**COMP4337:DIMY Protocol Implementation Report**

Liang Zhang - z5526077

David Shin - z5477450

# How to run the code.

1. Run this in the UNSW CSE server and you need a virtual environment
   * python -m venv /path/to/new/virtual/environment
   * source /path/to/new/virtual/environment/activate
   * 
2. Open 5 terminals: one for server(DimpyServer.py), three for nodes(Dimpy.py), and one for attacker node (attacker.py).
3. Make sure you activate the virtual environment in step 1. Install all the required python libraries from ‘requirements.txt’.
   * pip install -r requirements.txt
4. Run the server (DimpyServer.py)
   * python3 DimpyServer.py
5. Run the client (Dimpy.py) in three terminals
   * python3 Dimpy.py t k n
6. To report positive as a node, just enter y in the terminal at any time after at least one EncID was created.
7. Run the attacker (attacker.py), you may test the attacker.py after checking that both the Dimpy.py and DimpyServer.py are working correctly.
   * python3 attacker.py k

# Executive summary

This program implements a privacy-preserving system during covid-19 and the protocol is called Did I Meet You(DIMY). This program consists of two main components: nodes(Dimpy.py) and server(DimpyServer.py), in additionally, there is an attacker(attacker.py) to simulate a man in the middle attack on the inter-node communication.

When the nodes start running, they initially generates a 32-byte after every t seconds and will be splitted into n chunks with Shamir Secret Sharing mechanism, then the node will periodically broadcast via UDP to advertise these chunks with 50% probability of dropping messages to pretend the complexity of contacts in real life. After k out of shares being successfully advertised and received by nodes, that means the nodes have remained in contact, it reconstructs that node`s EphID then performs X25519 Diffe Hellman key exchange with its own private key and the reconstructed node`s public key to generate the secret Encounter ID(EncID). The EncID is deleted and encoded into Daily Bloom Filter after it is successfully constructed, all EncIDs will be stored in the same DBF. The DBFs are managed by a dictionary with timestamp and itselves, after every t\*6 seconds a new DBF is initiated and there is a maximum of 6 DBFs on each node. Any DBF that is older than Dt mins will be deleted from the dictionary which is the node`s storage. Every Dt minutes a node will union all the available DBFs to Query Bloom Filter, once a user who manually entered “y” in the command line as reporting positive, the node will combine all the available DBF into a Contact Bloom Filter and send CBF to the backend server via TCP, then it stops generating the QBFs.

As long as the backend server receives CBF, it will analyse whether the node is a close contact by comparing the intersection of bitarries within CBF and QBF, then send the analysis result back to the nodes. Task 11-a, 11-c, 11-d is explained in this report.

# Discussion of implementing the DIMY protocol.

All the tasks are finished, the detailed discussion of tasks are:

* EphID Generation & Shamir Secret Sharing
* Broadcasting Shares & re-construction
* Generate EncID by X25519 Diffie-Hellman exchange with reconstructed EphID and EphIDBytes
* Customised the original python bloom\_filter library for adding bloom-filter size in bits and number of hashes. All DBFs, QBFs, CBFs are 100k bits, 3 hashes and 0.1 error rate.(The bits size may varying for designing trade-offs in the end)
* Encoding encIDs into same DBF then deleted the encIDs and generate a new one DBF every t\*6 seconds with a maximum of 6.
* Combine available DBFs to a single QBF.
* Communications between back-end server and nodes via TCP with pickle
* Combine available DBFs to CBF after an user manually enter “y” in the command line of that node and then upload CBF to back-end server in serialized bit arrays with a tag “CBF”, the server will unpickle the serialized data deserialize the bitarrays and then store them in a list called storedCBF. Then inform the nodes “CBF uploaded successfully”
* After every Dt minutes, the nodes send the serialized bitarrays of QBF with a tag “QBF”, the server deserializes the bitarrays and starts analysing the QBF with CBF. If there is no CBF stored, the server immediately sends “Not close contact” back to the nodes, otherwise it will check if there are intersections of the bitarrays. There is an initialized status variable `match=False`. If there are intersections that means that node is a close contact then the server report “Close contact” to that node, otherwise the server reports “Not a close contact”

**For testing, the best option is:**

* Start the backend server and start with just two nodes, after at least a couple encIDs are created by those two nodes(node1 and node2), enter “y” in just one node`s terminal(for example node1) and check if this node receives messages indicating the CBF is received from the server. After the other node(node2) submits the QBF, check the security analysis result from the server in this node`s terminal to see if node2 is a close contact. Then run the third node(node3), wait for this node3 to submit the QBF and check if this node is a close contact.

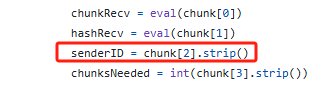
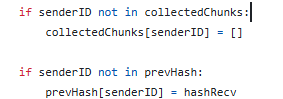
# Discussion of design trade-offs considered and made.

**List what you consider is special about your implementation.**

* We used a list ‘myChunks’ to store the secrets before broadcasting the shares, and once a node receives chunks, it splits the chunks by eval and compares the chunkRecv with myChunks, if they are the same, ignoring that broadcasting.



* However, the above method could not prevent the node encountering itself which means it generates encID with themself, so we added senderID in the broadcasting data to differentiate the node itselves.



* ephIDs are randomly generated from the X25519 private keys: although it is the most commonly used, it has better performance and fast arithmetic which decreases the loads on the client's side. But it lacks authentication mechanisms which is vulnerable to man in the middle attack, which is the replay attack we use in attacker.py.
* The inter-node communication and c to s communication are using eval and pickle, they are easy to implement, but they are not safe and lack integrity and authentication when serializing and deserializing data.

**Describe possible improvements and extensions to your program and indicate how you could realise them.**

We need to add extra mechanisms to ensure the inter-node UDP broadcasting and C2S TCP communication are not able to be tampered with while sending.

To achieve the Integrity & Authentication on UDP broadcasting, we can use Ed25519 to generate key pairs. Once the recipients receive the shares, they need to verify that the payload signed by the node`s public key is the same as the signature.

To achieve the I & A on C2S TCP communication, we could either implement HAMC or TLS to prevent the data being tampered while communicating with nodes.

# Indicate code that borrowed

attacker.py is finished with the help of ChatGPT

# Task 11

## Task 11-a: Explain the purpose of each of the security mechanism employed in the DIMY protocol.

Security Mechanisms:

* Ephemeral Identifiers
* K-out-of-n Shamir’s Secret Sharing
* Diffie-Hellman Key Exchange
* Bloom Filters
* Data expiry
* Hashing

**Ephemeral Identifiers**

In DIMY protocol, the implementation utilises constantly changing ephemeral identifiers (eID). These eIDs are randomly generated from the X25519 private keys. By using an elliptic curve, it helps increase the randomisation of the eID generation. By also constantly changing the eID within the set {15, 18, 21, 24, 27} seconds, the protocol also prevents long-term tracking of nodes. Even if an attacker intercepts these broadcasts, they cannot reliably link successive eIDs to a given node. Thus, increasing and enhancing user privacy between nodes, helping to keep all data private and secure.

**K-out-of-n Shamir’s Secret Sharing**

Each ephemeral identifier is split into n shares using Shamir’s Secret Sharing in the DIMY implementation. A minimum of k shares are required to recover the original secret; in the case of the DIMY implementation, the ephemeral identifier which was generated by the elliptic curve. Using Shamir’s secret sharing, it strengthens the temporal threshold for registering an encounter. Secret sharing also makes it incredibly difficult for an attacker to reconstruct the ephemeral identifiers unless they receive all k shares required. Thus, protecting the communicating nodes from passive eavesdropping attacks as well as dropped message reconstruction.

**Diffie-Hellman Key Exchange**

After reconstructing the ephemeral identifier, the DIMY implementation uses the Diffie-Hellman key exchange algorithm to compute a shared secret. By using Diffie-Hellman to arrive at an encounterID, it promotes secrecy and ensures mutual anonymity between the two nodes who have encountered each other. Even if the ephemeral private keys are compromised, the computed encounter IDs remain secure. As Diffie-Hellman key exchange does not send the shared secret over via a UDP or TCP connection, attackers are unable to arrive at the same shared secret as the shared secret is computed on their individual, local devices. Even after getting the private ephemeral ID, they will not be a step closer in achieving the ephemeral identifiers, making it resistant to intercepting and replay attacks.

**Bloom Filters**

The encounter IDs are deleted after being added to the daily bloom filter (dbf). Each node stores a maximum of 6 nodes, with each node having a lifespan. Bloom filters are probabilistic methods of checking if an element is within the array. The following bloom filters are sent over a local network and thus require secrecy and data authentication.

* Contact Bloom Filter
* Query Bloom Filter

Even if the filter is accessed, as the filters themselves do not contain any sensitive information such as the direct encounter IDs themselves, rather a True or False answer of if an element is stored within this bloom filter, if this data were to be intercepted, attackers would not gain any beneficial information. Additionally, the daily bloom filter is also periodically deleted, minimizing the exposure to historical contact data. These bloom filters preserve the privacy of the infected users and their close contacts.

**Data expiry**

Each node does not store old data, and periodically deletes data based on their lifespan. This ensures historical data is not accessed and ensures all data is up to date.

**Hashing**

The ephemeral identifiers are also hashed and verified by the receiving node to ensure data integrity.

## Task 11-c: Explain how the attacker node can possibly launch your selected attack on the communication between nodes and the backend server.

Currently, the client uploads CBF and QBF to the server with pickle serialise and deserialise. It is vulnerable as code execution, which means the server and client lack the verification during data transmission to ensure the integrity and authentication of the data. Therefore, there is still a great possibility of man-in-the-middle attack, directly decoding and reading the original data, tampering with it and making the data unusable. If this protocol is applied in real life, the impact and consequences include: failure to correctly handle close contact users, or user misreporting.

## Task 11-d: Discuss the countermeasures that can be taken to mitigate the effects of the attacks described in Segments 11-B and 11-C.

As the attack in 11-b simulated replay attacks and 11-A both lacks of measures to ensure data authentication and integrity, the countermeasure could be:

**Task 11-B**

Before officially splitting and broadcasting ephid, add a layer of "digital signature + anti-replay" protection in the protocol:

1. Signature key pair generation and distribution

* Whenever a node regenerates the EphID of this round, use Ed25519 to generate a pair of public and private keys.
* Broadcast the public key to all neighbor nodes via UDP broadcast for subsequent use.

2. Message packaging and signing

Assemble the original EphID into a message with metadata like this format:

* payload = (EphIDBytes + timestamp) + salt
* timestamp: used to verify the time, for example, when k is 3, the node should receive all chunks and reconstruct the ephID in about 9 seconds;
* salt: each node uses the same salt to splice the payload and the same salt to disassemble the payload. Regenerate every half an hour or an hour;

Generate a signature for the payload using the private key:

* Concatenate to get the complete signature data: signed\_payload = (payload & signature)

3. Secret sharing and broadcasting

* Apply Shamir secret sharing to signed\_payload to generate n shares.
* Broadcast each share via UDP in the format of share & senderID

4. Receiver verification process

1. After collecting at least k shares, reconstruct signed\_payload.
2. Extract payload and signature from signed\_payload, and further extract EphIDBytes, timestamp using salt.
3. Time verification: Check whether the timestamp difference is within an acceptable window (e.g. 9 seconds or up to 10 seconds);
4. Signature verification: Use the same signature algorithm on the client to perform hash verification on the decoded payload and compare the signatures. If the verification fails, it will be discarded. Only after success will the ephID be used for subsequent Diffie-Hellman exchanges.

**Task 11-C:**

The safer way to mitigate the vulnerabilities of pickle is using JSON or implementing HMAC or TLS for the transport layer to authenticate.

* Use JSON to serialize and deserialize bitarray

Convert the bitarray to a Base64 string and encapsulate it in a format such as {"tag":"QBF","bits":"xxxx"}. JSON does not need to execute additional code to parse the bitarray. But this is still risky because the data can still be intercepted and tampered with during transmission, so additional HAMC or TLS should be added.

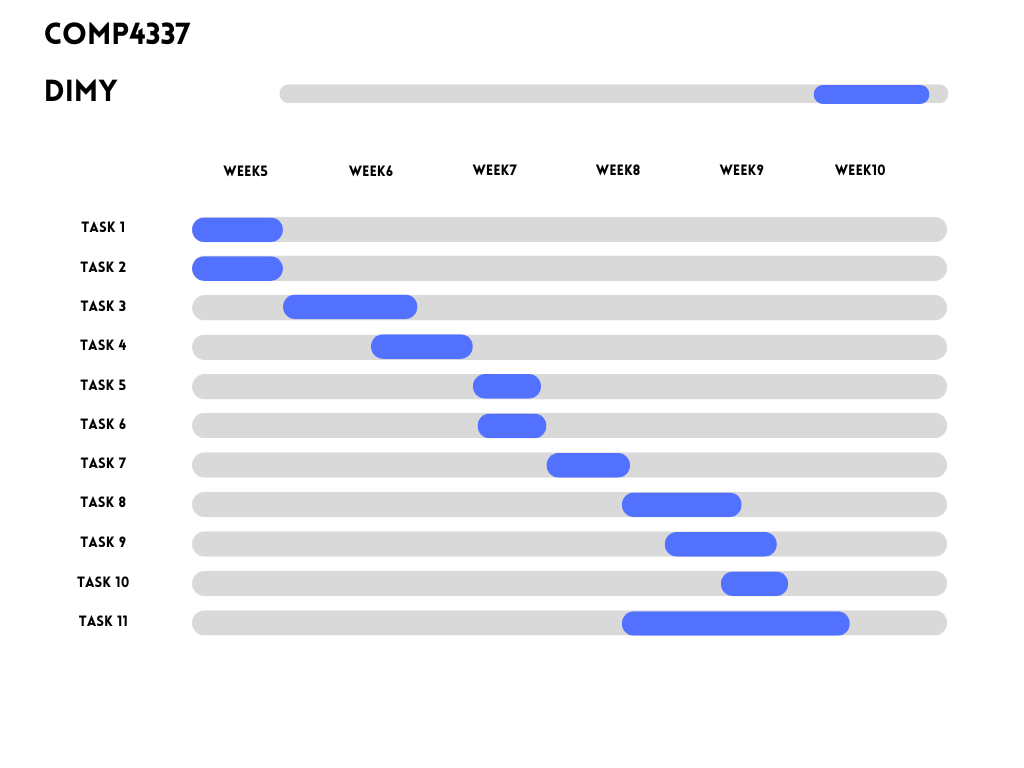
* Add HMAC verification

Both the server and the client verify the HMAC to ensure that the data has not been tampered with and comes from a trusted sender

* TLS

Implement two-way authentication TLS for data transmitted over TCP, both the client and the server verify their certificates, but this may be difficult to implement because additional certificate management needs to be considered.

# Assignment Diary



David Shin:

* Task 1 & Task 2: started from and completed in week 5.
* Task 3: started in later of week 5 and completed in week 6.
* Task 4: started in the middle of week 6 and completed at the end of week 6.
* Task 11: started in week 8 and finished in week 9.

Liang Zhang:

* Task 5: Started and finished in week 7.
* Task 6: Started and finished in week 7.
* Task 7: Started in week 7 and finished in week 8.
* Task 8: Started in week 8 and finished in early of week 9.
* Task 11 b,c, and d:

**Common part:**

* Task 9: started in week 8 and finished in week 9.
* Task 10: started in week 9 and finished in week 9.

Liang started a draft of task 9 (DBF combined to CBF) and Shin helped to complete task 9 (Optimized the code and completed the server part). After communication, we found that there were problems with the code's execution process and thread allocation, and Shin then independently completed the code optimization work.