Deep Packet Inspection Over Encrypted Traffic

Mrittika, Divyanshu, Romir

August 11, 2025

1 Introduction

Deep Packet Inspection (DPI) enables middleboxes in a network to inspect, monitor, and manipulate packet payloads for purposes such as intrusion detection (IDS), data exfiltration prevention, and parental filtering. Till date, network middleboxes cannot work without decrypting the traffic. Increasing use of HTTPS and other encryption protocols in data transmission over internet is thus forcing many enterprise networks to trade-off either privacy or security. Either traffic remains private but invisible to DPI tools, or it is decrypted at middleboxes, violating user privacy and the end-to-end security guarantees of SS-L/TLS.

BlindBox is a DPI system that bridges this gap by allowing middleboxes to perform DPI directly over encrypted traffic without decrypting it, thus preserving both privacy and security. This report provides a comprehensive analysis of the system architecture, cryptographic protocols, privacy models, performance, and applicability of BlindBox.

2 Overwiew

Standard middlebox deployment consists of the four parties: sender (S), receiver (R), middlebox (MB), and rule generator (RG). S and R send traffic through MB. MB allows S and R to communicate unless MB observes an attack rule in their traffic. RG generates these attack rules (also called signatures) that are used by MB in detecting attacks.Each rule attempts to describe an attack and it contains fields such as:one or more keywords to be matched in the traffic, offset information for each keyword, and sometimes regular expressions. The RG role is performed today by organizations like Emerging Threats, McAfee, or Symantec.

2.1 Use Cases

Before formalizing the architecture and implementation details, lets look at some scenerios where the both privacy and security both are necessary thus making BlindBox a plausible choice.

University network. Suppose the University policy requires that all student traffic be monitored for botnet signatures and illegal activity by a middlebox(MB) running an IDS. McAfee (RG) is the service that provides attack rules to the middlebox and all university students trust it. However, someone skeptical about her privacy at the middlebox end, can install BlindBox HTTPS with McAfee's public key, allowing the IDS to scan her traffic for McAfee's signatures, but not read her private messages.

ISP service. Suppose someone with a child at home wants all incoming traffic to be filtered for adult content, so he registers with his Internet Service Provider(ISP) enable parental filtering. However with increase in fraudulent activities at the ISPs end, like, selling user browsing data to marketers, the user is concerned with his privacy. So he installs BlindBox HTTPS on his home computer with the public key of a RG that he trusts, allowing his traffic to be scanned for the RG's rules, but no other data.

As clear from the above examples, one significant drawback is that there must exist a RG which Sender, Receiver and the MB trust with rule generation, otherwise the parties cannot use BlindBox.

2.2 System requirements

The fundamental system requirements for BlindBox are :

1. BlindBox must maintain MB's ability to enforce its policies (that is, to detect rules and drop/alert accordingly).
2. Endpoints must not gain access to the rules, rules must only be known to the MB.
3. The MB cannot read the user's traffic, except the portions of the traffic which are considered suspicious based on the attack rules.

The last requirement is the key feature that BlindBox adds in addition to the traditional IDS deployments.

2.3 Threat Model

There are two types of attackers in our setup.

Attacker at endpoints. One endpoint can behave maliciously, but atleast one endpoint must be honest. This is a fundamental requirement of any IDS because otherwise two malicious endpoints can agree on a secret key and encrypt their traffic under that key with a strong encryption key, making prevention impossible by the security properties of the encryption scheme. Similarly, the assumption that one endpoint is honest is also the default for exfiltration detection and parental filtering today. Parental filters can assume one endpoint is innocent under the expectation that 8-year-olds are unlikely replace their network protocol stack or install tunneling software.

Attacker at the middlebox. Unlike conventional IDS, an attacker at the MB is a new threat model introduced in the BlindBox setting. We assume that the MB performs the detection honestly, but this attacker at the MB tries to read all the data accessible to the MB. Given this threat model, the goal is to hide the content of the traffic from MB, while allowing MB to do DPI. We do not seek to hide the attack rules from the MB itself, many times these rules are hardcoded in the MB.

2.4 Privacy Models

The two kinds of privacy models that can be considered by BlindBox are:

Exact Match Privacy. MB learns only whether specific attack keywords appear at certain positions in the traffic.It learns nothing about the other parts of the traffic. Traffic which does not match a suspicious keyword remains unreadable to the MB.

Probable Cause Privacy. MB can decrypt the traffic only if a suspicious attack keyword is detected in it. robable cause privacy is useful for IDS tasks which require regular expressions or scripting to complete their analysis.

3 System Architecture

BlindBox involves four parties: (a) **Sender (S)** and **Receiver (R)**: Endpoints exchanging encrypted data. (b) **Middlebox (MB)**: Scans encrypted traffic for attack rules. (c) **Rule Generator (RG)**: Trusted authority issuing signed attack signatures/rules.

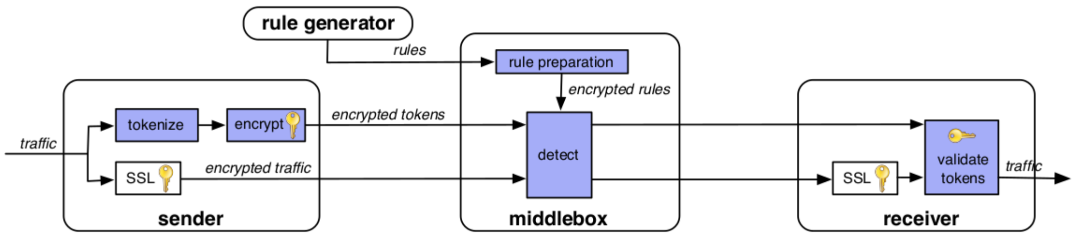


Figure 1: System architecture. Shaded boxes indicate algorithms added by BlindBox.

Prior to any connection, RG generates a set of rules which contain a list of suspicious keywords and signs the rules with its private key and shares them with MB, its customer. S and R, who trust RG, install a BlindBox HTTPS configuration which includes RG's public key. Beyond this initial setup, RG is never directly involved in the protocol. We now discuss the interactions between R, S, and MB when R and S open a connection in a network monitored by MB.

Connection setup. First the sender and receiver setup the connection by exchanging key k_0 using the regular SSL handshake and use it to derive three keys as follows,

- k_{SSL}: the regular SSL key used to the encrypt the traffic
- k: the key needed for the detection to work
- k_{rand}: used as a seed for randomness

Alongside, the MB performs the required connection setup with the RG and obtains the rules. After obtaining the rules, it encrypts them in the *rule preparation* phase, these encrypted rules are the ones that facilitate detection without decryption at the middlebox. This entire connection setup occurs in such a way that the MB learns nothing about any of the keys and the S and R learn nothing about the rules.

Sending traffic. At the sender side two kinds of encryptions take place. One is the usual SSL encryption of the traffic using the key k_{SSL} and the other is the encryption of the tokens, obtained by tokenizing the traffic, using the key k. This token encryption is performed using the encryption scheme *DPIEnc* introduced by BlindBox.

Detection. MB effciently compares the encrypted rules and received encrypted tokens from the sender using the *BlindBox Detect* algorithm. Everytime an encrypted rule and token match, the payload is dropped and the receiver is informed. This enables middlebox to ensure security even without decrypting the traffic. After completing detection, MB forwards the SSL traffic and the encrypted tokens to the sender.

Receiving traffic. On receiving the payload, the receiver first decrypts the traffic using k_{SSL} and retokenizes and reencrypts the tokens in the *Validate tokens* algorithm to prevent the *attacker at endpoints* threat model.

4 Protocols

BlindBox provides three protocols depending on DPI complexity. Protocol I helps in performing the basic detection, the two other protocols build on this. n Protocol I, a rule consists of one keyword. This protocol suffices for parental filtering but can support only a few IDS rules. In Protocol II, a rule consists of multiple keywords as well as position information of these keywords. This protocol supports a wider class of IDS rules. Protocol III additionally supports regular expressions and scripts, thus enabling a full IDS.

4.1 Protocol I: Single Keyword Detection

An attack rule in this protocol consists of one keyword. We go ahead by describing how each new algorithm introduced by BlindBox works.

4.1.1 Tokenization

The first step in the protocol is to tokenize the input traffic. Two kinds of tokenization scheme can be used, (a) Window-based tokenization and (b) Delimiter-based tokenization. In the window-based method, a specific window length is set and tokenization occurs accordingly, while in delimiter-based method a traffic is tokenized based on the position of delimiters. For example, if the packet stream is "hello?everyone=", the tokens generated by a window of length 8 bytes are "hello?ev","ello?eve", etc. and those generated based on the delimiters "?", "=" are "hello?" and "everyone=".

4.1.2 DPIEnc Encryption Scheme

The sender encrypts the tokens using this encryption scheme and the key k. For every token t, a random salt $salt$ is chosen and a random function H is used to obtain the encryption of a token t as $(salt, H(salt, AES_k(t)))$. Randomizing the encryption is necessary as otherwise every occurence of t will have the same ciphertext weakening security. The typical instantiation of H is SHA-1, but SHA-1 is not as fast as AES, so we implement H with AES keyed with $AES_k(t)$. The algorithm is now entirely implemented in AES, which makes it fast. Thus, the encryption of a token in DPIEnc is:

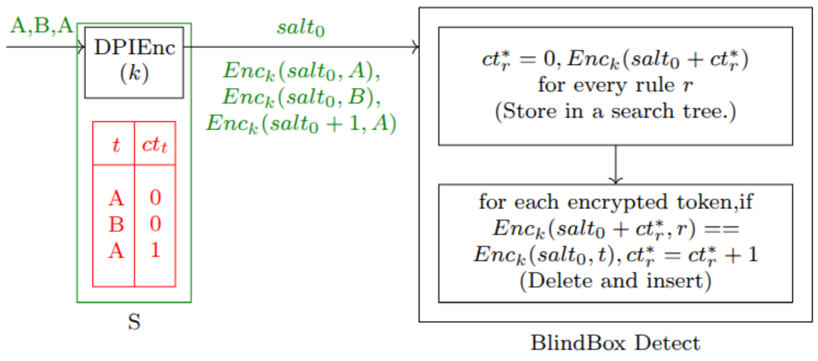
$$salt, AES_{AES_k(t)}(salt) \bmod RS$$

where salt is randomly chosen and RS simply reduces the size of the ciphertext to reduce the bandwidth overhead, but it does not affect security. For simplicity, for the rest of the report we denote $AES_{AES_k(t)}(salt)$ by $Enc_k(salt, t)$. In order to detect a match between a keyword r and an encryption of a token t, MB computes $Enc_k(salt, r) \bmod RS$ using salt and its knowledge of $AES_k(r)$, and then tests if it is equal to $Enc_k(salt, t) \bmod RS$. Hence, MB simply performs a match test for every token t and rule r, which results in a performance per token linear in the number of rules which is too slow. The *BlindBox Detect Protocol* addresses this slowdown by making this cost logarithmic in the number of rules.

4.1.3 BlindBox Detect Protocol

The basic idea would be to calculate $Enc_k(salt, r)$ for every rule r and for every possible salt, arrange the encrypted rules in a search tree. Then for each encrypted token t in the traffic stream, MB simply looks up $Enc_k(salt, t)$ in the tree and checks if an equal value exists. But, enumerating all possible salts for each keyword r is infeasible, what if fewer salts could be used? Reusing the salt for the same token violates security as MB will be able to see which tokens are equal, so to maintain the desired security, every encryption of a token t must contain a different salt (although the salts can repeat across different tokens).

Concretely, the sender maintains a *counter table* mapping each token encrypted so far to how many times it appeared in the stream so far. Instead of sending the salt in the clear along with every encrypted token, the sender sends an initial salt $salt_0$ and every time a token t appears more than once in the traffic, it is encrypted with the salt $(salt + ct_t)$, where ct_t is the number of times t appeared in the stream so far. Note that this provides the desired security because no two equal tokens will have the same salt. To prevent the counter table from growing too large, the sender resets it every P bytes sent. When the sender resets this table, the sender sets $salt_0 \leftarrow salt_0 + max_t ct_t + 1$ and announces the new $salt_0$ to MB.

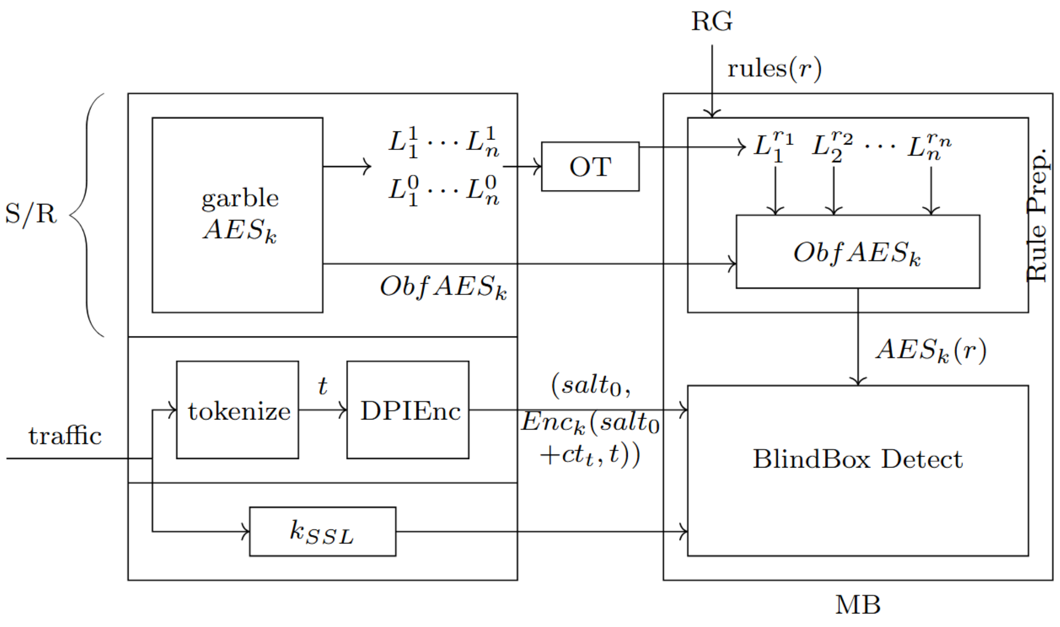

A,B,A
DPIEnc
(k)
t
ct_t
A 0
B 0
A 1
S
$salt_0$
$Enc_k(salt_0, A)$,
$Enc_k(salt_0, B)$,
$Enc_k(salt_0 + 1, A)$
$ct_r^* = 0, Enc_k(salt_0 + ct_r^*)$
for every rule r
(Store in a search tree.)
for each encrypted token,if
$Enc_k(salt_0 + ct_r^*, r) ==$
$Enc_k(salt_0, t), ct_r^* = ct_r^* + 1$
(Delete and insert)
BlindBox Detect

For detection, MB creates a table mapping each keyword r to a counter ct_r* indicating the number of times this keyword r appeared so far in the traffic stream. MB then creates a search tree containing the encryption of each rule r, $Enc_k(salt_0 + ct_r*, r)$. Whenever there is a match to r, MB increments ct_r*, computes and inserts the new encryption $Enc_k(salt_0 + ct_r*, r)$ into the tree, and deletes the old value. The above diagram summarizes the token encryption and keyword detection process along with an example. With this strategy, for every token t, MB performs a simple tree lookup, which is logarithmic in the number of rules. Other tree operations, such as deletion and insertion are also logarithmic in the number of rules.

4.1.4 Rule Preparation

The detection protocol assumes that the MB somehow computes $AES_k(r)$ for every keyword r, everytime a new connection (having a new key k) is setup. But that seems impossible, since the challenge here is that none of the parties, MB or S/R can compute $AES_k(r)$, as the MB knows the keyword but not the key and S/R knows the key but not the keyword. So computing the encrypted rules is the real challenge here which can be accomplished by using a technique called the *obfuscated rule encryption* which is implemented with *Yao garbled circuit*. The idea is that the sender provides the MB an obfuscation $ObfAES_k()$ of the function AES with the key k hardcoded in it, in such a way that the MB runs this obfuscation on the rule r and obtains $AES_k(r)$, without learning the key. While implementing this obfuscation using garbled circuits, MB cannot directly plug in r as input to $ObfAES_k()$, instead it must obtain from the endpoints an encoding of r that works with $ObfAES_k$ in such a way that r is not revealed to the endpoints. This can be achieved with help of the *oblivious transfer(OT)* protocol. Moreover, MB needs to obtain a fresh, re-encrypted garbled circuit for every keyword r as sending more than one encoding for the same circuit to the MB violates security.

Now, another problem is that MB might attempt to run the obfuscation on rules of its choice, as opposed to rules from RG. To prevent this attack, rules from RG must be signed by RG and the obfuscated (garbled) function must check that there is a valid signature on the input rule before encrypting it. If the signature is not valid, it outputs null. Again, based on our threat model, one endpoint could be malicious and attempt to perform garbling incorrectly to eschew detection. Such an attack can be prevented by making both the endpoints prepare the garbled circuit and send it to MB, if the garbled circuits and the labels match, MB is assured that they are correct because atleast one endpoint is honest. So, if the garbled circuits match and the signature of the RG is valid, then only we move forward to the *encrypted rule preparation* and hence to the detection. The below diagram gives a brief overview of the entire scheme, here S/R refers to the process that needs to be be performed by both the endpoints.



Rule preparation is the main performance overhead of BlindBox HTTPS. This overhead comes from the oblivious transfer and from the generation, transmission and evaluation of the garbled circuit, all of which are executed once for every rule.

4.1.5 Validate Tokens

The receiver runs the *validate tokens* procedure in order to check if the other endpoint is malicious. It reruns the tokenization and encrypt modules on the traffic decrypted using k_{SSL} and checks these encrypted tokens are same as those forwarded by the MB. If not, there is a chance that the other endpoint is malicious and flags the behaviour.

4.2 Protocol II: Multi-Keyword and Offset Matching

Protocol II supports a limited form of an IDS and the rules here contain, (1) multiple keywords to be matched in the traffic and (2) absolute and relative offset information within the packet. For example, lets consider rule number 2003296 from the Snort Emerging Threats ruleset and check when the rule will be triggered,

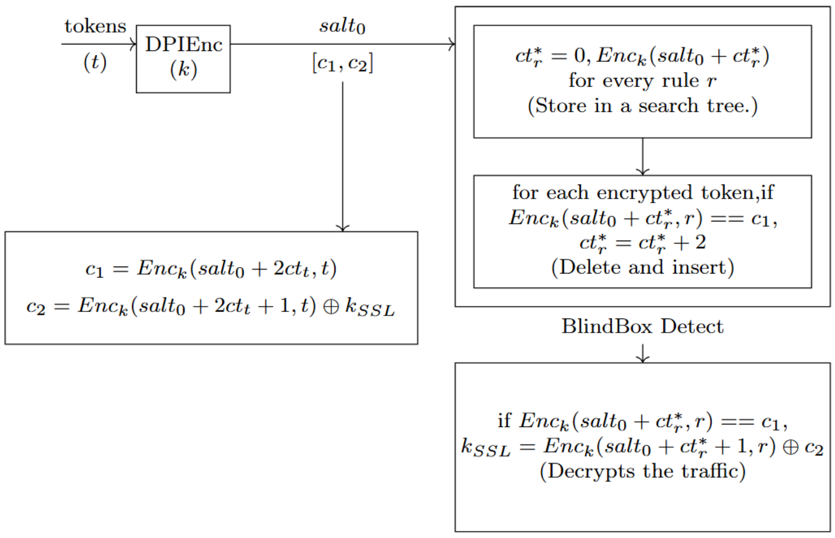
```
alert tcp $EXTERNAL_NET $HTTP_PORTS
                              -> $HOME_NET 1025:5000 (
   flow: established,from_server;
   content: "Server|3a| nginx/0.";
   offset: 17; depth: 19;
   content: "Content-Type|3a| text/html";
   content: "|3a|80|3b|255.255.255.255"; )
```

This rule is triggered if the flow is from the server, it contains the keyword "Server|3a|nginx/0." at an offset between 17 and 19, and it also contains the keyword "ContentType|3a|text/html" and "|3a|80|3b|255.255.255.255". The symbol "|" denotes binary data.

It is clear from the above example that there is a rule match and the packet is dropped if both the keywords and the corresponding offset information matched with the information in the rule packet. So the sender proccesses the stream in the same way as in Protocol I with the exception that the offset in the stream where the token appeared needs to be attached with every encrypted token in delimiter-based tokenization. While in window-based tokenization no offset information needs to be attached because a token is generated at each offset and hence the offset can be deduced. This protocol does not allow arbitrary regular expressions to be run over the payload, which is again dealt with in the next protocol.

4.3 Protocol III: Full IDS Support

Protocol III enables full IDS functionality, including regexp and scripts. In addition to Protocol I & II, this protocol provides *probable cause privacy* by letting the MB decrypt the traffic if a suspicious keyword matches a stream of traffic. However, if there is no such match, the MB should not be able to decrypt the traffic.



tokens
(t)
DPIEnc
(k)
$salt_0$
$[c_1, c_2]$
$ct_r^* = 0, Enc_k(salt_0 + ct_r^*)$
for every rule r
(Store in a search tree.)
for each encrypted token,if
$Enc_k(salt_0 + ct_r^*, r) == c_1$,
$ct_r^* = ct_r^* + 2$
(Delete and insert)
$c_1 = Enc_k(salt_0 + 2ct_t, t)$
$c_2 = Enc_k(salt_0 + 2ct_t + 1, t) \oplus k_{SSL}$
BlindBox Detect
if $Enc_k(salt_0 + ct_r^*, r) == c_1$,
$k_{SSL} = Enc_k(salt_0 + ct_r^* + 1, r) \oplus c_2$
(Decrypts the traffic)

Here, an encrypted token consists of two ciphertexts c_1, c_2, one is the usual as in Protocol I and and the other contains the k_{SSL} *XOR*ed with it. ct_t stands for the same thing as in the first protocol, but here for repeated tokens even counters are used to avoid the confusion with c_2 where odd counters are used. Thus it is clear from the above diagram that the MB can compute the correct k_{SSL} only when there is a successful detection, that is, $Enc_k(salt_0 + 2ct_t, t) = Enc_k(salt_0 + ct_r^*, r)$, failing which MB will not be able to decrypt the traffic correctly.

5 System Implementation

We have two implementations for the protocol. One focuses primarily on Protocol I and the other has gone forward with Protocol III. Here both of them have been explained in detail under Implementaion I and Implementation II respectively. In both implementations, the Client stand for the sender(in theory) and the Server for the receiver.

5.1 Implementation I

Components. (1) Client - Needs to be given a command to wake up, not bound to any port. (2) Middlebox - Docker container which is bind with localhost at port 5001. (3) Server - Docker container which is bind with localhost at port 5002 (4) Rule-generator - Docker container which is bind with localhost at port 5003

Implementation details.
Step 1. (Rule Generator $\rightarrow$ 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 $\rightarrow$ 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 $\leftrightarrow$ 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, (1) k_{SSL} (32 bytes) – for securing the communication channel, (2) k (16 bytes) – for encrypting messages, (3) k_{rand} (16 bytes) – for obfuscation in garbling
Step 4. (Client $\leftrightarrow$ Middlebox) Tokenisation & Salt Exchange: The client connects to the middlebox to exchange tokenisation parameters and a pre-generated salt value. Upon connection,

- The middlebox sends an option string containing tokenisation settings (option type and minimum token length).
- The client displays its current salt (in integer form) and transmits the salt to the middlebox.
- 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 and *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 that 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 and (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, and whenever there is AND gate it is update by +2.

1. **XOR Gate Garbling (Free-XOR)**
   - Given:
     - Input wires: $A(a_0, a_1)$ and $B(b_0, b_1)$
     - Output wire: $C(c_0, c_1)$
   - Process:
     - $c_0 = a_0 \oplus b_0$
     - $c_1 = c_0 \oplus \Delta$ (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: $A(a_0, a_1)$
     - Output wire: $C(c_0, c_1)$
   - Process:
     - $c_1 = a_0 \oplus P$ (P is a fixed public bit-flip mask)
     - $c_0 = c_1 \oplus \Delta$
   - This inverts the logical value without extra encryption.
3. **AND Gate Garbling (Half-Gate Optimization)**
   - Given:
     - Input wires: $A(a_0, a_1)$ and $B(b_0, b_1)$
     - Output wire: $C(c_0, c_1)$

- Process:
  - Extract select bits
    - $p_a = LSB(a_0)$
    - $p_b = LSB(b_0)$
  - First Half-Gate (G) $\rightarrow$ Garbler side
    - Compute hashes of both a_0 and a_1 with a counter.
    - Derive T_G using XOR of these hashes and $p_b * \Delta$.
    - Compute partial output X_G.
  - Second Half-Gate (E) $\rightarrow$ Evaluator side
    - Compute hashes of b_0 and b_1 with counter$+1$.
    - Derive T_E using XOR of these hashes and a_0.
    - Compute partial output X_E.
  - Combine outputs
    - $c_0 = X_G \oplus X_E$
    - $c_1 = c_0 \oplus \Delta$

This step concludes with outputting a **garbled table**. Only two cipher texts $[T_G, T_E]$ are stored. Only needed for AND gate as XOR,INV gates output labels can be generated without tables. Note that here the output labels are generated based on the gate type. So only input labels(key + plain-text) 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 that 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 $label_0, label_1, label_0, label_1$.
Step 10. (Client $\leftrightarrow$ 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 that 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 $\leftrightarrow$ 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.**
4. Process in handle_oblivious_transfer():
   - Key Generation
     - The garbler chooses a random scalar $y \in [1, n-1]$ 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).
   - Send Public Key
     - The garbler sends S (as bytes) to the evaluator.
     - This acts like a temporary public key for the OT session.
   - 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:
       * $k_i0_j = Hash(S, R_{i_j}, y * R_{i_j}) \rightarrow$ for label 0
       * $k_i1_j = Hash(S, R_{i_j}, (y * R_{i_j}) - T) \rightarrow$ for label 1
     - These keys are then XOR-masked with the actual wire labels:
       * $ci_0 = k_i0_j \oplus$ wire_label[offset+j][0]
       * $ci_1 = k_i1_j \oplus$ wire_label[offset+j][1]
   - 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.
5. 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).

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:

1. 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.
2. 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.
3. 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 $\leftrightarrow$ 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 $\leftrightarrow$ 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 $\leftrightarrow$ Middlebox $\leftrightarrow$ Client) Msg transfer: If everything worked fine, then server sends back decrypted message to middlebox and then to client.

5.2 Implementation II

This system is based on Protocol III, utilizing DPIEnc encryption and secure k_{SSL} recovery to ensure security and privacy.

Components

- *Middlebox:* The core DPI engine that performs rule encryption, manages AVL trees for token detection, and processes encrypted traffic.
- *Client:* Responsible for message tokenization, encryption, and providing a web interface for user interaction.
- *Protocol:* Implements BlindBox Protocol III, leveraging DPIEnc encryption and secure recovery of session keys (k_{SSL}).

Key Features and Performance

- *Tokenization:* Processes over 1 million tokens per second for message segmentation.
- *Token Encryption:* The main performance bottleneck, at about 46 tokens per second.
- *Rule Set:* Consists of 4,939 content keywords and 3,880 detailed rules to detect complex malicious patterns.

System setup and usage

Prerequisites: (1) Docker and Docker Compose installed (2) Windows PowerShell or Linux terminal for command execution.
Starting the system: Here is a step by step description of how to start the system:

1. Build Docker images and launch containers:

```
bash
docker-compose up
```

2. This starts three containers:
   - Middlebox container (blindbox-middlebox-1)
   - Client 1 container (blindbox-client1-1)
   - Client 2 container (blindbox-client2-1)
3. Start clients in separate terminals:

```
bash
docker attach blindbox-client1-1
# Press Enter twice
```

```
bash
docker attach blindbox-client2-1
# Press Enter twice
```

4. Access client web interfaces on:
   - Client 1: `http://localhost:5001`
   - Client 2: `http://localhost:5002`
5. Optionally monitor the middlebox and debug logs via Docker commands.
6. Shutdown the system with:

```
bash
docker-compose down
```

Detailed components

Middlebox:

- Written in middlebox.py
- Performs encryption of detection rules.
- Uses AVL tree data structures from avl_tree.py for encrypted token detection.
- Loads rules from JSON files:
  - rule_keywords.json (4,939 keywords)
  - detailed_rules_with_modifiers.json (3,880 complex rules)
- Listens on port 9999 internally.

Client:

- Written in client.py
- Tokenizes and encrypts messages before sending them.
- Uses Flask for a simple web chat interface.
- Runs on ports 5001 (Client 1) and 5002 (Client 2).

File Structure

```
BlindBox/
├── docker-compose.yml          # Container orchestration
├── middlebox/
│   ├── middlebox.py            # DPI engine code
│   ├── avl_tree.py             # AVL tree implementation
│   ├── enhanced_rule_matcher.py # Rule matching logic
│   ├── rule_keywords.json      # Content keywords (4,939
rules)
│   ├── detailed_rules_with_modifiers.json # Complex rules
(3,880)
│   └── Dockerfile              # Middlebox container build
├── client/
│   ├── client.py               # Client app code
│   ├── tokenizer.py            # Tokenization logic
│   ├── templates/index.html    # Web interface HTML
│   └── Dockerfile              # Client container build
├── test_messages.txt           # Test cases for validation
└── README.md                   # Documentation
```

Testing

The system includes categorized test messages to validate functionality:

- Clean messages that should pass through.
- Messages with single or multiple rule matches that should be blocked.
- Edge cases and case sensitivity tests.
- HTTP-based pattern detection.
- Incomplete and complete rule match scenarios.

Security Features

- Utilizes DPIEnc encryption:
  - $AES - ECB$ mode for rule encryption.
  - $AES - GCM$ mode for message encryption.
- Implements Protocol III with secure session key (k_{SSL}) recovery.
- Employs AVL tree to achieve O(log N) encrypted token matching.
- Salt management mechanisms prevent replay attacks and maintain forward secrecy.

Monitoring and Debugging

- Middlebox outputs rule encryption stats and detection metrics on the console.
- Debug logs are maintained in middlebox/debug.txt.
- Real-time log viewing is possible with Docker commands:

```
bash
docker exec -it blindbox-middlebox-1 tail -f
    ↪ debug.txt
```

- Clients output tokenization stats and encryption performance.

Configuration

- Modify middlebox rules by editing rule_keywords.json and detailed_rules_with_modifiers.json.
- Change port configurations in middlebox.py and docker-compose.yml.
- Customize tokenization methods and web interface in client files.

Troubleshooting

- Check container status:

```
bash
docker ps
```

- View logs for specific containers:

```
bash
docker logs blindbox-middlebox-1
docker logs blindbox-client1-1
```

- Restart containers if needed:

```
bash
docker-compose restart middlebox
```

- Verify network port availability:

```
bash
netstat -an | findstr ":5001"
```

Contribution Guidelines

1. Make changes in the relevant directory (middlebox/ or client/).
2. Test with provided test messages.
3. Rebuild containers using:

```
bash
docker-compose build
```

4. Bring the system up and verify:

```
bash
docker-compose up
```

6 Future Directions

Some improvements that can be done in the implementation to make the protocol more efficient are: (1) Optimizing token encryption using Hardware acceleration (AES-NI), Parallel processing or Algorithm optimization, (2) Optimizing network by improving compression and serialization, (3) CO15 protocol is used for OT which is based on elliptic curve which is not safe in quantum systems. Can be upgraded to LWE(Learning with Errors) based OT, (4) Garbled tables are sent to middlebox, can be improved in security by adding random gates or labels mismatching but it is not cost efficient, (5) Key exchange can be modified to use QKD such as BB92 oe E91. Another significant improvement would be to use *Oblivious Pseudorandom Functions(OPRFs)* in place of garbling and OT, for which we may also need to change the way the tokens get encrypted. In additon, using some lattice-based OPRF would render the necessary quantum security.

7 Conclusion

BlindBox shows that privacy and network security can coexist. Through cryptographic innovation and system-level design, it enables middleboxes to operate over encrypted traffic without compromising privacy. This is a critical step toward resolving the tension between policy enforcement and end-to-end encryption in modern networks.

8 Acknowledgemnt

In the entire internship, we have mainly followed the original 2015 paper “BlindBox: Deep Packet Inspection Over Encrypted Traffic” by Sherry et al. Some of the figures used in this report have been taken from the said paper. In addition, for undertanding the idea of *Yao garbled circuit* we have gone through some of the rferences cited in the paper. A few other papers that we have come across in the course of this internship are “PrivDPI”, “SPABox”, etc.