasting one Shamir secret share every 10 seconds. The sender iterates through the global send_shares variable, sending one Shamir secret share every 10 seconds. Every 60 seconds, the send_broadcast_thread again sends the first share in the global send_shares variable, which has been updated with new shares by the ephID_thread.

Receive Broadcast Thread (recv_broadcast_thread): The recv_broadcast_thread also starts after the ephID_thread is called and runs for the duration of the program. When the recv_broadcast_thread receives a new share, it stores that share in the global recv_shares variable. It then checks to determine if at least 3 shares have been received for that hash and if so, proceeds to construct the sender's Ephemeral ID. If the Ephemeral ID is constructed, the shared Encounter ID is then calculated using ECDH and stored in the Daily Bloom Filter. The Encounter ID is then deleted.

new_dbfs_thread: Every 10 minutes, new_dbfs_thread creates a new DBF and stores up to 6 DBF's in the DBF_list variable.

combine_dbs_thread: Every 60 minutes, combine_dbfs_thread combines all of the DBF's in DBF_list into one QBF. This QBF is then sent to both the blockchain server and the centralised server.

Upload CBF: In addition to the series of threads that run throughout the program, the user can also upload their CBF to either the blockchain or centralised server in the event that they test positive for COVID-19. In order to do this, the user can type *uploadCBF* to send their CBF to the blockchain or *uploadCBFCentralised* to send their CBF to the centralised server. Their device will then combine the user's DBF's into one CBF and send them to be stored in the blockchain or centralised server. If the user uploads their CBF, the combine_dbfs_thread is then stopped, so QBF's are no longer created and sent to the blockchain and centralised servers.

Centralised Server Implementation: For the centralised backend server, we used the Python Flask framework and Mongodb database. To allow users to upload their Contact Bloom Filter (CBF) to the database, we built an *upload* endpoint which requests a user's CBF in json format and uploads this CBF along with a timestamp to the database. To allow users to check if they have been in contact with a person who has tested positive for COVID-19, we built a *query* endpoint which requests a user's Query Bloom Filter (QBF), matches the QBF with all of the stored CBF's for the previous 21 days, and returns "matched" or "not matched", similar to the Blockchain backend implementation.

Design Trade-offs, Project Extensions, Special Features

Design Tradeoffs: The largest design tradeoff made related to the amount of modularity versus automation of the program. We initially designed the program so that each task could be run independently; however, due to the threaded nature of the program, this was ultimately deemed infeasible, and now the entire program is run with one command.

Another trade-off made relates to the custom implementation of bloom filters for this project. Ideally, existing bloom filter packages would have been used to enable greater focus elsewhere on the project; however, existing implementations lacked any way to interact with the actual internal bitarray and/or could not create the bloom filter with the parameters we desired such that implementing our own bloom filter class proved to be the better option.

Project Extensions: Several improvements could be made to the program in order to produce a program that is ready to be used by governments for COVID-19 contract tracing. The most significant improvement to the program would be to communicate between devices using Bluetooth as opposed to WiFi which is currently used. As this program would ultimately be used with smartphones, WiFi would not be an acceptable communication protocol. Furthermore, each device would need to store its DBFs for 21 days on its device, as opposed to local variables that are lost when the program ends.

In order to make the program more scalable, we could adopt behavioural design patterns such as the Observer, Chain of Responsibility, or State design patterns. These would improve the predictability of the solution and minimise any risks of desynchronisation that may occur.

Lastly, the usability of the program would need to be improved in order for the program to be ready for use by governments for COVID-19 contact tracing. Ultimately, both Android and iOS applications would need to be designed in order for the program to achieve mass adoption amongst any large population. We could also better handle errors, particularly when interacting with the blockchain server, as the server was often inaccessible. If the blockchain server were inaccessible when the user uploads their CBF or queries the server with their QBF, we would ideally notify the user and automatically attempt the query at a later time.

Special Features: While much of the functionality uses third-party libraries, such as the ECDH key exchange and Shamir's secret sharing mechanism, our implementation is special due to its centralised server and bloom filter implementations. Unlike the original DIMY protocol, our implementation uses Flask and PyMongo. If a user tests positive for COVID-19 and wants to upload their CBF, they can query the centralised server, where the Flask server receives the CBF and sends it to the PyMongo database where it is stored. If a user wants to check if they have been in contact with anyone who has tested positive for COVID-19, they can send their QBF to the Flask server; the server responds by comparing this QBF with all of the CBF's stored over the previous 21 days in the database and returns a "match" or "not matched" response.

Secondly, our Bloom Filter implementation was unique. No packaged implementation existed which exposed the internal bitarray for operations or provided an initiator that satisfied the specifications to the degree we wanted. Furthermore, the documentation available for existing implementations were unreliable at times. This necessitated the creation of our own Bloom Filter class that wraps the bitarray package to enable Pythonic operations and encourages Object-Oriented code.

Code References

The Ephemeral ID's and Encounter ID's were calculated using Elliptic Curve Diffie-Hellman, making use of the following Python library:

tlsfuzzer (2021). python-ecdsa. GitHub repository. Retrieved from https://github.com/tlsfuzzer/python-ecdsa

The UDP broadcast communication between devices references the following Github Repository:

ninedraft (2020). python-udp. GitHub repository. Retrieved from https://github.com/ninedraft/python-udp

The Bloom Filter implementation references the following online article:

Geeks for Geeks (2020). Bloom Filters - Introduction and Implementation. Retrieved from https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/

Assignment Diary

Week 5

Mike and William thoroughly read the DIMY and assignment specifications, informing ourselves of what the DIMY protocol requires.

Week 6

Mike completed tasks 1 to 3, except for the correct generation of the Ephemeral ID. The main issue was in creating an EphID that followed the DIMY Elliptical Curve Diffie-Hellman key exchange specifications. In order to overcome this issue, we plan on using the tinyec Python library, which specifies two recommended 128-bit elliptic curve parameters using the curve 'secp128r1'.

Week 7

Mike writes midterm report. William begins code refactoring and assigned tasks. Mike and William record midterm deliverable. Mike corrects ECDH implementation, finishes tasks 4 and 5. Mike tests implementation with Raspberry Pi.

Week 8

William completes assigned tasks and first-stage refactoring. Mike completes Task 9.

Week 9

William begins preparing report. Mike builds backend server and database, completing Task 11.

Week 10

Mike and William clean code output. Mike and William finish report. Mike makes video. Mike and William prepare to package up the solution for submission, including a README.md that has instructions on how to prepare the solution for operation.