Client -> Needs to be given s command to wake up, not bound to any port.
Middlebox -> Docker container which is bind with localhost at port 5001
Server-> Docker container which is bind with localhost at port 5002
Rule-generator-> Docker container which is bind with localhost at port 5003

Step 1-> (Rule Generator → Client) Public Key & Signature of ruleset Exchange:
This step establishes authenticity and integrity before sending sensitive rule data. The Rule Generator first generates a 2048-bit RSA key pair. It serializes the ruleset (e.g., `['hack', 'malware']`) into bytes, computes a SHA-256 digest, and then double-hashes it for consistency. Using its private key, it creates a PKCS#1 v1.5 signature over the hashed digest. When the client connects, the Rule Generator sends the length of its public key, the public key itself (in PEM format), and the signature. The client will later use this public key to verify that the ruleset came from the legitimate Rule Generator and has not been altered. This handshake acts as the trust anchor for the rest of the secure communication protocol.

Step 2-> (Rule Generator → Middlebox) Ruleset Transfer:
Once the Rule Generator has established trust with the client, it sends the actual ruleset to the Middlebox for processing. It opens a socket connection to the Middlebox's designated port (`5001`) and serializes the ruleset using Python's `pickle` module. This serialized byte stream is sent directly over the socket. The Middlebox will later use these rules to evaluate network data or enforce policies. This transfer assumes that authenticity was already ensured earlier via the Rule Generator's key exchange with the client.

Step 3-> (Client ↔ Server) Key Exchange & Salt Transmission:
The client and server establish a secure communication channel using a Diffie–Hellman–based key exchange. The client generates its own public–private key pair and sends its public key to the server. The server responds with its encrypted public key, which the client receives in full.
After exchanging keys, the client sends a **salt** value to the server for use in later encryption and tokenization steps. Using the received ciphertext and its private key, the client decrypts the shared secret.
From this shared secret, the client derives three cryptographic materials via SHA-512 hashing:

- **SSL key (32 bytes)** – for securing the communication channel
- **Encryption key (16 bytes)** – for encrypting messages
- **Randomization key (16 bytes)** – for obfuscation in garbling

Step 4-> (Client ↔ Middlebox) Tokenisation & Salt Exchange
The client connects to the middlebox to exchange tokenisation parameters and a pre-generated salt value. Upon connection:

1. The middlebox sends an **option string** containing tokenisation settings (option type and minimum token length).

2. The client displays its current salt (in integer form) and transmits the **salt** to the middlebox.

3. The middlebox responds with a **ruleset hash digest**, used later for verifying the integrity of the filtering ruleset.

The received option string is parsed into:

- **Tokenisation option** – type of tokenisation algorithm or rule applied

- **Minimum token length** – smallest substring size considered in token generation

This process ensures both parties share a consistent salt and tokenisation parameters before secure processing begins.

Step 5 -> (Client) Signature Verification using ruleset received from middlebox:

Step 6 -> (Client) Parsing the Bristol Circuit File:
This function processes a Bristol-format Boolean circuit file to prepare it for garbled circuit execution. The Bristol format specifies the circuit's wires, gates, and their connections.

Steps:

1. Read Header Information
    - Line 1: Number of gates and total number of wires.
    - Line 2: Number of input wires from the client (garbler), number of input wires from the middlebox (evaluator), and number of output wires.

2. Parse Gate Descriptions (from line 3 onwards)
    - For each gate, extract:
        - Number of inputs/outputs
        - IDs of input and output wires
        - Gate type (e.g., AND, XOR, INV)
    - Count the usage frequency of each wire (to identify output wires later).

3. Identify Real Output Wires
    - Output wires are those never used as inputs to other gates.

4. Return Structured Circuit Representation
    - Total number of wires
    - Inputs from each party
    - Number of outputs
    - List of output wire IDs

    - List of gates with their structure and type

This parsed data becomes the blueprint for garbling, evaluation, and output decoding in the secure computation protocol.

Step 7 -> (Client) Generating randomness in circuit using k_rand:

2 values **delta, X** are generated using k_rand as seed (both of them are generated such that have lsb as 1). A value **P** is generated using X^delta (lsb as 0 since both have 1) . Now the wire labels are generated using this k_rand as seed as well. Here, wire labels means generating 2, 32-byte strings one is assigned to bit 0 and other to bit 1.

Here, the "FREE-XOR" optimisation has been used. Using this optimisation we generate the label for bit 1, using the label of bit 0, which reduces the overhead greatly. So label of bit 0 is 32-byte random value and label for bit 1 is (label_0 ^ delta).

Note: Here we generate the sample values of all the wire labels but in fact only the first 256 wire labels have their true value, the rest are just given an initial value(this could even have been /x00*16 or anything).

At the end of this step we have:

1) 2 wire values corresponding to bit 0 and bit 1 for all the wires presnet in the circuit, generated using FREE-XOR
2) P, delta

Step 8 -> (Client) Garbling the circuit:

Every circuit can be broken into 3 basic gates INV, AND and XOR. Here for the XOR,INV gates FREE-XOR optimisation has been used while for AND gates the HALF-GATE optimisation has been used. We already have the input and ouput wire number/labels for a gate using the methods above. A counter is initialised as 0, an whenever there is AND gate it is update by +2.

1. XOR Gate Garbling (Free-XOR)
Given:

- Input wires: `A(a0, a1)` and `B(b0, b1)`
- Output wire: `C(c0, c1)`

Process:

- `c0 = a0 ⊕ b0`
- `c1 = c0 ⊕ Δ` (Δ is the global difference between `0`-label and `1`-label)

No cryptographic hashing is needed — labels are derived using XOR only.

---

2. INV (NOT) Gate Garbling (Free-XOR)
Given:

- Input wire: `A(a0, a1)`
- Output wire: `C(c0, c1)`

Process:

- `c1 = a0 ⊕ P` (P is a fixed public bit-flip mask)
- `c0 = c1 ⊕ Δ`

This inverts the logical value without extra encryption.

---

3. AND Gate Garbling (Half-Gate Optimization)
Given:

- Input wires: `A(a0, a1)` and `B(b0, b1)`
- Output wire: `C(c0, c1)`

Process:

1. **Extract select bits**
   - `p_a = LSB(a0)`
   - `p_b = LSB(b0)`
2. **First Half-Gate (G) -> Garbler side**
   - Compute hashes of both `a0` and `a1` with a counter.
   - Derive `T_G` using XOR of these hashes and `p_b * Δ`.
   - Compute partial output `X_G`.

3. **Second Half-Gate (E) -> Evaluator side**
   - Compute hashes of `b0` and `b1` with counter+1.
   - Derive `T_E` using XOR of these hashes and `a0`.
   - Compute partial output `X_E`.

4. **Combine outputs**
   - `c0 = X_G ⊕ X_E`
   - `c1 = c0 ⊕ Δ`

Result of this step:

- **Garbled Table**: Only two ciphertexts `[T_G, T_E]` are stored. Only needed for AND gate as XOR,INV gates output labels can be generated without tables.

Note: Here, the output labels are generated based on the gate type. So only input labels(key +plaintext) are random while other are generated using these labels. So, k_rand is used to generated the input wires(here 256) and the rest are generated using these 256 wires.

At then end of this step client has the complete circuit with wire labels correctly generated and a garbled tables list for AND gates.

Step 9-> (Client) Generating the sample evaluator package to be sent to middlebox.

The package consists of :

1) P (needed for INV gates. Safe to give as random and middlebox has no extra information about the bit based on this)
2) Middlebox input wire indices
3) Middlebox input wire labels.(Since it is sample so client has the plaintext from which he can choose the wire label corresponding to the bit the plaintext has)
4) Output map(needed for sampling the output wire label to its corresponding bit)
5) Client input indices(key)
6) Client input labels
7) Output wire indices
8) Gates. (the circuit is made public within a network).
9) The client output (need for checking the validity of the circuit)

Note: Here, 3,6 means that based on the input it gives the corresponding correct wire label. Like suppose key is 0101, then the package consists of label0,label1,label0,label1.

Step 10 -> (Client ↔ Middlebox) Sample connection for checking validity of the circuit:

The client evaluates the circuit based on the input it chose and gives it to middlebox to match along with the evaluator package. If the client evaluation byte string and middlebox byte string do not match then middlebox closes it's connection with error = “Circuit Error”.

Note. Here no OT is required since this is sample.

Step 11-> (Client) Generating the real evaluator package to be sent to middlebox.

The package consists of :

1) P (needed for INV gates. Safe to give as random and middlebox has no extra information about the bit based on this)
2) Middlebox input wire indices
3) Output map(needed for sampling the output wire label to its corresponding bit)
4) Client input indices(key)
5) Client input labels
6) Output wire indices
7) Gates. (the circuit is made public within a network).
8) Garbled tables.

Step 12-> (Client ↔ Middlebox) Connection for OT+sending the package:

1) Client sends package.
2) Middlebox responds with the number of ruleset tokens which are generated based on the tokenisation scheme selected above.
3) OT takes place between the middlebox and client for the ruleset tokens as client should not know the middlebox input and middlebox should not get to know both the labels.(Delta has not been shared with the middlebox so he can't generate both wire labels..). **OT is based on CO15 protocol.**

Process in `handle_oblivious_transfer()`:

1. **Key Generation**
    - The garbler chooses a random scalar `y ∈ [1, n-1]` from the elliptic curve group.
    - Computes `S = y * G` (where `G` is the generator point of the curve).
    - Computes `T = y * S` (used for masking the alternative choice).

2. **Send Public Key**
    - The garbler sends `S` (as bytes) to the evaluator.
    - This acts like a temporary public key for the OT session.

3. **For each input bit position** (`num` wires × 128 bits per label ID):
    - The evaluator sends an elliptic curve point `R_i_j` (its OT choice encoding).
    - The garbler derives **two symmetric keys** using hashing:
        - `k_i0_j = Hash(S, R_i_j, y * R_i_j)` → for label 0
        - `k_i1_j = Hash(S, R_i_j, (y * R_i_j) - T)` → for label 1

    - These keys are then XOR-masked with the actual wire labels:
        - `ci_0 = k_i0_j ⊕ wire_label[offset+j][0]`
        - `ci_1 = k_i1_j ⊕ wire_label[offset+j][1]`

4. **Send Both Encrypted Labels**
    - The garbler sends `ci_0` and `ci_1` to the evaluator.
    - The evaluator, knowing only the correct decryption key for its chosen bit, can recover **exactly one** of the two labels.

Security Intuition:

- The evaluator's choice bit is hidden in `R_i_j`.
- The garbler cannot tell which label was actually recovered.
- The evaluator cannot decrypt the other label without solving a hard elliptic curve problem (Discrete Log).

Step 12-> (Server ↔ Middlebox) Connection for OT+sending the package:

Steps 11-12 repeated at server side. He can generate same random values as he has k_rand from key exchange. Middlebox only proceeds if evaluator package and labels he receives after OT from both are same.

This eliminates the possibility of a threat actor.

Step 13 -> (Middlebox) Creation of AVL tree based on ruleset.

AVL tree is a self balanced binary search tree and is used here as it has logarithmic times for lookup,insertion and deletion. Middleobx stores the ruleset encrypted values here for faster lookup and updation/deletion.

Step 14 -> (Client) Tokens generation:

Tokenisation Mode Selection

- The client receives `option` (1 or 2) and `min_length` from the middlebox.
- If `option == 1`, **window-based tokenisation** is applied to `msg` with the specified window size (`min_length`).
- If `option == 2`, **delimiter-based tokenisation** is applied.
- Both methods return a list of `tokens` and associated `salts`.

Circuit-Based Pre-Encryption

- Each token is first **PKCS#7 padded** (128-bit block size) and converted to bits.
- The padded token bits are concatenated with `k_bits` (derived secret key bits) and fed into `evaluate_circuit()`, which implements a garbled-circuit evaluation.
- The output bits are converted back to bytes and stored as **pre-encrypted tokens**.

Final Token Encryption with Salt

- For each pre-encrypted token, an AES-ECB cipher is instantiated with the token bytes as the AES key.
- The associated salt is converted to 8 bytes, padded, and then encrypted under this AES key.
- The ciphertext is reduced modulo `RS` (bandwidth optimization) and stored as the final **encrypted token**..

Step 15 -> (Client ↔ Middlebox) payload transfer:

Payload consists of encrypted message, tokenisation length and encrypted tokens. The middlebox does:

1) Parses the payload to get the 3 things.
2) **Checks if tokenisation length is same as decided before, if not stops the connection**

3) Parses the encrypted tokens and update the AVL tree based on if lookup return true.
4) Calculates the percentage of tokens found in lookup and takes action base on it
5) If less than 0.1 ignores (Since sometimes the tokens can be part of a large legit word)
6) If between 0.1 and 0.3 warns the server bu does not drop the packet.
7) If greater than 0.3 then drop the packet and warns ther server that this client should be blacklisted.

Step 16 -> (Server ↔ Middlebox) payload transfer:

If malicious percentage is less than 0.3 then server gets the payload.

Here, the server does:

1) Parses the data, and then decrypts the message.
2) **It generates the encrypted tokens and then matches with the received encrypted tokens, if they do not match then treats it as malicious.**

Step 17 -> (Server ↔ Middlebox ↔ Client) Msg transfer:

If everything worked fine, then server sends back decrypted message to middlebox and then to client.