# COMP4337 – DIMY ASSIGNMENT

## MID-TERM REPORT

z5116870 – Roark Menezes

**VIDEO LINK:** https://www.dropbox.com/s/lx83mp4wl8cb8zd/DIMY%20Demo.mp4?dl=0

## Executive Summary

This assignment features an implementation of the Did I Meet You (DIMY) protocol outlined in the paper by N. Ahmed, R. A. Michelin, W. Xue, G. D. Putra, S. Ruj, S. S. Kanhere and S. Jha, incorporating key technologies such as elliptic curve based Diffie-Hellman key exchange (ECDH), Shamir secret sharing (SSS), bloom filters and communication with blockchain technology using REST API's (rather than implementation of blockchain technology). ECDH is used to generate a shared secret "encounter" identifier used to represent an encounter between two devices. Each device creates its own "ephemeral" identifier, every minute, which distinguishes itself from surrounding devices, and then broadcasts this identifier to surrounding devices. The method of broadcasting securely is by using SSS, with $k = 3$ and $n = 6$. A device that has been in contact with the broadcasting device for long enough, i.e. having received at least 3 shares, can reconstruct that devices' ephemeral identifier. These shares are broadcast every 10 seconds, so after 1 minute all shares would have been broadcast. The shared encounter identifier can then be computed using both devices ephemeral identifiers. This means that both devices will each store the same encounter identifier. Encounter identifier storage implementation is done using Bloom Filters to save on client storage and leakage.
The encounter identifier is sifted through 3 separate hashes, each of these outputs modulo the bloom filter length (for this implementation its 800000 bits) gives us the index of the bloom filter array to make 1. Every 10 minutes, a new bloom filter, called a daily bloom filter (DBF) is used for encoding the encounter identifiers, and every DBF older than 1 hour is deleted, therefore there can only be a maximum of 6 DBFs in existence. Every 60 minutes, all 6 DBFs are compiled into a single DBF, still size 800000 bits, called a query bloom filter (QBF). This QBF is then sent with a json message, as the json payload, to the backend REST API, which is hosted on amazon web service. This REST API mediates between the client device and the blockchain. Querying the QBF returns either a match, meaning the client is at risk of having contracted COVID-19, or no match indicating the client is safe. Similarly, if a client does test positive for COVID-19, all current DBF's are compiled into a single DBF, still size 800000 bits, called a contact bloom filter (CBF) which is uploaded to the blockchain via the same REST API. It is stored for comparison with future QBFs.

The communication between client devices was done using UDP port communication, using 2 different ports on the host PC's local server, 127.0.0.1. Client 1 used port 8000 to receive packets and client 2 used port 6000.

This forms a brief overview of the implementation of the DIMY protocol in the COMP4337 assignment. A brief discussion of how exactly these technologies of ECDH, SSS, bloom filters and REST API communication were implemented in the chosen programming language of Python will be detailed in the following section.

## Discussion of DIMY Protocol Implementation

The protocol was implemented in short tasks as was outlined in the COMP4337 Assignment brief. Each of these tasks is numbered in the provided Dimy.py file to make it easier to see exactly where each task has been completed. In **task 1**, we were to generate a 16-Byte ephemeral ID (EphID) every minute. This was done using the python *ecdsa* library, which has an inbuilt elliptic curve capable of producing a 16-byte public and private key. The public key is what will act as the ephemeral identifier. The *ecdsa* library however adds a leading compression constant (0x02) which was cut off before sending to the receiving UDP ports, and then added once again before building the shared secret. Once the EphID was generated, it was split into 6 chunks, with only 3 needed to rebuild the EphID (as specified by the assignment brief). This completed **task 2.**

After this was **task 3** where, each chunk, along with its index number i.e. chunk number, and the sha1 hash of the entire EphID, was added to an "advertisement" list which was sent to the other clients UDP port. Since python UDP programming does not allow for sending of lists directly, the *pickle* library was used to dump the advertisement list into binary data, and then once again rebuilt on the other side. Finding the *pickle* library was difficult since research had to be done on how to send lists through UDP packets, which was necessary for both the EphID and chunk (referred to as "share" in the code) to be transferred at the same time. Simple socket programming was used to initiate UDP communication between the two clients, with 2 sockets being set up for each: one for sending data and one for receiving data. The UDP port reading function was also run on a thread so it could continuously receive data, and all subsequent tasks were completed in this thread once data had been read. Once the correct number of shares had been received at the other client, reconstruction can begin. The reconstructed key was then compared with the hash sent in the advertisement list. If the hashes matched then the correct key was built, but if not, then this meant either not enough shares were received OR the carryover problem was encountered (as described in the DIMY protocol paper). It was explicitly said simultaneous advertisements were not necessary for this implementation, hence these scenarios were just regarded as errors, and the program continued. By verifying that the hash of the reconstructed key matches the hash in the measurements, **task 4** was completed.

Now that a client has both their own EphID and the reconstructed EphID of the other client, they can proceed with generating the shared secret; the encounter identifier (EncID). This was done using the *ecdsa* library's inbuilt secret key generation function. All that needed to be specified was the reconstructed key of the other client, and it would output the shared secret EncID. By confirming that both clients received the same EncID (using debugging messages to the terminal) **task 5** was completed. Encoding of the newly generated EncID into a daily bloom filter (DBF) was the next step and this was done by passing the EncID through 3 separate hash functions: md5, sha1 and sha3. The output of each hash function (i.e. the hex digest output) was run through the modulus operator with 800000 bits to produce 3 indexes (i.e. index1 = md5(EncID) % 800000, index2 = sha1(EncID) % 800000, index3 = sha3(EncID) % 800000). These indexes were then referenced in the DBF, and the corresponding bits were set to 1, thus completing **task 6.** A new DBF was required to be written into every 10 minutes, hence a simple variable was used to index a previously created 6 x 800000 array, where each row corresponds to each of the DBFs. Additionally, a thread was created to zero out all DBFs that were older than 1 hour. This was done as such: the first time we write into the first 6 DBFs, no deleting needs to be done since 1 hour has not passed yet. However after the first hour passes, all 6 DBFs will have been written into. At this point we need to start resetting the DBFs starting from DBF[0]. So the previously mentioned thread featured an infinite while loop (while true) and sleep call for 10 minutes, that would count through each DBF and zero them after 10 minutes from the first hour. i.e. DBF[0] zeroed out at 60 minutes, DBF[1] zeroed out at 70 mins, DBF[2] zeroed out at 80 mins and so on until the program was terminated. This means that every DBF older than 1 hour will have been reset to 0 and allow for reuse the next time the program writes into them. This then completed **task 7.**

For every 6 new DBF's created i.e. every 6 sets of 10 minutes (60 minutes) all these DBFs were be combined into a single DBF called the query bloom filter (QBF). This was done using a simple if statement that checked a counter that kept track of how many "new" (younger than 1 hour) DBFs were currently present. When this value reached 6, all 6 DBFs were run through a logical or function that produced a single 800000-bit array; the QBF, and the counter was reset. This meant every 60 minutes a new QBF was created the QBF was then sent to a function that requested a post message to the given backend REST API i.e. URL: http://ec2-3-26-37-172.ap-southeast-

2.compute.amazonaws.com:9000/comp4337/qbf/query. The QBF first however needed to be turned into the correct data type, which was specified as an ASCII base 64 string. To do this, the original QBF was turned into a single binary string. Then this string was iterated over and each number in the binary string was placed into a bit array, using the *bitarray* library. This bitarray was then converted to its byte representation, encoded in base64, using the *base64* library and finally decoded into its ASCII representation. The size of this was checked to be approximately 133kB, slightly larger than the specified 100kB, but this was due to the added overhead by the *base64* library (as confirmed by the tutors). This transformed QBF, now in base64 ASCII, was added to a json request, using the *requests* library, and then sent to the URL (as a post request). The response was interpreted as a json message (as specified by the given backend REST API) and then printed to the terminal. The response can be a match, meaning the user is at risk or no match meaning the user is safe. Now that the QBF was created and sent to the backend, both **task 8** and **task 9** were completed.

When a user becomes positively diagnosed with COVID-19, they have the *option* to upload a DBF, like the QBF, to the blockchain, via the backend REST API. This is called the contact bloom filter (CBF) and is made in pretty much the same way as the QBF, except it is not on a 60-minute timer. When the user chooses to do so, all 6 current DBFs are run through a logical or function, just as with the QBF, to produce a single bloom filter, size 800000 bits, i.e. the CBF. This CBF is then uploaded to the blockchain, via the CBF upload URL, and a message is returned confirming whether the CBF was successfully uploaded. The CBF is encoded in base 64 ascii through the same method the QBF was. Additionally, after sending the CBF to the blockchain, the user stops generating QBFs at all, so a simple if statement was added to check if a CBF was uploaded, and the QBFs were no longer created every 60 minutes. Debugging messages were printed for confirming each of the tasks, with UDP messaging used to communicate between both clients (as mentioned before). The DBF, CBF and QBF were all 100kB in size, using 3 hashes (md5, sha1 and sha3) for encoding of the EncID. Thus, **task 10** was completed.

All features of the DIMY protocol have been implemented as specified by the Assignment spec, however the extension part was not completed. For the demo, one PC was used, with two command prompt terminals acting as the 2 clients, both running different instances of the same Dimy.py file. The first one connected to port 8000 for receiving data and sending data to port 6000 and vice versa for the second terminal.

Using UDP communication is a huge trade off since it does not allow for communication of one-to-many clients unless the receiving ports of all clients is known. A wireless communication system like Bluetooth technology would have been more fitting for the DIMY protocol. This implementation is relatively short (only 260 lines of code including comments) running only 3 main threads, and 2 additional ones for sending a CBF and querying current QBF. Additionally, by using the *bitarray* library, the bloom filters are stored literally as binary arrays instead of strings, saving on space and time to encode into base64 ASCII. A future improvement would be to implement the smart contract on the blockchain that handles matching and uploading. Running the program on the command line is also rather dull, so an easy to use and view UI could be designed and coded in something like the android developer kit. Testing device to device communication with two real mobile phones would also be a good extension to complete, bringing the protocol another step closer to real implementation. **No segments of code were borrowed from the internet or and books.**

# Assignment Diary

**Week 5/4/21-11/4/12**

After the EphID was split into the 6 chunks, it was to be transmitted to the other client, but no progress was made on client-to-client communication yet. So two simple UDP sockets were set up, one for receiving on and one for sending to. Command line arguments were used to specify the port numbers, so 2 instances of the Dimy.py file could be run, one as ***python Dimy.py 8000 6000*** and the other with the port numbers swapped around. So one client sends to the others receiving port and vice versa. The receiving data part was run on a thread, to continuously receive data, while the sending was done every 10 seconds, for each of the 6 shares. The sending was done in a for loop, iterating over all the shares, and the receiving was done in an infinite while loop. The received shares were tracked on the other side by being printed out however an issue was how to send the share, its index, and the hash of the EphID all in one UDP message. A specific library was found to this called *pickle* and it had 2 useful functions dumps and loads, which can turn the 3-element list of share, share index and hash into a binary string and then rebuild it on the receiver side.

**Week 12/4/21-18/4/21**

A counter was used to track that at least 3 shares were received and then a rebuild was attempted. If successful, its' hash was compared with the hash sent in the UDP message and only if these matched was it confirmed as the reconstructed ID. To build the EncID a lot of thought went into the mathematical side of extracting the random number from the rebuilt EphID, and then raising the original ecdh curve used to this random number. However after some more research, it was found the *ecdsa* library has within it a function to get the shared EncID, given the other persons EphID (i.e. the one reconstructed from at least 3 received shares). The output of this function was compared on both terminals and found to be the same. This EncID was then encoded into a bloom filter by running it through 3 separate hashes, getting the result of this modulo 800000 (to get a value within the size of the bloom filter) and then changing the value of this index in the bloom filter to 1. Essentially creating a bloom filter with 3 hashes. The EncID was then deleted. Every 3 shares of a new EphID received, the same process was done; reconstruct the EphID, get the EncID, encode the EncID into this daily bloom filter and then delete the EncID. The bloom filter counter was rolled over every 10 minutes, creating 6 of them per hour, and every 10 minutes after that we come back to the first BF, reset it, write in it for 10 minutes and repeat for the next one and so on. This essentially deletes all bloom filters older than 1 hour.

**Week 19/4/21-25/4/21**

We were given an extension to Monday 5pm (26/4/21) so more time was available to complete the assignment. For every 6 new DBFs created, a QBF was created by combining all 6 of these into a single BF, same length as one DBF, using bitwise OR. This was then sent to the backend API using the *requests* module in python. The QBF was put into a base64 ASCII format by first decoding it into a bitarray, to save on space, and then encoding into base64 ASCII. It was then put into the json payload of a post request and sent to the QBF query URL given with the assignment. Similarly, for sending CBFs, the same function used to the send the QBFs was used, except the generation of the CBF was a little bit different to the QBF. Unlike being generated every 60 mins, the CBF is generated at will. This was simulated by running a thread at around 1 hour and 10 minutes, all 6 DBFs at that time were compiled into the CBF and sent to the CBF upload URL. Another thread was created to simulate sending a QBF at will to check for infection. Risk analysis was then confirmed by seeing that the first QBF sent (at 60 minutes) returned no match, but the second QBF sent (after uploading the CBF) returned a match. This completed all the tasks presented in the assignment.