Towards Practical Encrypted Network Traffic Pattern Matching for Secure Middleboxes

Shangqi Lai*, Xingliang Yuan*, Shi-Feng Sun*, Joseph K. Liu*, Ron Steinfeld*, Amin Sakzad*, Dongxi Liu†
*Monash University, Australia, {shangqi.lai,xingliang.yuan,shifeng.sun,joseph.liu,ron.steinfeld,amin.sakzad}@monash.edu
†Data 61, CSIRO, Australia, dongxi.liu@data61.csiro.au

***Abstract*—Network Function Virtualisation (NFV) advances the development of composable software middleboxes. Accordingly, cloud data centres become major NFV vendors for enterprise traffic processing. Due to the privacy concern of traffic redirection to the cloud, secure middlebox systems (e.g., BlindBox) draw much attention; they can process encrypted packets against encrypted rules directly. However, most of the existing systems supporting pattern matching based network functions require tokenisation of packet payloads via sliding windows at the enterprise gateway. Such tokenisation introduces a considerable communication overhead, which can be over 100× to the packet size. To overcome the above bottleneck, in this paper, we propose the first bandwidth-efficient encrypted pattern matching protocols for secure middleboxes. We start from a primitive called symmetric hidden vector encryption (SHVE), and propose a variant of it, aka SHVE+, to enable encrypted pattern matching with constant, moderate communication overhead. To speed up, we devise encrypted filters to further reduce the number of accesses to SHVE+ during matching. We formalise the security of our proposed protocols, and implement a prototype and conduct comprehensive evaluations over real-world rulesets and traffic dumps. The results show that our design can inspect a packet over 20k rules within 100 μs. Compared to prior work, it brings a saving of 94% in bandwidth consumption.**

I. Introduction

Large-scale adoption of Network Function Virtualisation (NFV) facilitates easy realisation, deployment, and management of advanced network functions (aka middleboxes) for enterprises. Under this paradigm, cloud data centres become major NFV vendors [1]. Traditionally dedicated and tightly coupled hardware/software is transformed into composable software middlebox modules, which can run on commodity cloud instances with unlimited scalability. Such a significant technology shift also raises crucial privacy concerns because the traffic of enterprises is re-directed and exposed to cloud data centres [2], [3]. Even HTTPS is widely adopted nowadays, commercial middlebox services intercept and decrypt the encrypted traffic in the middle to retain advanced network functions like deep packet inspection (DPI) [4], [5].

To address this privacy concern and promote secure adoption of NFV, privacy-preserving middlebox systems [6]–[13] have received much attention; these middleboxes can process encrypted traffic against encrypted processing rules without decryption. As a result, both sensitive traffic payloads and proprietary middlebox rules are protected without sacrificing the underlying operations of network functions, such as pattern matching, header inspection, and regular expression. Existing studies in this field can be classified into two categories, i.e., software-based solutions [6]–[9] and hardware-based solutions [10]–[13]. Unfortunately, solutions in neither categories are practically deployable due to efficiency or security issues.

Mainstream software-based solutions [6]–[9] adapt a cryptographic technique named *searchable encryption* [14], which allows middleboxes to match the encrypted patterns extracted from the rules against the encrypted string streams (i.e., tokens) parsed from traffic payloads. However, those designs are communication inefficient. The fundamental reason is that traffic payloads need to be tokenised into string streams via sliding windows with varied sizes (i.e., enumerating the sizes of all patterns). As shown in prior work [6], [7], such cost can be tens of times to the original packet size. Consequently, long latency will be introduced in token transmission, which is not acceptable in wide networked applications. Besides, high I/O consumption between the enterprise and cloud will greatly increase the capital cost of data transfer.

Hardware-based solutions [10]–[13] rely on hardware enclave (i.e., Intel SGX) to execute middlebox functions in a trusted environment. Traffic is fed into the enclave and processed within it. Although using SGX brings benefits on efficiency and functionality for secure middleboxes, recent side-channel attacks against SGX [15]–[17] raise a serious doubt – whether SGX is satisfactory to be deployed in practice.

In order to tackle the above limitations, in this paper, we aim to propose practical cryptographic protocols for a wide range of pattern matching-based secure middleboxes. Our design expects to offer convincing performance in both time and communication towards network environments while ensuring strong protection for rules and traffic payloads.

As mentioned, existing designs based on searchable encryption fall short of achieving bandwidth efficiency. To overcome this bottleneck, we observe that a cryptographic primitive named hidden vector encryption (HVE) [18] is suitable to be a starting point of building a bandwidth-efficient protocol for encrypted network traffic pattern matching. Specifically, HVE generates the key and ciphertext from two same-sized vectors, respectively; HVE decryption can be performed only if the non-wildcard positions of the two vectors are the same. In our context, the packet payload is encrypted from a vector of payload byte stream $(b_1, ..., b_n)$, while the rule pattern is encrypted from a predicate vector $(*, \ldots, *, p_1, ..., p_m, *, \ldots, *)$ with wildcard $(*)$ positions. The offset of the pattern in the above vector is specified by the inspection rule. Later, the middlebox can perform HVE decryption on the encrypted

payload and pattern to check if there is a match. As a result, the communication overhead with HVE becomes constant because traffic is encrypted in byte-wise. For efficiency, we exclude public-key HVE scheme and resort to symmetric-key HVE scheme, aka SHVE [19].

The original SHVE scheme [19] is designed for encrypted membership testing only, where the message is not embedded in the SHVE ciphertext. Thus, it cannot be directly applied to pattern matching-based middlebox functions like DPI, because a DPI rule contains patterns and the corresponding action (e.g., alert, drop), and the entire rule should fully be protected without matching [7], [20]. To solve this problem, we propose a variant of SHVE called SHVE+ which supports both encrypted byte-wise matching and message encryption. Our new primitive preserves the same security guarantees of the existing secure middlebox systems [6], [7]. The equality of byte strings in a packet payload is fully hidden, and the action can be triggered only if a match is found in the encrypted payload at the specified positions in the DPI rules.

To improve efficiency, we design a fine-grained progress filtering protocol to reduce the number of accesses on SHVE+ during the matching process: if a packet is filtered out, i.e., being identified as a mismatch, the middlebox will not continue to process it. As a result, our middlebox saves processing time significantly as most of the packets are commonly legitimate [7], [21], [22]. To apply filtering to encrypted traffic, we propose an encrypted filter structure via SHVE. It is carefully designed in a way that the encrypted packet payload can be used for both filtering and pattern matching. Namely, introducing the filters does not incur extra bandwidth cost.

For completeness, we formalise the security of our proposed protocols. First, we formally capture the capabilities of adversaries considered in the targeted middlebox system. One adversarial model aims to infer sensitive information from encrypted packets. The other aims to deduce information from encrypted rulesets. We note that security analysis of existing designs [6], [7], [9], [23] falls short of capturing the above adversaries at the same time. To bridge the gap, we adapt the real/ideal paradigm to define two groups of games under the above two adversarial models. We prove that even if an adversary is capable of selecting the packet or ruleset to be challenged in advance, she only learns a controlled leakage profile regarding the packet and ruleset.

We implement a prototype and deploy it on a commodity machine. We use real-world patterns (Snort and ETOpen) and network traffic (iCTF08) to evaluate its performance. Regarding latency, our middlebox system can inspect a packet for ETOpen ruleset (20k+ rules) within 100 μs and Snort ruleset (1.5k rules) within 60 μs. In a multi-session scenario (100 concurrent connections), the throughput per connection reaches 5000 packets per second for Snort ruleset and 3000 packets per second for ETOpen ruleset. The overall throughput is over 1 GBps and 500 MBps, respectively. Regarding bandwidth consumption, our design consumes the least bandwidth among all prior arts (including the one in [24] also with constant complexity): it only costs 5 times more bandwidth in terms of the original packet size, which saves more than 94% comparing the designs [6]–[8] using tokenisation. Our approach significantly saves the cost of deploying pattern matching middleboxes in the cloud. The cost estimation based on AWS pricing information demonstrates that the monthly maintenance cost of our middlebox is $\$460.8$, which is only one-fourth of the tokenisation-based approaches.

Our contributions can be summarised as follows:

- We design a new variant of the SHVE scheme called SHVE+, which preserves the functionality, efficiency, and security properties of SHVE while additionally supporting message encryption.
- We propose the first bandwidth-efficient encrypted pattern matching protocol built from SHVE+, which enables middleboxes to perform pattern matching over encrypted traffic with constant, moderate bandwidth overhead.
- We propose a secure filter to filter out legitimate packets, and it further improves the efficiency of middleboxes by $5\times$ to $8\times$. Meanwhile, it does not incur extra bandwidth cost as it reuses the encrypted traffic for pattern matching.
- We are the first to comprehensively formalise the security of encrypted pattern matching protocols for secure middleboxes. We formally prove that our protocol protects against the adversary who wants to compromise traffic and rules throughout pattern matching, respectively.
- We implement a system prototype and evaluate it with real-world rulesets and a traffic dump. We evaluate the setup time, storage overhead, inspection delay, bandwidth overhead, throughput, and deployment cost of our system, and compare them with two prior encrypted pattern matching protocols (i.e., BlindBox [6] and SEST [24]).

II. Related Work

Software-based secure middleboxes. Our work is related to software-based (aka cryptographic) solutions for secure middleboxes. Blindbox [6] is the first system that supports pattern matching based network functions over the encrypted traffic payloads. It is also the first to use searchable encryption for encrypted pattern matching. Later, a line of work is proposed to improve the design of Blindbox, including the realisation of header matching [8], [23], dedicated inspection rule [7] and regular expression [9] support. As mentioned, these designs built from searchable encryption require tokenisation of the payloads, which is a critical performance bottleneck of the system (can lead 77 – $120\times$ bandwidth overhead in terms of the original traffic size). There are some other studies which are built from advanced cryptographic tools. Splitbox [20] uses secure multi-party computation techniques for rule matching, which is also not communication efficient.

Hardware-based secure middleboxes. There are also solutions based on trusted hardware, i.e., Intel SGX. These designs [10]–[13] aim to achieve the same goal of processing encrypted traffic, yet using trusted hardware enclave. As mentioned, Intel SGX is currently vulnerable to side-channel attacks [15]–[17], which can break the security guarantee of the trusted enclave. Besides, deploying those systems requires

TABLE I: Summary of the performance of representative software-based secure pattern matching middleboxes.

Scheme	Communication Cost	Storage Cost	Inspection delay
SEST [24]	High	Medium	ms level
Splitbox [20]	High	Medium	ms level
Tokenisation [6]–[9]	High	Low	μs level
Our middlebox	Low	High	μs level

the cloud servers to be equipped with SGX and enforces the enterprises to trust the hardware vendor. The above constraints would limit the adoption of these SGX-based systems.

Pattern matching on encrypted data. In the literature, some theoretical work also investigates pattern matching on encrypted data. A recent scheme [24] based on cryptographic pairing achieves a constant communication overhead to the packet size. Unfortunately, such a theoretical design still introduces unaffordable bandwidth overhead in practice, i.e., $64\times$ larger than the original traffic. Besides, the pairing based matching operation is too slow to be deployed in traffic processing. A detailed comparison can be found in Section VI.

Some early studies are working on substring matching [25], [26]. Those studies focus on different application scenarios, where a long string is encrypted and stored at the server, and later a substring (pattern) query will be issued to be processed against the long string for matching.

To summarise, we present a comparison table (Table I). It shows that our proposed design outperforms the existing cryptographic works [20], [24] in terms of the communication cost and inspection delay. It highly reduces the communication cost in tokenisation-based approaches [6]–[9] while preserving a microsecond-level inspection delay. Although its storage cost is higher than the other solutions, it is not an issue for in-cloud middleboxes. In networked applications, latency is crucial to user experience and quality of service. The latency of traffic processing is more sensitive to bandwidth, while rule encryption and upload is one-time setup cost. Besides, bandwidth is much more expensive than storage in the modern cloud (see Section VI). Our solution offers a significant maintenance cost saving in the real-world deployment.

III. Overview

A. System Architecture

Our proposed design employs the same architecture as existing secure middleboxes [2], [7], [8], [27] (just to list a few); it redirects an enterprise's traffic from the enterprise gateway to a third-party middlebox service for pattern matching-based packet processing. During this process, the enterprise leverages the middlebox to thoroughly inspect all traffic and enforce its security rules to defend against malicious activities. In addition, the enterprise aims to protect the ruleset in the outsourced environment, because this can either be proprietary ruleset subscribed from professional vendors [6] or customised open-sourced ruleset with private information [7], e.g., enterprise's trade secrets, or intellectual property.

Fig. 1 presents the system architecture[1]. It has two parties:

[1]If an enterprise endpoint connects to an external network, the processed traffic from the middlebox is sent back to the gateway, then sent out [8], [27].

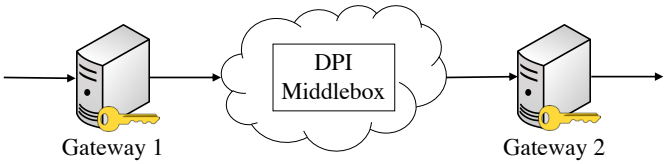


Fig. 1: System Architecture. The arrows indicate traffic from the sender network to the receiver network; the response traffic follows the reverse direction.

the gateway (*GW*) maintained by the enterprise and the middlebox (*MB*) deployed in the service provider, like public clouds. We also use the term "endpoint" to denote the server within the enterprise. The system flow involves three phases:

Initialisation. Before initiating any connection, *GW* randomly chooses a key msk and uses it to generate encrypted rules to be used by *MB* for detecting malicious packet payloads. In practice, each rule describes an attack via its representative patterns, which may include suspicious strings in the payload and the offset information for the string [28]. Each rule also indicates the corresponding actions (e.g., alert, drop) once a match is found. Thus, *GW* creates an encrypted list for the pattern-action list extracted from the ruleset, which is later used for *MB* to match those strings in the encrypted payload and perform the associated action. Meanwhile, *GW* builds an encrypted filter, which can quickly process the mismatches in traffic, and it accelerates the pattern matching process. The generated encrypted filter and pattern list are uploaded to *MB*. Later, *MB* can perform packet inspection for all incoming traffic through the encrypted filter and pattern list.

Preprocessing. *GW* should preprocess the packet payload before sending it for inspection. Specifically, *GW* scans the packet payload in byte-wise and uses msk to generate encrypted traffic dedicated to the pattern matching service like DPI. Then, *GW* will send the encrypted traffic from the enterprise network to *MB*.

Inspection. Upon receiving the encrypted traffic, *MB* withholds the incoming traffic and executes the proposed encrypted pattern matching protocol to inspect the traffic with the precomputed encrypted pattern list. If an action can be recovered after checking the encrypted patterns, *MB* will apply the action to the packet; otherwise, the packet is considered as legitimate, and *MB* then sends it out to the external network. To improve the efficiency of the above process, *MB* exploits a secure filtering protocol. In specific, for each packet, *MB* utilises the pre-built encrypted filter to quickly evaluate whether the current position in the encrypted traffic is a possible matching position. As a result, *MB* separates the innocuous input from the possible malicious traffic, and it only runs the pattern matching protocol for those possible matching positions instead of checking the whole traffic with the encrypted patterns.

Remark. Following prior studies [6], [7] that support pattern matching over the encrypted traffic, the real network traffic is protected by SSL. That is, the sender gateway initialises a normal SSL connection with the receiver and sends the SSL traffic with encrypted traffic to *MB* for inspection. The receiver

can use its SSL session key to recover the real network traffic.

B. Threat Assumption

We assume that *GW* in the enterprise network is a trustworthy party. It follows the proposed protocol and does not disclose the ruleset to other parties. On the other hand, the *MB* service provider is assumed to be semi-honest. It also follows the protocol to offer pattern matching service but attempts to extract sensitive data from the encrypted traffic passing through the middlebox and infer the private ruleset owned by the enterprise. Also, the middlebox can be compromised or eavesdropped as it is deployed in an untrusted environment [6]. Therefore, the main goal of the proposed system is to hide both the content of traffic and the ruleset from *MB* while allowing *MB* to perform pattern matching over the encrypted traffic.

We also assume that at least one endpoint in the communication is honest. This is consistent with the threat model in existing privacy-preserving pattern matching middleboxes [6], [7], [9]. Note that detecting two malicious endpoints is an orthogonal work, and we do not consider this case in our paper.

C. Building Blocks

Basic cryptographic tools. We leverage pseudo-random function PRF, which is a polynomial-time computable function family that is computationally indistinguishable from random functions to any probabilistic polynomial-time adversary. Besides, we make use of symmetric key encryption scheme Sym, which consists of three probabilistic polynomial-time algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$. $\mathsf{KeyGen}(\cdot)$ generates the secret key k. A message m can be encrypted as a ciphertext $c \leftarrow \mathsf{Enc}(k, m)$ and decrypted by $m \leftarrow \mathsf{Dec}(k, c)$. The formal definitions of the PRF and Sym can be found in [19].

SHVE. SHVE [19] is a predicate encryption scheme that supports conjunctive, equality, comparison and subset membership queries over the encrypted data. Compared to the public-key HVE schemes [18], SHVE is much faster as it only relies on the PRF and symmetric key encryption. We present a brief definition of SHVE on below.

Let Σ be an attribute set and $*$ be a wildcard symbol ("don't care" value). We define $\Sigma_* = \Sigma \cup \{*\}$. Let $\mathbf{x} = (x_1, ..., x_n)$ with $x_i \in \Sigma$ be an attribute vector, and $\mathbf{v} = (v_1, ..., v_n)$ with $v_i \in \Sigma_*$ be a predicate vector. The predicate function $P_{\mathbf{v}}(\mathbf{x}) = 1$ if and only if for each $i \in [1, n]$, we have $x_i = v_i$ or $v_i = *$. In other words, the predicate function returns "1" only when the vector $\mathbf{x}$ matches $\mathbf{v}$ in all non-wildcard positions. The SHVE scheme uses a PRF F_0 and the symmetric key encryption Sym as described above. It comprises four probabilistic polynomial-time algorithms:

- $\mathsf{SHVE.Setup}(\lambda)$: On input the security parameter λ, the algorithm outputs the master secret key $msk \xleftarrow{\$} \{0,1\}^\lambda$.
- $\mathsf{SHVE.KeyGen}(msk, \mathbf{v})$: On input the master secret key msk and a predicate vector $\mathbf{v} = (v_1, ..., v_n)$, the algorithm outputs the query trapdoor $\mathbf{s} = (d_0, d_1, S)$, where d_0 is a masked random key, d_1 is a symmetric ciphertext and S keeps all non-wildcard positions in $\mathbf{v}$.
- $\mathsf{SHVE.Enc}(msk, \mathbf{x})$: On input the master secret key msk and an attribute vector $\mathbf{x} = (x_1, ..., x_n)$, this algorithm sets $c_l = F_0(msk, x_l||l)$ for each $l \in [n]$, and outputs the ciphertext $\mathbf{c} = (\{c_l\}_{l \in [n]})$.
- $\mathsf{SHVE.Query}(\mathbf{s}, \mathbf{c})$: The query algorithm takes as input a trapdoor $\mathbf{s}$ and a ciphertext $\mathbf{c}$. If the algorithm recovers 0 from $\mathbf{s}$ and $\mathbf{c}$, the query algorithm outputs "True" (indicating $P_{\mathbf{v}}(\mathbf{x}) = 1$) else it outputs $\perp$.

IV. The Proposed System

A. Construction of SHVE+

SHVE [19] can be adapted to achieve efficient encrypted pattern matching for network traffic. However, it cannot be directly used for middlebox functions like DPI. As mentioned in Section III-A, the inspection rule consists of the inspection patterns and corresponding action. To fully protect the rules during the matching process, both of them should be encrypted. Also, to preserve the functionality, the action needs to be recovered for *MB* further processing when the pattern is matched. To this end, the action should be considered as a message encrypted with the pattern in SHVE. Similar to the design in prior work [7], [20], the above design encrypts both the rule and action to minimise the leakage in the outsourced middlebox. Nonetheless, we also take the performance into consideration and choose to reveal the action for those matched patterns. This trade-off enables our middlebox to efficiently and securely handle a large volume of packets at a moderate cost and well-defined leakage. We note that the original SHVE construction [19] can only be used for membership testing, whereas the message encryption is yet to be supported. To address this issue, this section presents a new SHVE scheme, dubbed SHVE+, which enables message encryption on SHVE.

Construction. The original SHVE (see Section III-C) leverages a random key K to encrypt "0" in the SHVE trapdoor (d_1 in the trapdoor), and it refers to the predicate vector $\mathbf{v}$ to masks the random key and keeps the masked key in d_0. If $\mathbf{v}$ matches the attribute vector encrypted in the ciphertext $\mathbf{c}$ at all non-wildcard positions, the encrypted "0" can be recovered from the trapdoor after $\mathsf{SHVE.Query}$, and SHVE outputs "True". Intuitively, we can exploit the d_1 term in the trapdoor to store the other encrypted message. Then, $\mathsf{SHVE+.Query}$ is changed to return the decryption of d_1 after decrypting d_1 successfully.

We now present the details of our SHVE+ construction. Note that only the modified algorithms are given here, the other algorithms remain the same as in Section III-C.

- $\mathsf{SHVE+.KeyGen}(msk, \mathbf{v}, m)$: On input the master secret key msk, a predicate vector $\mathbf{v} = (v_1, ..., v_n)$ and a message m, the algorithm extracts all non-wildcard positions $S = \{l \in [n] | v_l \neq *\}$ from $\mathbf{v}$. Let these positions be $l_1 < \ldots < l_{|S|}$, the algorithm samples $K \xleftarrow{\$} \{0,1\}^\lambda$ and computes: $d_0 = \oplus_{j \in [|S|]} \left(F_0(msk, v_{l_j}||l_j)\right) \oplus K$, $d_1 = \mathsf{Sym.Enc}\,(K, m)$ Finally, it outputs the trapdoor $\mathbf{s} = (d_0, d_1, S)$ corresponding to the predicate vector $\mathbf{v}$.
- $\mathsf{SHVE+.Query}(\mathbf{s}, \mathbf{c})$: The query algorithm takes as input a trapdoor $\mathbf{s}$ and a ciphertext $\mathbf{c}$. Then, it computes $K' = \left(\oplus_{j \in [|S|]} c_{l_j}\right) \oplus d_0$ and returns $\mu = \mathsf{Sym.Dec}\,(K', d_1)$.

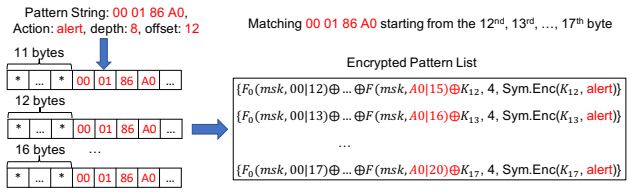


Fig. 2: An example of the encrypted pattern generation: each pattern string is inserted into multiple matching arrays to match the pattern in every possible position in the payload.

Algorithm 1 Encrypted Rule Generation

Input: The master secret key msk from SHVE+.Setup; the ruleset R
Output: The encrypted pattern list $\mathcal{E}$

function GENERATE(msk, R)
  Parse R as a pattern-action list $T = \{(pat, act)\}$
  for each (pat, act) in T **do**
    $start \leftarrow pat.\text{offset} > 0 ? pat.\text{offset} : 1$
    $end \leftarrow pat.\text{depth} > 0 ? start + pat.\text{depth} : 1500$
    for $i = start : end - pat.\text{string}.len + 1$ **do**
      $\mathbf{v}_i \leftarrow *^{1500}$
      Insert $pat.\text{string}$ at $\mathbf{v}_i[i]$
      $\mathbf{t}_i \leftarrow \text{SHVE+.KeyGen}(msk, \mathbf{v}_i, act)$
      Store $\mathbf{t}_i$ in $\mathcal{E}$
  return $\mathcal{E}$

Under the SHVE+ scheme, the proposed system can encrypt the pattern as a SHVE+ trapdoor and then encrypt traffic as the SHVE+ ciphertext to make the inspection. In specific, *GW* in the proposed system generates a pattern array initialised with wildcard character '$*$' in all positions. Then it inserts each byte of the pattern string into the pattern array according to the rule (string content, start/end position), and uses the array as the predicate vector and the action as the message to compute the encrypted pattern via SHVE+.KeyGen. Later, *GW* parses the traffic into a byte array and uses it as the attribute vector to get the encrypted traffic by SHVE+.Enc. Finally, on *MB*, the encrypted pattern can examine traffic in the form of SHVE+ ciphertext and properly recover the action if a match is found according to the definition of SHVE [19].

Security. SHVE+ retains SHVE's security properties for membership testing, which guarantees that the pattern matching process only reveals whether the encrypted traffic includes the pattern in the position specified by rules, but nothing more. Moreover, it ensures that the message can only be recovered when the traffic matches the pattern, which is consistent with the security requirement of the proposed middlebox service. A detailed analysis is given in Section V.

B. The Proposed Encrypted Pattern Matching Protocol

In order to support secure pattern matching over the encrypted traffic, the existing work [6], [7] leverages an encrypted index built from the pattern-action list. More specifically, the encrypted index is indexed by the encryption of each pattern string. When a given inspection token matches the encrypted indexing term, *MB* can recover the action from the index and execute it. However, due to the complexity of matching patterns (various size, matching position, etc.), this approach has to tokenise the original packet payload into a large number of tokens, and it can blow up the bandwidth consumption ($24\times$ as reported in [6]). To enable pattern matching in a bandwidth-saving manner, our system is built from SHVE+ because it does not rely on any tokenise algorithm. Instead, it encrypts the payload and queries the pattern in byte-wise. Consequently, its bandwidth consumption is a constant no matter how long the pattern is (see Section IV-A).

Algorithm 2 Rule Matching

Input: *GW* inputs the master secret key msk, the payload P; *MB* inputs the encrypted pattern list $\mathcal{E}$;

function MATCH($msk, \mathcal{E}, P$)
On *GW*:
  Parse P as a byte array and compute the encrypted traffic $\mathbf{c} \leftarrow \text{SHVE+.Enc}(msk, P)$
  Send $\mathbf{c}$ to *MB*
On *MB*:
  for each encrypted pattern $\mathbf{t}$ in $\mathcal{E}$ **do**
    $act' \leftarrow \text{SHVE+.Query}(\mathbf{t}, \mathbf{c})$
    Execute act' if it is valid

Pattern matching for arbitrary pattern strings. Algorithm 1 summarises the detailed encrypted rule generation procedure run by *GW*. As mentioned, each inspection rule is parsed as a pattern-action tuple, and our protocol generates the encrypted pattern list from it. In practice, the inspection rules often involve qualifiers that specific a range of positions to be checked in the packet payload. For example, the following Snort rule [28] specifies the "depth" (only search 8 bytes instead of 1500 bytes for the pattern) and "offset" (start to search the pattern from the 12th byte of the payload).

```
alert udp $EXTERNAL_NET any -> $HOME_NET 111
(flow:to_server; content:"|00 01 86 A0|",
depth 8,offset 12)
```

Thus, our protocol takes the above two qualifiers into consideration when generating pattern arrays and encrypted patterns. As shown in Fig. 2, our protocol first generates offset $-$ string.$len+1$ pattern arrays with wildcard character '$*$'. Then, it inserts the pattern string into the pattern arrays at each possible starting positions, which is 12 to 17 in our example. For each pattern array, our protocol inputs the action and runs SHVE+.KeyGen to encrypt the pattern string and position after concatenating them together (see Section IV-A). This ensures that matches only happen on the positions specified by the rule. The result encrypted pattern list is able to match the pattern in the specific positions over any incoming traffic from *GW*.

The above protocol supports encrypted pattern matching in wildcard positions, i.e., the pattern can be found in all positions in a packet. For this case, our protocol generates encrypted patterns for all positions to find matches in traffic. Due to the MTU restriction, the maximum payload size is 1500 bytes, which means that each rule needs 1500 encrypted patterns at most to match all position. Note that the size of each encrypted pattern is a constant (see Fig. 2) without regarding the length of original pattern strings. Moreover, it is a tiny data structure: each encrypted pattern is only 23 bytes (see Section VI).

The matching process is outlined in Algorithm 2. After uploading the encrypted pattern list to *MB*, *GW* generates the encrypted traffic from the packet payload via SHVE+.Enc.

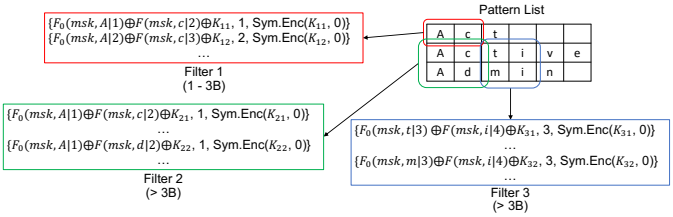


Fig. 3: The proposed filter structure

Later, *MB* uses the pattern list to check the traffic from *GW* via SHVE+.Query. If a target packet includes a required pattern, *MB* can recover an action and apply it to the packet.

Matching packet header and regular expression. The protocol can be extended to support packet header inspection. In particular, the header inspection focuses on the field information (e.g., HTTP header, HTTP method), which can only be found in the specific place in the header. Thus, the protocol can parse the field information by extracting the field value as the pattern string and referring the header structure to compute the positional information. Finally, the processed header inspection rule can be used by the original protocol to tackle with patterns that appear in the header.

For regular expression matching, it is common for the real-world pattern matching system like Snort to parse the regular expression as sub-strings and apply pattern matching algorithms to check those sub-string respectively [21], [22]. For instance, the regular expression "ap*e" aims to find the string start with 'ap' and end with 'e'. The pattern matching system checks 'ap' and 'e' separately and returns match if the matching position of 'e' is behind the one for 'ap'. Therefore, our protocol can follow the same strategy to check the regular expression for the encrypted traffic. That is, the protocol generates encrypted patterns for 'ap' and 'e' separately and leverages the secret sharing scheme in [7] to share the action into two encrypted patterns, and the action can only be recovered when two encrypted patterns are matched orderly.

Remark. The security properties of SHVE+ guarantee that the action can only be recovered if the encrypted payload includes the pattern (i.e. both the position and string should be matched). Also, SHVE+ ensures that the equality of byte strings in the packet is not revealed to *MB*, because the SHVE+ combines the packet payload and positions when generating the encrypted payload. Thus, the ciphertext of two identical bytes is different if they are in the different position of the payload. A detailed security proof is given in Section V.

Regarding the efficiency of the basic matching protocol, for each pattern, it performs a byte-to-byte match on the incoming traffic. Recall that the protocol generates multiple encrypted patterns to match all specified starting positions of patterns in the traffic, its performance can be optimised via parallel processing. In particular, the middlebox can use those independent encrypted patterns to perform pattern matching in specified positions concurrently on the encrypted traffic. However, the drawback of this basic matching protocol is that its performance may degrade rapidly with the increasing size of the ruleset. That is because each newly-added rule can have

Algorithm 3 Encrypted Filter Generation

Input: The master secret key msk from SHVE.Setup; the ruleset R
Output: The encrypted filter $\mathcal{F} = \{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3\}$

function FILTERGENERATE(msk, R)
  Parse R as a pattern-action list $T = \{(pat, act)\}$
  for each pat.string in T **do**
    $start \leftarrow pat.\text{offset} > 0 ? pat.\text{offset} : 1$
    $end \leftarrow pat.\text{depth} > 0 ? start + pat.\text{depth} : 1500$
    $s_1 \leftarrow pat.\text{string}.substring(1, 2)$
    for $i = start : end - pat.\text{string}.len + 1$ **do**
      $\mathbf{v}_i \leftarrow *^{1500}$
      Insert s_1 at $\mathbf{v}_i[i]$
      $\mathbf{t}_i \leftarrow \text{SHVE.KeyGen}(msk, \mathbf{v}_i)$
      if $pat.\text{string}.len \leq 3$ **then**
        Store $\mathbf{t}_i$ in F_1
      else
        Store $\mathbf{t}_i$ in F_2
        $s_2 \leftarrow pat.\text{string}.substring(3, 2)$
        $j = i + 2$
        $\mathbf{v}_j \leftarrow *^{1500}$
        Insert s_2 at $\mathbf{v}_j[j]$
        Store $\text{SHVE.KeyGen}(msk, \mathbf{v}_j)$ in F_3
  return $\mathcal{F}$

up to 1500 more corresponding encrypted patterns. It indicates that *MB* may need to perform 1500 more SHVE+.Query on a given packet if the size of the ruleset increased by one. Next, we will introduce a secure filter to address the above issue.

C. Secure Filtering

One key observation is that only a small fraction of the traffic includes malicious payloads (less than 0.01% as shown in [21]). Consequently, if we can efficiently distinguish the innocuous traffic from the malicious one, and only do pattern matching on the malicious traffic, the performance of the overall middlebox system can be highly improved. To achieve our goal, we propose a secure filter system that can quickly evaluate whether the packet includes a match and where is the possible starting position to match. Also, the filter is encrypted to prevent *MB* from learning any private information about the traffic and ruleset as in Section III-B.

The proposed filter consists of three filters that run in two-level (see Fig. 3). The first level has two filters: Filter 1 stores information about the pattern strings that less than 4 characters (bytes), while Filter 2 accounts for the longer patterns. Both of them keep an encrypted pattern list of the beginning two bytes of each pattern string; it also combines the position information to check all position in the packet. The filter in the second level (Filter 3) works together with Filter 2; it is a progressive filter generated from the next 2 bytes in the pattern string. The progressive filter matches the following two bytes in each pattern string if it matched in Filter 2, and it reduces the false positive rate when matching a longer pattern. Note that similar design philosophy is also adapted in plaintext traffic pattern matching systems [21], [22].

Algorithm 3 presents the steps of building the encrypted filter. For each pattern string, *GW* extracts the first two characters and generates encrypted patterns via SHVE. Then, it inserts the encrypted patterns into either F_1 or F_2 by referring the length of the pattern string. For those longer patterns

Algorithm 4 Secure Filtering

Input: *GW* inputs the master secret key msk, the payload P; *MB* inputs the encrypted filter $\mathcal{F}$
Output: A list of possible matching positions M

```
1: function FILTERING(msk, F, R)
2: On GW:
3:     Parse P as a byte array and compute the encrypted traffic c ← SHVE.Enc(msk, P)
4:     Send c to MB
5: On MB:
6:     for i = 0 to F_1.len do
7:         if SHVE.Query(F_1[i], c) ="True" then
8:             Add F_1[i].S to M
9:     if c.len > 3 then
10:        for i = 0 to F_2.len do
11:            if SHVE.Query(F_2[i], c) ="True" then
12:                for j = 0 to F_3.len do
13:                    if SHVE.Query(F_3[j], c) ="True" then
14:                        Add F_2[i].S to M
15:    return M
```

(more than 3 bytes), the next two bytes are also generated as encrypted patterns and stored in F_3. Finally, *GW* uploads $\mathcal{F}$ with the above three sub-filters to *MB* as the encrypted filter.

To execute the secure filtering algorithm (cf. Algorithm 4), *MB* reuses the encrypted traffic to check the encrypted filter. In specific, as SHVE supports secure membership testing, *MB* is capable of recovering a "True" after SHVE.Query if the upcoming payload has two bytes that match the ruleset pattern. After applying the secure filtering, *MB* only requires to check the position that returns "True" when running the following pattern matching process. Hence, the secure filtering can highly boost the overall pattern matching procedure, because for each rule, instead of using all encrypted patterns to check the whole encrypted traffic, only a few patterns corresponding to the filtered positions need to be checked.

Filtering in parallel. The secure filtering relies on two separate groups of filters (filters for pattern ≤ 3 bytes and > 3 bytes). Therefore, we can use the output to check the encrypted pattern corresponding to the pattern string ≤ 3 bytes and > 3 bytes, respectively. This can reduce the workload in the pattern matching process further because only the pattern that fits the size requirement needs to be checked after adopting this optimisation. To achieve this, we slightly modify Algorithm 4: The matching positions output from $\mathcal{F}_1$ and $\mathcal{F}_3$ are kept in two matching position lists (M_1 and M_2). Also, we employ two separate buckets to store the encrypted patterns for the pattern strings ≤ 3 bytes and > 3 bytes separately. As a result, *MB* can use the position information in M_1 to check the short patterns while utilising M_2 to check those longer patterns.

V. Security Analysis

We give a security analysis to demonstrate that *MB* cannot learn the sensitive data in the ruleset as well as traffic during the pattern matching process. We are the first to formalise the adversary capability in two aspects: 1) The adversary can select the packet to be challenged and get the encrypted patterns and filter selected by himself/herself. The goal of the adversary is to learn the sensitive data in the packet; 2) The adversary can select the ruleset to be challenged and get the encrypted packet chosen by himself/herself. The adversary aims to learn information about the ruleset other than the pattern matching result. Note that the existing work only considers either the security of the packet [6], [9] or the security of the ruleset [7], [23].

We follow the simulation-based security [19] to define a leakage function $\mathcal{L}$ for our encrypted pattern matching protocol $\mathcal{P}$ and then construct a simulator to show that $\mathcal{P}$ is $\mathcal{L}$-secure against adversaries as described in above. More specifically, we construct a simulator $\mathcal{S}$ and prove that $\mathcal{S}$ can simulate $\mathcal{P}$ by using the leakage function $\mathcal{L}$ only. This implies that the proposed protocol does not reveal any information about the packet payload and rules beyond the leakage function.

Security of SHVE+. SHVE+ has a similar security model as SHVE [19] except that SHVE+ has a non-empty message space to support message encryption. Recall that the security model of SHVE defines the *attribute-hiding* property, which indicates that the adversary can only learn two leakage functions: $\alpha(\mathbf{v})$ representing the wildcard pattern (positions) of a given predicate vector $\mathbf{v}$, and $\beta(\mathbf{v}, \mathbf{x})$ describing the leakage after queries (i.e., leaking whether $\mathbf{v}$ and $\mathbf{x}$ are matched or not). An adversary can arbitrarily request the HVE trapdoors given the above two leakage functions to $\mathcal{S}_{\mathsf{SHVE}}$. However, no more information about $\mathbf{v}$ and $\mathbf{x}$ will be leaked. The following theorem from [19] states the security of SHVE:

Theorem 1: SHVE is attribute-hiding in the ideal cipher model under the security model defined in above.

We keep the $\alpha(\mathbf{v})$ unchanged because the wildcard pattern of SHVE+ is exactly the same as in SHVE, while the definition of $\beta(\mathbf{v}, \mathbf{x})$ is modified as follows: $\beta^{'}(\mathbf{v}, \mathbf{x}) = m$ if $P_{\mathbf{v}}(\mathbf{x}) = 1$, otherwise $\beta^{'}(\mathbf{v}, \mathbf{x}) \xleftarrow{\$} \{0,1\}^{\lambda}$. The following theorem states the security of SHVE+:

Theorem 2: SHVE+ is attribute-hiding in the ideal cipher model under the security model defined in above.

We omit the proof of Theorem 2 because it is identical to the one in [19]. In the rest of this section, we directly apply the simulator of SHVE+ when simulating $\mathcal{P}$.

Security of the pattern matching protocol. Let $\mathcal{P}$ be the pattern matching protocol. The security of $\mathcal{P}$ is formally defined via two groups of real/ideal game definitions. The first real/ideal game definition depicts the security of $\mathcal{P}$ against the adversary $\mathcal{A}_1$ who aims to compromise the confidentiality of the encrypted packets. This adversary is identical to the one in [4] who entrenches in *MB* and can get any number of the encrypted patterns and filter tokens to examine the encrypted packet. The following games and theorem state that $\mathcal{P}$ can protect the packet confidentiality in the presence of $\mathcal{A}_1$:

- $\mathbf{Real}^{\mathcal{P}}_{\mathcal{A}_1}(\lambda)$: The adversary $\mathcal{A}_1$ chooses a packet P for the game to generate the encrypted packet $\mathbf{c}$ and gives $\mathbf{c}$ to $\mathcal{A}_1$. Then, $\mathcal{A}_1$ adaptively chooses a rule r to query. To respond, the game GENERATE(msk, r) and FILTERGENERATE(msk, r). Later, the game gives the protocol outputs to $\mathcal{A}_1$. Eventually, $\mathcal{A}_1$ outputs a bit.
- $\mathbf{Ideal}^{\mathcal{S}}_{\mathcal{A}_1}(\lambda)$: The game initialises a counter $i = 0$ and an empty rule list R. The adversary $\mathcal{A}_1$ chooses a packet

P, and the game runs $\mathbf{c} \leftarrow \mathcal{S}(\mathcal{L}_1(P))$ and gives the encrypted packet $\mathbf{c}$ to $\mathcal{A}_1$. Then, $\mathcal{A}_1$ adaptively chooses a rule r to query. To respond, the game records the rule as $R[i]$, and gives the output of $\mathcal{S}(\mathcal{L}_1(P, R))$ (R keeps all history rules) to $\mathcal{A}_1$. Later, the game increases i by 1. Eventually, $\mathcal{A}_1$ outputs a bit.

Theorem 3: $\mathcal{P}$ is $\mathcal{L}_1$-secure against $\mathcal{A}_1$, assuming that the SHVE+ scheme is selectively simulation-secure, that the SHVE scheme is selectively simulation-secure.

Proof: First, we describe the leakage function $\mathcal{L}_1$ towards $\mathcal{A}_1$. On input a packet P and the adaptively chose ruleset R, the leakage function can be parameterised as $\mathcal{L}_1(P, R) = \{\mathsf{Pos_f}, \mathsf{Pos_e}, \mathsf{Act}\}$ formed as follows:

- $\mathsf{Pos_f}$ is the possible matching positions in P w.r.t. R. Formally, $\mathsf{Pos_f}[i]$ is an array of possible matching positions in P w.r.t. $R[i]$.
- $\mathsf{Pos_e}$ is the matched positions in P w.r.t. R. Formally, $\mathsf{Pos_e}[i]$ is an array of matched positions in P w.r.t. $R[i]$.
- Act is the actions that need to be performed on P. Formally, if $R[i]$ is matched in P, $\mathsf{Act}[i] = R[i].act$, otherwise, $\mathsf{Act}[i] = \perp$.

Next, we show that we could combine the above leakage function, $\mathcal{S}_{\mathsf{SHVE}}$ and $\mathcal{S}_{\mathsf{SHVE+}}$ to simulate $\mathcal{P}$. Suppose $\mathcal{A}_1$ provides a packet P to $\mathcal{S}_1$. $\mathcal{S}_1$ invokes $\mathbf{c} \leftarrow \mathcal{S}_{\mathsf{SHVE+}}(\lambda)$ to generate the ciphertext of P and gives it to $\mathcal{A}_1$. Upon receiving the i-th rule from $\mathcal{A}_1$, $\mathcal{S}_1$ refers to $\mathsf{Pos_f}[i]$ to simulate the filter. In specific, for each possible matching position $p_f \in \mathsf{Pos_f}[i]$, $\mathcal{S}_1$ sets $\{p_f, p_f + 1\}$ as $\alpha_{i1}(\mathbf{v}_{i1})$ and $\beta_{i1}(\mathbf{v}_{i1}, P)$ as "True". If $|R[i]| \geq 3$, $\mathcal{S}_1$ additionally sets $\{p_f + 2, p_f + 3\}$ as $\alpha_{i2}(\mathbf{v}_{i2})$ and $\beta_{i2}(\mathbf{v}_{i2}, P)$ as "True". Then, it runs $\mathcal{S}_{\mathsf{SHVE}}(\alpha_{i1}(\mathbf{v}_{i1}), \beta_{i1}(\mathbf{v}_{i1}, P))$ and $\mathcal{S}_{\mathsf{SHVE}}(\alpha_{i2}(\mathbf{v}_{i2}), \beta_{i2}(\mathbf{v}_{i2}, P))$ (if $|R[i]| \geq 3$) to get the corresponding filter. Similarly, for each matched position $p_e \in \mathsf{Pos_e}[i]$, $\mathcal{S}_1$ sets $\alpha_i(\mathbf{v}_i) = \{p_e, ..., p_e + |R[i]| - 1\}$ and $\beta_i^{'}(\mathbf{v}_i, P) = \mathsf{Act}[i]$. $\mathcal{S}_1$ then calls $\mathcal{S}_{\mathsf{SHVE+}}(\alpha_i(\mathbf{v}_i), \beta_i^i(\mathbf{v}_i, P))$ to get the token for $R[i]$. $\mathcal{A}_1$ finally receives the simulated filter and encrypted pattern corresponding to $R[i]$.

It is obvious that the simulated ciphertext is indistinguishable from the real ciphertext as it is computed via PRF. Additionally, Theorem 1 and Theorem 2 directly ensure that the simulated trapdoors for the filter and encrypted patterns are indistinguishable from the real trapdoors generated by $\mathbf{Real}^{\mathcal{P}}_{\mathcal{A}_1}(\lambda)$. Thus, it concludes that for every adversary $\mathcal{A}_1$, it has a negligible probability to learn more information from P than the defined leakage function $\mathcal{L}_1$. ∎

Theorem 3 shows that the adversary cannot infer any information about the packet beyond $\mathcal{L}_1$ after receiving the encrypted pattern and filter. It indicates that *MB* cannot know any information about a legitimate packet as it does not match any rule. Meanwhile, to fulfil the requirement of pattern matching middleboxes, *MB* is allowed to learn the matching information and action on a malicious packet. The revealed information enables the pattern matching middlebox to apply the inspection rule on the malicious packet efficiently.

We also consider the adversary $\mathcal{A}_2$ who wants to learn unintended information from the ruleset, which captures the capacity of adversaries either in endpoints. Similar to the adversary in [7], $\mathcal{A}_2$ is able to use arbitrary packet payload to examine the ruleset deployed on *MB*. The following games and theorem indicate the security guarantee on the ruleset in the presence of $\mathcal{A}_2$:

- $\mathbf{Real}^{\mathcal{P}}_{\mathcal{A}_2}(\lambda)$: The adversary $\mathcal{A}_2$ chooses a ruleset R and a packet list $\mathbf{q}$. Then, the game runs $\textsc{Generate}(msk, R)$ and $\textsc{FilterGenerate}(msk, R)$ to generate the encrypted pattern $\mathcal{E}$ and filter $\mathcal{F}$. Later, the game runs $\textsc{Filtering}(msk, \mathcal{F}, \mathbf{q}[i])$ and $\textsc{Match}(msk, \mathcal{E}, \mathbf{q}[i])$ for $1 \leq i \leq |\mathbf{q}|$. The ciphertext $\mathbf{c_i}$ of $\mathbf{q}[i]$ and pattern matching results are stored in $\mathbf{t}[i]$ as the transcript. The generated $\mathcal{E}$, $\mathcal{F}$ and $\mathbf{t}$ are given to $\mathcal{A}_2$. Eventually, $\mathcal{A}_2$ outputs a bit.
- $\mathbf{Ideal}^{\mathcal{S}}_{\mathcal{A}_2}(\lambda)$: The adversary $\mathcal{A}_2$ chooses a ruleset R and a packet list $\mathbf{q}$. Then, the game runs $\{\mathcal{E}, \mathcal{F}, \mathbf{t}\} \leftarrow \mathcal{S}_2(\mathcal{L}_2(R, \mathbf{q}))$ and gives the outputs to $\mathcal{A}_2$. Eventually, $\mathcal{A}$ outputs a bit.

Theorem 4: $\mathcal{P}$ is $\mathcal{L}_2$-secure against $\mathcal{A}_2$, assuming that the SHVE+ scheme is selectively simulation-secure, that the SHVE scheme is selectively simulation-secure.

Proof: We start with the definition of $\mathcal{L}_2$: Let $\mathbf{q}[i], 1 \leq i \leq |\mathbf{q}|$ be a query packet and $R[j], 1 \leq j \leq |R|$ is a rule in the given ruleset. We have $\mathcal{L}_2(R, \mathbf{q}) = \{\mathbf{r}, |\mathcal{E}|, |\mathcal{F}|, \mathsf{Match_P}, \mathsf{Match}, \mathsf{Act}, \mathsf{IP}\}$ formed as follows:

- $\mathbf{r}$ is an array storing the length of each rule, i.e., $\mathbf{r}[i]$ is the length of $R[i]$.
- $|\mathcal{E}|$ is the size of the encrypted pattern list.
- $|\mathcal{F}|$ is the size of the encrypted filter.
- $\mathsf{Match_P}$ is the possible match position pattern of each packet $\mathbf{q}[i]$ w.r.t. each rule $R[j]$, which is a bidimensional array: $\mathsf{Match_P}[i, j]$ is all positions in $\mathbf{q}[i]$ that probably match $R[j]$.
- Match is the matched position pattern of each packet $\mathbf{q}[i]$ w.r.t. each rule $R[j]$, which is a bidimensional array: $\mathsf{Match}[i, j]$ is all positions in $\mathbf{q}[i]$ that match $R[j]$.
- Act is the action pattern of each packet $\mathbf{q}[i]$ w.r.t. each rule $R[j]$, which is a bidimensional array. If $R[j]$ matches in $\mathbf{q}[i]$, $\mathsf{Act}[i, j] = R[j].act$, otherwise, $\mathsf{Act}[i, j] = \perp$.
- IP is the intersection pattern of any two rules $R[j], R[k], 1 \leq j, k \leq |R|, j \neq k$ matched the packet $\mathbf{q}[i]$, which is a three dimensional array. Particularly, $\mathsf{IP}[i, j, k]$ stores the intersection position of $R[j], R[k]$.

Next, we show how to simulate $\mathcal{P}$ via $\mathcal{L}_2$, $\mathcal{S}_{\mathsf{SHVE}}$ and $\mathcal{S}_{\mathsf{SHVE+}}$. First, $\mathcal{S}_2$ initialises a bi-dimensional array A to store the auxiliary information for the simulation. Then, $\mathcal{S}_2$ leverages the outputs from IP and Match to simulate the encrypted packet. In specific, for each query packet $\mathbf{q}[i], 1 \leq i \leq |\mathbf{q}|$:

1) $\mathcal{S}_2$ generates an empty array $\mathbf{c_i}$.
2) For each rule $R[j], 1 \leq j \leq |R|$:
   • For each matched position $pos \in \mathsf{Match}[i, j]$ of $R[j]$, $\mathcal{S}_2$ checks whether $A[R[j], l], pos \leq l \leq pos + \mathbf{r}[j] - 1$ is set as the wildcard symbol.
   • If any of the above value in $A[R[j], l]$ is wildcard and $\forall \mathsf{IP}[i, j, k] = \emptyset, 1 \leq k \leq |R|, k \neq j$, then $\mathcal{S}_2$ simulate a

ciphertext $A[R[j],l] \xleftarrow{\$} \{0,1\}^{\lambda}$.
- Otherwise, if $\exists \mathsf{IP}[i,j,k] \neq \emptyset$, $A[R[j],l] = A[R[k],l]$.
- If $\mathbf{c_i}[l]$ is not set, $\mathcal{S}_2$ sets $\mathbf{c_i}[l] = A[R[j],l]$.

3) For all empty entries in $\mathbf{c_i}$, $\mathcal{S}_2$ randomly generates a ciphertext and fills it into those empty entries.

In the next stage, $\mathcal{S}_2$ leverages the encrypted packet and the leakage functions to simulate the encrypted patterns, filter and transcript, for each encrypted packet $\mathbf{c_i}, 1 \leq i \leq |\mathbf{q}|$:

1) $\mathcal{S}_2$ puts $\mathbf{c_i}$ into $\mathbf{t}[i]$ and sets $\mathbf{c_i}$ as $\mathbf{x}_i$.
2) For each rule $R[j], 1 \leq j \leq |R|$:
   - $\mathcal{S}_2$ puts $\mathsf{Match}[i,j]$, $\mathsf{Match_P}[i,j]$ and $\mathsf{Act}[i,j]$ into $\mathbf{t}[i]$.
   - For each matched position $pos \in \mathsf{Match}[i,j]$ of $R[j]$, $\mathcal{S}_2$ sets $\alpha_i(\mathbf{v}_j) = \{pos, ..., pos + \mathbf{r}[j] - 1\}$ and $\beta_i^{'}(\mathbf{v}_j, \mathbf{x}_i) = \mathsf{Act}[i,j]$.
   - $\mathcal{S}_2$ calls $\mathcal{S}_{\mathsf{SHVE+}}(\alpha_i(\mathbf{v}_j), \beta_i^{'}(\mathbf{v}_j, \mathbf{x}_i))$ to get the corresponding encrypted pattern and put it into $\mathcal{E}$.
   - For each possible match position $pos_p \in \mathsf{Match_P}[i,j]$ of $R[j]$, $\mathcal{S}_2$ sets $\alpha_{i1}(\mathbf{v}_{j1}) = \{pos_p, pos_p + 1\}$ and $\beta_{i1}(\mathbf{v}_{j1}, \mathbf{x}_i) = \text{True}$; If $\mathbf{r}[j] \geq 3$, $\mathcal{S}_2$ also sets $\alpha_{i2}(\mathbf{v}_{j2}) = \{pos_p + 2, pos_p + 3\}$ and $\beta_{i2}(\mathbf{v}_{j2}, \mathbf{x}_i) = \text{True}$.
   - Then, $\mathcal{S}_2$ calls $\mathcal{S}_{\mathsf{SHVE}}(\alpha_{i1}(\mathbf{v}_{j1}), \beta_{i1}(\mathbf{v}_{j1}, \mathbf{x}_i))$ and $\mathcal{S}_{\mathsf{SHVE}}(\alpha_{i2}(\mathbf{v}_{j2}), \beta_{i2}(\mathbf{v}_{j2}, \mathbf{x}_i))$ (if $\mathbf{r}[j] \geq 3$) to get the corresponding filter and put it into $\mathcal{F}$.

Finally, $\mathcal{S}_2$ generates dummy HVE trapdoors to pad $\mathcal{E}$ and $\mathcal{F}$ to $|\mathcal{E}|$ and $|\mathcal{F}|$, respectively.

Due to the security properties of SHVE and SHVE+, the adversary cannot distinguish the real and simulated $\mathcal{E}$ and $\mathcal{F}$. Moreover, the transcript $\mathbf{t}$ is also indistinguishable since the ciphertext is simulated under the ideal cipher model, and the query history under real and ideal games are identical. Even if the adversary uses the $\mathcal{E}$ and $\mathcal{F}$ to examine the ciphertext, the output result is also indistinguishable. Therefore, $\mathcal{A}_2$ only has a negligible probability to learn more information than the defined leakage function $\mathcal{L}_2$ from the ruleset. ■

Theorem 4 shows that the adversary cannot get information about the ruleset more than $\mathcal{L}_2$ after receiving the ciphertext of chosen packets. This guarantees an untrusted *MB* cannot learn the ruleset with arbitrary legitimate packets. On the other hand, as a part of the pattern matching middlebox requirement, the matching information and action can be revealed towards the malicious packet. Hence, *MB* can still effectively inspect the packet and execute actions on malicious packets.

VI. Experiment and Evaluation

Environment setup and implementation. We choose two open-source rulesets, i.e., Snort ruleset [28] (1522 rules, 1116 patterns) and ETOpen ruleset[2] (24804 rules, 12634 patterns) to initialise our pattern matching middlebox module and use the traffic dump iCTF08[3] to evaluate its performance.

We implement the middlebox module and a gateway client in C++. Recall that SHVE+ combines each pattern string with all possible positions to generate the encrypted pattern list. This treatment ensures that matching can only happen in the positions indicated in the rules. To save storage, we choose AES-CMAC as the PRF to implement SHVE and SHVE+ and truncate the output of PRF to 5 bytes as in [6], [27]. Because the PRF outputs in the above scheme are used to mask the random key for the symmetric key encryption scheme Sym, truncating them does not affect the correctness of them. Hence, we can use the PRF to mask a 5-byte random value and employ a KDF (Key Derivation Function) to generate the key for Sym from the random value. Also, we substitute the non-wildcard position array S (see Section IV-A) to a 2-byte integer value indicating the length of each pattern. Note that this will not affect the correctness of SHVE+, because those positions represent a continuous string under the pattern matching application. After optimisation, each encrypted pattern requires 23 bytes (1 PRF value + 1 AES ciphertext + 1 pattern length), and the encrypted pattern list for Snort ruleset costs 43.5 MB, while the one for ETOpen requires 808 MB. On the other hand, the secure filtering protocol generates SHVE trapdoors for the beginning 2 bytes of each distinct pattern, and it further generates SHVE trapdoors for the following 2 bytes if the pattern is larger than 3 bytes. We observe that the filtering protocol generates a 32 MB filter from Snort ruleset, and 129 MB for ETOpen ruleset. These storage costs are moderate to a cloud server where the middlebox is supposed to deploy. In addition, computing and uploading the above lists are one-time costs in the initialisation phase, and it enables the middlebox to save bandwidth during the inspection phase tremendously.

For performance evaluation, we deploy the middlebox on a server equipped with Intel Core i7-6700 3.4GHz CPU and 16GB RAM and use a desktop with Intel Core i5-6500 3.2GHz CPU and 8GB RAM as the gateway client.

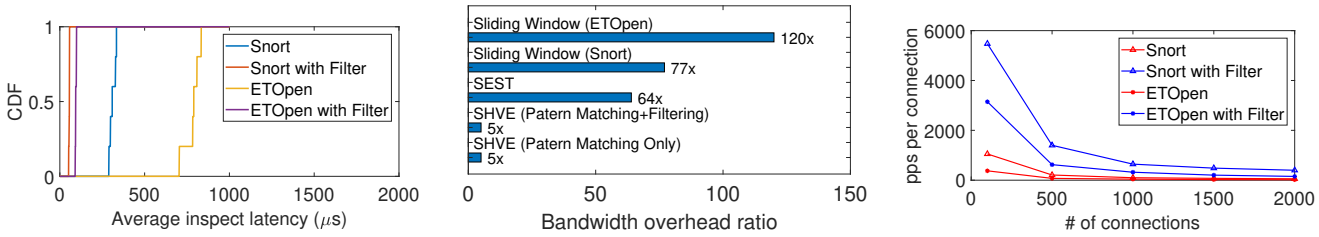
Performance evaluation. First, we show the setup time for the middlebox module. For the generation of the encrypted pattern list, it runs SHVE+.KeyGen for each pattern and its all possible positions. This takes 18.9 s in Snort ruleset and 287.3 s in ETOpen ruleset. Similarly, the filter generation combines each distinct 2 bytes extracted from the beginning of pattern with all possible positions and runs SHVE.KeyGen operations to get the filter trapdoor, it also processes the next 2 bytes for the longer pattern. Our evaluation shows that it requires 14 s and 45.5 s for our two rulesets, respectively.

Next, we report the runtime performance of the middlebox module. In Fig. 4a, we evaluate the average inspection latency under two rulesets, respectively. For Snort ruleset, the inspection delay is less than 300 μs. For the larger ETOpen ruleset, the inspection delay is less than 850 μs. We further examine the inspection latency after applying our secure filtering protocol. As a result, the middlebox only takes less than 60 μs to inspect a packet in Snort ruleset, and 100 μs for ETOpen ruleset, because in the iCTF08 dataset, only $1/6$ packets and $1/9$ packets need further inspection against Snort ruleset and ETOpen ruleset, respectively.

As shown in Fig. 4b, the bandwidth overhead in our proposed design is a constant, i.e., 5 times in terms of the original packet size. This overhead is much smaller than any existing secure middlebox system using sliding window tokenisation

[2]Emerging Threats ruleset: https://rules.emergingthreats.net

[3]iCTF08 dumps: https://ictf.cs.ucsb.edu/archive/2008/dumps/



(a) Insepction latency (b) Bandwidth Overhead (c) Throughput

Fig. 4: The performance evaluation for the proposed middlebox module.

TABLE II: Throughput of our middlebox for different rulesets.

Ruleset	Snort	Snort (filter)	ETOpen	ETOpen (filter)
Throughput	206 MBps	1442 MBps	75 MBps	578 MBps

TABLE III: Theoretical performance comparison between the existing pattern matching middleboxes and our middlebox. n is the packet size and l is the size of pattern string.

Scheme	Pattern size	Traffic size	Inspection time
SEST [24]	$1+\lceil l/8 \rceil$	$64n$	$(n-l+1)T_{\mathsf{p}}$[b]
BlindBox [6]	$16\lceil l/16 \rceil$	$5((n+1)\lvert D_{\mathsf{R}}\rvert$[a] $-\Sigma D_{\mathsf{R}})$	$((n+1)\lvert D_{\mathsf{R}}\rvert$ $-\Sigma D_{\mathsf{R}})T_{\mathsf{M}}$[c]
Our middlebox	$23(n-l+1)$	$5n$	$(n-l+1)$ $(lT_{\mathsf{XOR}}$[d] $+T_{\mathsf{Dec}}$[e]$)$

[a] An array with all distinct pattern sizes in the ruleset
[b] Time taken to compute a pairing
[c] Time taken to access a tree index in memory
[d] Time taken to compute an XOR operation
[e] Time taken to decrypt a symmetric ciphertext

algorithms [6], [24] because those algorithms enumerate all possible window sizes when tokenising the traffic payload. More specifically, for Snort ruleset, the number of distinct pattern sizes is 82 (1 – 214 bytes), and the sliding window tokenisation enlarges the bandwidth consumption by 77×. For ETOpen ruleset, there are more distinct pattern sizes than Snort ruleset, i.e., 130 (1 – 196 bytes). Therefore, the bandwidth overhead of the sliding window tokenisation reaches 120×. In comparison, our system performs encryption and queries in byte-wise. Namely, it only scans the traffic once, and thus, the bandwidth overhead keeps constant, and it saves 94% – 96% bandwidth comparing to the sliding window tokenisation in Snort and ETOpen ruleset. Another approach (SEST [24]) with constant bandwidth overhead is based on the elliptic curve. However, the ciphertext in the elliptic curve is much longer than that in our symmetric building blocks, and this approach still leads to a prohibitive bandwidth overhead (64×).

We simulate a multi-session scenario (100 to 2000 clients) to measure the throughput of the middlebox on our two different rulesets. The results are given in Fig. 4c and Table II. As our middlebox can perform filtering and matching efficiently in parallel, the throughput for each connection can reach up to 5000 packets per second (pps) for 100 connections under Snort ruleset, and around 300 pps when there are 2000 connections, and the overall throughput achieves 1442 MBps. For ETOpen ruleset, the throughput per connection still reaches 3000 pps for 100 connections, and the overall throughput is 578 MBps.

Comparison between prior designs. We provide theoretical and real-world performance comparisons between SEST [24], BlindBox [6] and our middlebox. Note that the work [7], [8] adopts a similar approach based on searchable encryption and tokenisation as BlindBox. Therefore, the comparison with BlindBox can also demonstrate our advantages to the above work. The source code of [6], [24] is not publicly available, so we only compare our results with the one reported in their paper. We note that the test machine of SEST has similar capabilities as ours, while BlindBox is evaluated on a much better machine (Intel Xeon E5-2650 2.6GHz, 128GB RAM).

TABLE IV: Performance comparison between the existing pattern matching middleboxes and the proposed middlebox using a 1500-byte packet and Snort ruleset.

Scheme	Traffic size (bytes)	Inspection time (1 rule (100 bytes), 1 packet)
SEST [24]	96000	600 ms
BlindBox [6]	115504	5 μs
Our middlebox	7500	43 μs

In Table III, we compare the theoretical performance from three perspectives of the listed works, i.e., the size of encrypted patterns, the size of the encrypted traffic ciphertext sending to the middlebox, and the inspection time on the middlebox. The encrypted pattern size mainly affects the storage consumption of the middlebox. Although our scheme has the largest storage overhead, it is still a moderate cost to a cloud server, as mentioned in the performance evaluation part. On the other hand, the encrypted traffic size of our proposed scheme is much smaller than the other two schemes; it implies that our middlebox can save enormous bandwidth comparing with [6], [24]. In terms of the inspection time, all schemes are linear in the length of the packet from the complexity view. Nonetheless, the inspection time of our middlebox is comparable to BlindBox: both of them achieve a microsecond-level inspection delay because the inspection using the SHVE scheme is based on ultra-fast operations, i.e., XOR and Sym.Dec, which is only slightly slower than the index access operations in BlindBox. However, the inspection delay of SEST is larger because it relies on cryptographic pairing, which can take a millisecond for each pairing.

We report the performance comparison over real-world data in Table IV. We encrypt a 1500-byte packet as the encrypted traffic and use Snort ruleset to inspect the traffic on the middlebox. The result shows that our client only sends 7500 bytes to the middlebox to inspect the given packet, which is 13-15 times smaller than [6], [24]. When inspecting a 100-byte pattern in the ruleset, although SEST [24] and our middlebox

TABLE V: Monthly cost estimation between the tokenisation-based middleboxes and the proposed middlebox with ETOpen ruleset under AWS pricing information.

Scheme	Instance	Network	Total
Blindbox [6]	$244.8	$1620 (10 Gb bandwidth)	$1864.8
Our middlebox	$244.8	$216 (1 Gb bandwidth)	$460.8

leverage the linear scan to inspect the packet, SEST needs 600 ms to finish the inspection as it is based on the public-key cryptographic scheme, which is very slow in practice, while our middlebox based on SHVE only needs 43 μs. As reported in BlindBox [6], inspecting one rule against a packet only requires 5 μs. Note that our inspection delay is also in the microsecond-level, which is negligible in real-world scenarios. Also, the testbed machine they used is much better than ours.

Deployment cost comparison. To further illustrate the practicality of our middlebox, we estimate the deployment cost of our scheme and the representative (i.e., BlindBox [6]) of tokenisation-based approaches [6]–[8].

In this cost estimation, we assume that the enterprise deploys a 7/24 pattern matching middlebox on AWS to examine all its traffic. In particular, the enterprise hires a c5.2xlarge EC2 instance (8 cores, 16 GB RAM) to host the middlebox module. Note that this instance has sufficient memory for the proposed middlebox since the previous evaluation shows that 1 GB is enough to store the encrypted pattern list and filter generated from ETOpen ruleset. To have a consistent and stable network connection with higher bandwidth and throughput, the enterprise connects its network to the EC2 instance through AWS Direct Connect [29]. However, due to the traffic size under BlindBox is 15 times larger than our middlebox, BlindBox needs higher network capacity in order to achieve similar performance as our middlebox. For instance, if our middlebox requires 1 Gbps bandwidth to guarantee a low delay, then BlindBox should use the 10 Gbps plan instead.

We refer to the pricing information in the U.S. East region (Virginia) to compute the price, and the monthly cost estimation result according to the above assumptions is listed in Table V. The result shows that even though the instance cost is the same for BlindBox and our middlebox, BlindBox has to pay 7.5$\times$ more ($1620 versus $216) to get the same network performance as ours. In total, the monthly cost of deploying our middlebox in AWS only needs $460.8 while BlindBox takes $1864.8, which indicates a 300% extra cost.

VII. Conclusion

In this paper, we design a system that allows outsourced middleboxes to perform pattern matching over encrypted traffic without revealing both traffic content and patterns. We first design a customised SHVE scheme (SHVE+) and then build an encrypted pattern matching protocol based on SHVE+ to protect pattern and network traffic during the pattern matching process. Next, we design a secure filtering protocol that can quickly find the starting positions for each possible match, which improve the pattern matching process further. Our system is implemented as a prototype, and our evaluation on real-world ruleset and traffic dump illustrates its advantages in terms of bandwidth, inspection delay and throughput.

References

[1] N. Cranford, "Verizon Brings Virtual Network Services to Amazon Cloud," https://www.rcrwireless.com/20170816/verizon-brings-virtual-network-services-to-amazon-cloud-tag27 [online], 2017.

[2] C. Wang, X. Yuan, Y. Cui, and K. Ren, "Toward Secure Outsourced Middlebox Services: Practices, Challenges, and Beyond," *IEEE Network*, vol. 32, no. 1, pp. 166–171, 2017.

[3] N. Sultana, M. Kohlweiss, and A. W. Moore, "Light at the Middle of the Tunnel: Middleboxes for Selective Disclosure of Network Monitoring to Distrusted Parties," in *HotMIddlebox'16*, 2016.

[4] J. Sherry *et al.*, "Making Middleboxes Someone Else's Problem: Network Processing as A Cloud Service," in *ACM SIGCOMM'12*, 2012.

[5] Z. Zhou and T. Benson, "Towards a Safe Playground for HTTPS and Middle Boxes with QoS2," in *HotMIddlebox'15*, 2015.

[6] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep Packet Inspection over Encrypted Traffic," in *ACM SIGCOMM'15*, 2015.

[7] X. Yuan, X. Wang, J. Lin, and C. Wang, "Privacy-Preserving Deep Packet Inspection in Outsourced Middleboxes," in *IEEE INFOCOM'16*, 2016.

[8] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, "Embark: Securely Outsourcing Middleboxes to the Cloud," in *USENIX NSDI'16*, 2016.

[9] J. Fan, C. Guan, K. Ren, Y. Cui, and C. Qiao, "SPABox: Safeguarding Privacy During Deep Packet Inspection at A Middlebox," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3753–3766, 2017.

[10] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, "Safebricks: Shielding Network Functions in the Cloud," in *USENIX NSDI'18*, 2018.

[11] H. Duan *et al.*, "LightBox: Full-stack Protected Stateful Middlebox at Lightning Speed," in *ACM CCS'19*, 2019.

[12] B. Trach *et al.*, "ShieldBox: Secure Middleboxes using Shielded Execution," in *SOSR'18*, 2018.

[13] J. Han, S. Kim, J. Ha, and D. Han, "SGX-Box: Enabling Visibility on Encrypted Traffic using A Secure Middlebox Module," in *APNet'17*, 2017.

[14] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions," *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.

[15] J. Van Bulck *et al.*, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security'18*, 2018.

[16] S. Van Schaik *et al.*, "RIDL: Rogue In-Flight Data Load," in *IEEE S&P'19*, 2019.

[17] P. Kocher *et al.*, "Spectre Attacks: Exploiting Speculative Execution," in *IEEE S&P'19*, 2019.

[18] V. Iovino and G. Persiano, "Hidden-Vector Encryption with Groups of Prime Order," in *Pairing'08*, 2008.

[19] S. Lai *et al.*, "Result Pattern Hiding Searchable Encryption for Conjunctive Queries," in *ACM CCS'18*, 2018.

[20] H. J. Asghar, L. Melis, C. Soldani, E. De Cristofaro, M. A. Kaafar, and L. Mathy, "Splitbox: Toward Efficient Private Network Function Virtualization," in *HotMIddlebox'16*, 2016.

[21] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han, "DFC: Accelerating String Pattern Matching for Network Applications," in *USENIX NSDI'16*, 2016.

[22] C. Stylianopoulos, M. Almgren, O. Landsiedel, and M. Papatriantafilou, "Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization," in *IEEE ICCP'17*, 2017.

[23] Y. Guo, C. Wang, X. Yuan, and X. Jia, "Enabling Privacy-Preserving Header Matching for Outsourced Middleboxes," in *IEEE/ACM IWQoS'18*, 2018.

[24] N. Desmoulins, P.-A. Fouque, C. Onete, and O. Sanders, "Pattern Matching on Encrypted Streams," in *ASIACRYPT'18*, 2018.

[25] M. Chase and E. Shen, "Substring-Searchable Symmetric Encryption," *Proceedings on Privacy Enhancing Technologies*, vol. 2015, no. 2, pp. 263–281, 2015.

[26] F. Hahn, N. Loza, and F. Kerschbaum, "Practical and Secure Substring Search," in *ACM SIGMOD'18*, 2018.

[27] X. Yuan, H. Duan, and C. Wang, "Bringing Execution Assurances of Pattern Matching in Outsourced Middleboxes," in *IEEE ICNP'16*, 2016.

[28] Snort, "Snort Community Ruleset," https://www.snort.org/downloads/#rule-downloads [online], 2019.

[29] AWS, "AWS Direct Connect," https://aws.amazon.com/directconnect/ [online], 2019.