Privacy-preserving Deep Packet Inspection in Outsourced Middleboxes

Xingliang Yuan*†, Xinyu Wang*†, Jianxiong Lin*, and Cong Wang*†
*City University of Hong Kong, Hong Kong, China
†City University of Hong Kong Shenzhen Research Institute, Shenzhen, 518057, China
{xinglyuan3-c, jianxilin2-c}@my.cityu.edu.hk, {xinywang, congwang}@cityu.edu.hk.

***Abstract*—Middleboxes are essential for a wide range of advanced traffic processing in modern enterprise networks. Recent trend of deploying middleboxes in cloud as virtualized services further expands potential benefits of middleboxes while avoiding local maintenance burdens. Despite promising, designing outsourced middleboxes still faces several security challenges. First, many middlebox processing services, like intrusion detection, require packet payload inspection, while the ever-increasing adoption of HTTPS limits the function due to the end-to-end encryption. Second, many packet inspection rules used by middleboxes can be proprietary in nature. They may contain sensitive information of enterprises, and thus need strong protection when configuring middleboxes in untrusted outsourced environments. In this paper, we propose a practical system architecture for outsourced middleboxes to perform deep packet inspection over encrypted traffic, without revealing either packet payloads or inspection rules. Our first design is an encrypted high-performance rule filter that takes randomized tokens from packet payloads for encrypted inspection. We then elaborate through carefully tailored techniques how to comprehensively support open-source real rulesets. We formally analyze the security strength. Implementations at Amazon Cloud show that our system introduces roughly 100 millisecond latency in each connection initialization, with individual processing throughput over 3500 packets/second for 500 concurrent connections.**

I. Introduction

Middleboxes are ubiquitous in modern enterprise networks for providing a wide range of specialized network functions [1]–[3], such as intrusion detection, exfiltration prevention, firewall, etc. Yet, maintaining in-house middlebox infrastructure is known to incur expensive and complex management burdens for enterprises [1]. Thus, recent trends have been calling for moving the middlebox processing to public clouds as virtualized services [1], [3], while relieving the enterprises of local maintenance burdens. Such outsourced middleboxes can further benefit the enterprises with easy management, cost effectiveness, scalability, fault tolerance, and beyond.

Despite very promising, outsourcing middlebox processing has also brought several new security challenges that are yet to be fully addressed. First, redirecting traffic to outsourced middleboxes would give the cloud provider full access to all the traffic flows. While adopting off-the-shelf HTTPS ensures the end-to-end traffic confidentiality [4], [5], the encrypted packet payload would simultaneously limit the middlebox processing capabilities, such as intrusion detection and exfiltration prevention, which would otherwise require deep packet inspection (DPI). Second, many packet inspection rules are customised by enterprises, and can be proprietary in nature [6], [7], containing potentially sensitive information like trade secrets, intellectual property, etc. Thus, from the enterprise perspective, strong protection of these valuable rulesets is also highly demanded [7], [8], especially when middleboxes are deployed in the outsourced cloud environment.

Most of existing middleboxes handle HTTPS through a walk-around but limited approach, simply by intercepting and decrypting the encrypted traffic [1], [9]. This approach, besides somewhat undesirably revealing the packet payloads at middleboxes, would easily constitute possible man-in-the-middle attacks [10]. A recent design called BlindBox [5] is the first to enable middleboxes to perform DPI over HTTPS. But it is still not suitable in the context of middlebox outsourcing, since it does not consider protecting the rules against middleboxes in untrusted environments. We also remark that BlindBox currently is not ready for practical deployment, due to its expensive connection setup, involving a secure two-party computation protocol between each endpoint and the middlebox.

The above shortcomings of existing approaches motivate us to investigate a privacy-preserving and practical DPI system. Our research aims to enable the outsourced middleboxes to perform packet inspection over encrypted traffic without revealing the sensitive inspection rules or the packet payloads. To address the challenges, our first insight is to formulate the problem as encrypted token matching. Specifically, traffic packet payloads can be parsed and encrypted into randomized tokens. The suspicious strings and the responsive actions[1] from packet inspection rules can also be extracted as key-value pairs, e.g., (“password”, “alert”), based on which an encrypted rule filter can be built to index those encrypted pairs. By feeding the encrypted traffic tokens into the encrypted rule filter, such a blueprint can be immediately instantiated via existing searchable encryption techniques [12], [13].

However, turning the blueprint into a secure and usable DPI system still encounters non-trivial obstacles. The first is how to build an encrypted rule filter with convincingly high performance. Middleboxes that are capable of inspecting packet content with low latency, superior throughput, memory efficiency, high-speed setup, etc., are indispensable for any us-

[1]In general, the actions in existing instruction detection systems include alert (i.e., generate an alert to administrators), log (i.e., log the packet), drop (i.e., block the packet), stop (i.e., reject the connection), etc [11].

978-1-4673-9953-1/16/$31.00 ©2016 IEEE

able DPI systems. Directly applying existing generic primitives of searchable symmetric encryption (SSE), e.g., [12], [13] into our specific contexts does not necessarily achieve all of our design requirements. To that end, we propose to bridge the security framework of SSE and one recent high-performance hash table design [14] to derive our encrypted rule filter from the ground up. The resulting design encompasses all the aforementioned performance features while can be provably secure. Only if suspicious strings are matched, the actions will be recovered, triggered against intrusion and exfiltration. Throughout the entire process, the middlebox can never learn the content of packet payloads or the semantic information of rules except the triggered actions.

The second obstacle is how to provide comprehensive support of pragmatic inspection via complicated inspection rules. In general, some rules specify inspection attributes, e.g., packet fields or payload offsets. Revealing them may compromise the confidentiality of rules and packet payloads. Others with multi-conditions need all the conditions are matched before revealing actions. Therefore, supporting them as well as other special ones demands more than a simple SSE application. Consequently, we design carefully tailored techniques in a in-depth manner to handle each of them respectively, while the strong protection on rules and payloads is still ensured. Among others, one of our technique highlights is the adoption of secret sharing to encrypt the actions embedded in the encrypted filter design, in case of multi-condition matching. As a result, if and only if all the suspicious strings in the rule are matched, the action will be triggered. In addition, we also consider the rules sharing common strings, i.e., cross-rule inspection, and the rules inspecting repeated strings across different connections, i.e., cross-connection inspection.

Furthermore, we optimize the system implementation for better security and performance. To hide the equality of incoming tokens, which are generated from the same underlying strings, we trade space for security; that is, every suspicious string is duplicated in multiple copies, where those copies can be later matched by different tokens. To protect the token equality against other connections, a fresh encrypted filter should be built for a newly established connection. We further decouple this procedure from the connection initialization such that multiple filters are pre-built periodically in an asynchronous process. Therefore, the inspection can be initiated immediately when a connection is established. In summary, our contributions are listed as follows:

- We design the first secure DPI system that enables the outsourced middleboxes to perform deep packet inspection over encrypted traffic while providing the strong protection on both packet payloads and inspection rules.
- We propose an encrypted high-performance filter with high throughput, fast setup, and low-memory consumption, and carefully tailor the design for broad support of inspection rules with detailed protocol illustration. We formulate the security definition, and prove the proposed system against adaptive chosen-keyword attacks.
- We implement the system prototype and deploy it on Amazon Cloud. Real rulesets are used for evaluation. The results show that our design can directly detect over 90% suspicious packets in most of selected intrusion detection traffic dumps, and the throughput per connection achieves up to $3,600$ packets per second on an Amazon instance with 500 concurrent connections.

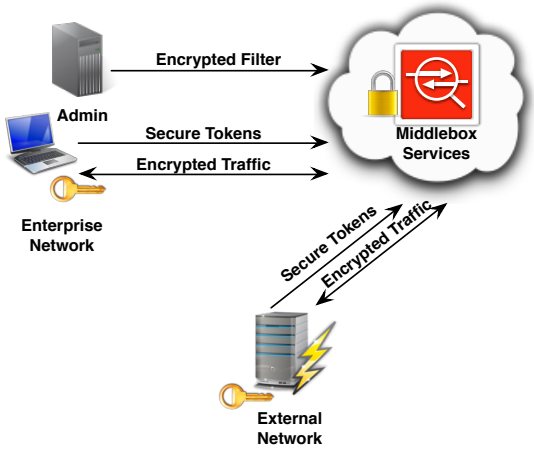


Fig. 1: System Architecture

II. Problem Statement

Fig. 1 illustrates our system architecture. It consists of four parties: the *client* inside the enterprise network, the *host* in the external network, the *admin server* (*AS*) maintained by enterprise administrators, and the *middlebox* (*MB*) deployed in the outsourced environments as a cloud or network service. We also use the term "endpoint" to denote the *client* and the *host*, where a HTTPS connection is established.

A. Application Scenarios and Trust Assumptions

We consider a common application scenario before presenting our threat model and system architecture. An enterprise needs to thoroughly inspect the content of incoming and outgoing network packets to defend against malicious activities. For cost efficiency, fault tolerance, and good scalability, it subscribes remote middlebox services, e.g., intrusion detection, exfiltration prevention, etc. Meanwhile, the clients will establish end-to-end encryption to outside hosts against eavesdropping. Here, we assume that the outsourced middleboxes, for example cloud-based middlexboxes [3], [5], are powerful with abundant computation resources, where multiple servers can be launched to handle a huge amount of encrypted traffic and process a large number of tokens concurrently.

In addition, the enterprise administrator, i.e., the rule creator, needs to protect the rules [5], [7], because leaving them in cleartext will compromise the privacy of enterprise. In practice, the enterprise may subscribe the ruleset from professional vendors [5], e.g., Symantec or some public rulesets from open-source DPI systems, e.g., Snort [11]. Then it customizes the rules to prevent data exfiltration or tunes the rules to improve the inspection accuracy [6]. Namely, the rules can be highly related to the sensitive information, e.g., trade

secrets, intellectual property, etc. Here, we assume that *AS* is trustworthy. It will neither expose the ruleset to *MB* nor grant the access of rules to the endpoints.

Semi-honest middlebox: In this work, we consider the *MB* service provider could be semi-honest [7], [15], such as the public cloud [1], [3]. It offers *MB* services faithfully, but intends to exploit the sensitive information from the traffic passed by, and tries to infer the proprietary inspection rules. On the other hand, *MB* is deployed in an untrusted environment. It is likely to be hacked and eavesdropped. Thus, both rules and traffic should be encrypted to achieve defense in depth. Currently, how to verify outsourced *MB* services is not our focus, and remains to be practically addressed as stated in [16]. We will explore the threats of malicious *MB*s in the near future.

Trust assumption on endpoints: We assume that at least one endpoint is honest, similar to the threat model in existing DPI systems [5], [6]. We also note that detecting two malicious endpoints is orthogonal to our work. For example, covert channels between malicious endpoints can be detected via traffic pattern analysis [17] and application-specific detection [18].

B. Overview of System Architecture

The overview of our proposed system is depicted in Fig. 1. It functions in four stages with four parties introduced before.

Initialization: First of all, two endpoints S and R run the standard SSL protocol to establish an encrypted connection. Then they need to register at *AS* for the request of key K_S and key K_R respectively via encrypted channels. In the meantime, *AS* builds an encrypted filter that securely indexes the encrypted string-action pairs extracted from the rules, and uploads it to *MB*. After that, *MB* can perform packet inspection for this connection through the appropriate encrypted filter.

Preprocessing: Once the initialization is completed, one endpoint starts to send encrypted traffic. Meanwhile, it will parse the packet payloads into a set of strings based on predefined principles, and use K_S to transform plaintext strings to randomized tokens, which will be sent to *MB* as well. We note that the inspection is bidirectional. The other endpoint parses the packet payloads in terms of same principles, and use its own key K_R for token generation.

Inspection: As long as the encrypted traffic and the tokens arrive, *MB* will hold up the traffic and execute the proposed secure DPI protocol to process the tokens over the encrypted filter in a streaming fashion. If a token correctly recovers an entry of the filter, *MB* will take the resulting action, e.g., alerting *AS*, dropping packets, etc. After all the tokens in a packet are checked without a match, the packet is considered legitimate and allowed through. If verification is required, *MB* continuously computes a cryptographic digest from a batch of the processed tokens, and send it to the other endpoint.

Verification: Similar to [5], to detect dishonest/malicious misbehaviors of the other endpoint, the receiving endpoint will use the SSL session key to decrypt the payloads, and then reconstructs the digest for token verification.

Remark: Following prior studies that call for middlebox outsourcing as a service [1], [3], [15], we assume the existence of *AS*, which can be a readily available gateway server and controlled by the trustworthy enterprise administrator to build the encrypted filter and redirect enterprise traffic to the cloud. Only the *AS* needs to be cloud-aware. In this outsourcing setting, we also note that cloud with wide geographic footprint could help reduce the extra traffic detour latency significantly [1]. Techniques for redirecting traffic to outsourced middleboxes are well-discussed before [1], [3], and we do not go into detail here. Our major focus is on designing an encrypted filter with carefully tailored techniques to cover a wide range of inspecting rules. As we mention below, even this is a non-trivial task that requires thoughts from achieving security, performance, and functionality simultaneously.

Algorithm 1 Build the encrypted filter

Input: $\{(str_1, act_1), \cdots, (str_n, act_n)\}$: the string-action pairs extracted from the ruleset $\mathcal{R}$; K_1, K_2: private keys; F_1, F_2, P_1, P_2: PRFs; τ: the load factor; d: the number of entries in each bucket; β: the cuckoo threshold.

Output: $\mathcal{F}$: the encrypted filter.

1: Initial hash table T_1 and T_2 with capacity $\lceil \frac{n}{2d\tau} \rceil$;
2: $\forall str_i \in \{str_1, \cdots, str_n\}$, compute $t = F_1(K_1, str_i)$, $t_1 = P_1(t, 1)$, and $t_2 = P_2(t, 2)$, and place a_i to 1 of d entries in $T_1[t_1]$ or $T_2[t_2]$ if there exists an empty entry.
3: If no empty entry exists, randomly select 1 entry from $T_1[t_1]$ or $T_2[t_2]$, replace the inside act_i' with act_i, and re-insert act_i' as shown in Step 2 within β recursive trials.
4: After all actions are inserted, encrypt each of them via $act_i \oplus s$, where the mask $s = F_2(K_2, str_i)$, and fill the empty entries with random strings.

III. The Proposed System

In this section, we will present the proposed system, and explain the design intuition regarding security, performance, and functionalities. We first propose an encrypted rule filter, i.e., the core building block of our system. The filter enables *MB* to perform private and efficient DPI over encrypted traffic without seeing packet payloads or inspection rules. Furthermore, we tailor the designs for a wide range of rule support, and present the inspection in an in-depth manner.

A. Encrypted Filter

In order to enable DPI over encrypted traffic, one straightforward approach is to encrypt the suspicious strings indicated in the rules into randomized tokens for *MB* so that it can later perform token matching with the ones coming from the packet payloads of the endpoints. But such approach requires a linear scan over all the rule tokens for the inspection of each incoming token [5]. It is not scalable since the time complexity increases as the number of rules grows, which could become the performance bottleneck for bandwidth-insensitive applications.

To improve efficiency and scalability, we propose a high-performance encrypted filter, which is built upon the security framework of searchable symmetric encryption [12], [13] and one of the latest efficient hash table designs [14]. Based on the security techniques used in [12], [13], we transform a memory

efficient, high throughput hash table into an encrypted index while preserving its original performant features.

Unlike prior encrypted index designs [12], [13], which need to store both encrypted strings and actions in a dictionary, our proposed filter shrinks the size by only storing the encrypted action while still preserving the correctness. Explicitly, we first use the token generated from a string to seek the available space for its action, and then securely embed the string into a random mask to encrypt the action. As a result, only if the mask and the token are derived from the same suspicious string, the action will be recovered. In addition, cuckoo hashing is applied to make the filter extremely compact [19]. The actions are allowed to be relocated among different spaces so that the filter achieves high load factors, e.g., 95% [14].

Construction: As mentioned, a pair of a suspicious string and its responsive action can be extracted from the inspection rule. Then all the string-action pairs are inserted to the proposed filter in a random fashion, and all buckets inside are encrypted.

The steps of building the encrypted filter are presented in Algorithm 1, and an illustrative example is depicted in Fig. 2. Initially, two hash tables T_1 and T_2 are created. For each of total n string-action pairs, the first step is to insert the action act_i into one of the two buckets $T_1[t_1]$ and $T_2[t_2]$, where t_1 and t_2 are transformed by pseudo-random functions from the suspicious string str_i, i.e., $t = F_1(K_1, str_i)$, $t_1 = P_1(t, 1)$, and $t_2 = P_2(t, 2)$. To handle hash collisions from different strings, each bucket contains d entries, so every insertion will have $2d$ choices to place the action.

If there is no empty entry in two buckets, the data relocation in cuckoo hashing is trigged. Explicitly, one of the entries will be randomly selected, and the action act_i' inside will be replaced by act_i. After that, act_i' is re-inserted in a recursive way starting from the first step. As all actions are inserted, they are encrypted via XORing random masks, i.e., $act_i \oplus s$, where $s = F_2(K_2, str_i)$. Based on this design, str_i does not need to be stored, while the action act_i can still be recovered if a token is matched. The detailed protocol will be shown later in Section III-B. Lastly, the rest of few empty entries are filled with random strings.

Consideration on security and usability: Based on the proposed encrypted filter, only when the tokens are generated from suspicious strings, *MB* is given a capability to recover the action. But directly deploying this filter does not provide strong security strength. First, current construction leaks the equality of incoming tokens no matter they are matched or not, since they are all generated from a deterministic one-way function. Second, the same private keys are used to generate tokens and the encrypted filter. Thus, anyone who gets the keys may output tokens for malicious purposes, e.g., DoS attacks.

Regarding the wide range of rule support, current design for single string matching, only works for a small percentage of inspection rules. In reality, many other advanced rules are defined to improve the detection accuracy and capture complex malicious activities [11]. Specifically, those rules may contain specific attributes or inspection ranges, or need multi-condition inspection. Yet, it is not clear how to support them by simply

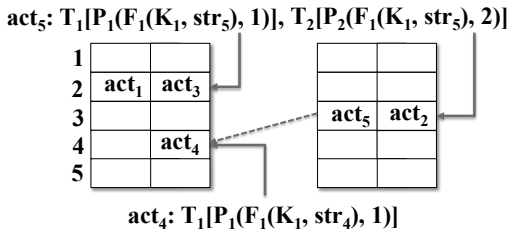


Fig. 2: An example of insertion to the encrypted filter. Here, each bucket has 2 entries, and the buckets for act_5 are full. Then act_4 is randomly selected and relocated, and act_5 is placed in the entry where act_4 is previously placed.

using the proposed single token matching algorithm. Next, we will propose a secure protocol for single token matching which addresses the security concerns, and then present four tailored designs for comprehensive DPI support.

B. Secure Single Token Inspection

This protocol enables single token inspection as shown in the following rule from an open-source ruleset [11]. Two aforementioned security concerns should be addressed. The equality of streaming tokens needs to be hidden, and authorized token generation needs to be enforced against DoS attacks.

> Snort rule #2: alert tcp $EXTERNAL_NET$ any -> $HOME_NET$ 7597 (flow:to_server, established;content:"qazwsx.hsq";)

For the former concern, we propose to trade space for security. On the one hand, inspecting each individual connection requires an encrypted filter with fresh keys, so the same strings in different connections will no longer be the same. On the other hand, each rule string is duplicated and securely indexed in the filter, so even the same suspicious strings are matched by randomized tokens. Particularly, we introduce an incremental counter c bounded by a pre-defined security threshold C, which is concatenated with the duplicate string as follows:

$$\{str||0, \cdots, str||C-1\}$$

Accordingly, t is equal to $F_1(K_1, str||c)$, and s is equal to $F_2(K_2, str||c)$, while t_1 and t_2 are computed in the same way for action insertion and encryption. If C is sufficiently large, the repeated strings in one connection will map to different tokens. Otherwise, the counter c will be reset to 0. Even in that case, the security strength is still improved, since the distribution of appeared strings in the payloads is obfuscated.

At the same time, we enforce an access control mechanism to the endpoints via broadcast encryption [20], which is inspired from the multi-client support in a searchable symmetric encryption scheme [12]. Only the authorized endpoint will be given the ability to generate valid tokens. And if a malicious endpoint is detected, it will be revoked efficiently.

Detailed protocol: Given a key generation function $\mathsf{KGen}(1^k)$, a broadcast encryption scheme $\mathsf{BE}(\mathsf{Enc}, \mathsf{Dec}, \mathsf{Add})$, a ruleset $\mathcal{R}$, the proposed algorithm Build, the protocol of single token matching for an individual connection is presented as follows:

• INITIALIZATION:

1) *AS* generates $\{K_1, K_2, K_3\} \leftarrow \mathsf{KGen}(1^k)$, where k is the security parameter, and builds $\mathcal{F} \leftarrow \mathsf{Build}(K_1, K_2, \mathcal{R}, C)$.
2) For two registered endpoints *S* and *R*, *AS* computes state information $st \leftarrow \mathsf{BE.Enc}(K_3, \mathbf{P}, r)$, where $\mathbf{P} = \{S, R, MB\}$, and r is a k-bit random string. Then it generates $K_S \leftarrow \mathsf{BE.Add}(K_3, S)$ and $K_R \leftarrow \mathsf{BE.Add}(K_3, R)$.
3) $\{\mathcal{F}, r, st\}$ are sent to *MB*, and $\{K_1, K_2, K_S, st, C\}$ and $\{K_1, K_2, K_R, st, C\}$ are sent to *S* and *R* respectively.
4) *S* and *R* run the standard SSL key exchange protocol.

• PRE-PROCESSING:

1) *S* parses the packet payloads, and transforms the strings into tokens, (t, s) for each, where $t = F_1(K_1, str||c)$, $s = F_2(K_2, str||c)$, c is cached for every distinct string, and incremented when another repeated string appears.
2) *S* computes $r \leftarrow \mathsf{BE.Dec}(K_S, st)$, and $\sigma \leftarrow G(r, t||s)$, where G is a pseudo-random permutation keyed by r. After that, *S* sends σ to *MB*.
3) *S* encrypts the traffic with the SSL key and send it to *MB*.

• INSPECTION:

1) *MB* holds up the encrypted traffic, and computes $t||s = G^{-1}(r, \sigma)$ for each received σ. Then it generates $t_1 = P_1(t, 1)$, $t_2 = P_2(t, 2)$, and performs $e \oplus s$ for every entry e in $T_1[t_1]$ and $T_2[t_2]$.
2) Only if a token related to a suspicious string is matched, the action act will be recovered via the XOR operation, or the token is marked legitimate. When all tokens in a packet are legitimate, this packet is allowed to pass by.

• VERIFICATION:

1) *MB* continuously generates secure digests for a batch of passed tokens via a cryptographic hash function, e.g., SHA256, and sends them with the encrypted traffic to *R*.
2) *R* decrypts the traffic, generates tokens based on the same principle as *S*, and computes the digests for verification.
3) If the digest is not the same, *S* is considered malicious. *MB* is notified and revokes *S*: $st' \leftarrow \mathsf{BE.Enc}(K_3, \mathbf{P}, r')$, where $\mathbf{P} = \{R,\ MB\}$, and r' is a fresh k-bit random string.

Remark: Existing principles for parsing the payloads like delimiter-based segmentation [5] and window-based *n-gram* analysis [11] can be applied at the endpoints during the preprocessing so as to facilitate effective inspection on binary and text data. In this paper, we implement those two principles for our experiments, and more detailed discussion can be found in [5]. We also remark that the above listed protocol is just a basic operational outline. Further protocol-level variations could be possible in practice, depending on different choices actually adopted for traffic redirection to outsourced middlebox, such as simple traffic bouncing, IP or DNS redirection [1].

C. Secure Inspection with Attributes

In practice, the inspection rules usually include specified attributes for a given suspicious string. They can be categorized as a packet field or a range of positions in the packet payload. For these rules, single token matching can no longer work. One straightforward approach is to reveal the attributes to MB, as adopted by BlindBox [5], so that MB can perform post-inspection after the token is matched.

However, we observe that exposing them to *MB* will violate the secrecy of rules and the confidentiality of traffic payloads. For example, revealing "http_stat_code" in the rule below will tell *MB* that the rule is checking the statues code of HTTP. Since the corresponding message space is limited, the rule string and the matched payload could be compromised.

> Snort rule #746: alert tcp $HTTP_SERVERS$ $HTTP_PORTS -> $EXTERNAL_NET$ any (flow:to_client, established; content:"403"; http_stat_code;)

Besides, revealing the inspection positions indicated in the rule below also threatens the privacy of enterprises, because such kind of inspection always checks the vulnerability of specific network protocols and applications [11]. If a token is matched, *MB* would tell exactly which applications or protocols the endpoint runs.

> Snort rule #3235: alert tcp $EXTERNAL_NET$ any -> $SMTP_SERVERS$ 25 (flow:to_server, established;content:"EMF"; depth:4; offset:40;)

To ensure strong protection on the rules, we tailor the token generation, where the inspection attributes are concatenated to the rule strings. Because the attribute is now transformed with the string together via pseudo-random functions, it is protected while the inspection can still function correctly. Explicitly, the token is generated as below:

$$(t, s) = (F_1(K_1, str||c||field), F_2(K_2, str||c||field))$$

The value of $field$ is "http_stat_code" for the former example rule. Regarding a range of positions in the payload for the latter example, all possible positions should be specified in the value of $field$, i.e., $\{t_{beg}||s_{beg}, \cdots, t_{end}||s_{end}\}$, where

$$(t_i, s_i) = (F_1(K_1, str||c||i), F_2(K_2, str||c||i)), i \in [beg, end]$$

beg is equal to $offset$, and end is equal to $beg + depth - sizeof(str)$. Consequently, total $depth - sizeof(str)$ duplicate acts should be stored in the filter. Meanwhile, the endpoints are required to generate the tokens with the same treatment, the corresponding suspicious string can be detected.

D. Secure Multi-condition Inspection

Some of the rules check multiple strings simultaneously to achieve less false positives [11]. The reason is that many vulnerabilities or malicious behaviors take place under multiple conditions. Thus, our system should also support multiple token inspection as the rule shown below.

> Snort rule #127: alert tcp $EXTERNAL_NET$ any -> $HOME_NET$ 21 (flow:to_server, established; content:"RETR"; content:"passwd";)

One trivial approach is to attach the rule id to its action, i.e., $id||act$. The recovered id will tell whether the matched tokens are related to the same rule or not. Here, we assume that id is a pseudonym, which does not give any information about the content of the rule. However, simply encrypting $id||act$ does not meet the security guarantee. If any one of the tokens

is matched, *MB* will know the action, which unnecessarily reveals more information. Here, we target on the same security strength as previous designs. Namely, only if all related tokens in a rule are matched, the responsive action will be recovered. Otherwise, nothing is revealed.

Encryption through secret sharing: To overcome the above security problem, we adopt an efficient (n, n) secret sharing scheme [21]. Specifically, given a rule with n strings, its action act is treated as a secret. Then $n-1$ random strings $\{p_1, \cdots, p_{n-1}\}$ are generated with the same bit length of act, and $p_n = p_1 \oplus, \cdots, \oplus p_{n-1} \oplus act$, where each p_i is a share of act. Accordingly, each share is placed based on the token t generated from one of the suspicious strings. Only when all the n shares are obtained, act is recovered in the follow way:

$$act = p_1 \oplus, \cdots, \oplus p_n$$

As a result, any strict subset of n shares cannot decrypt act. This design guarantees that if and only if all the suspicious strings are matched, the shares will be recovered, and the rule action will be revealed. Otherwise, the rule is kept secret. Meanwhile, the adopted secret sharing is compatible to the rules of single string inspection. In that case, the share will be the action itself. We emphasize that each share p is still protected via the random mask s.

Inspection: To adapt the secret sharing based encryption, the inspection at *MB* should be updated accordingly. The implementation is stated in Algorithm 2. Given an incoming token (t, s), *MB* computes t_1 and t_2, and attempts to recover the share if the token is matched by XORing s with each entry e in buckets $T_1[t_1]$ and $T_2[t_2]$. At the same time, *MB* leverages an in-memory associative map $\mathcal{M}$ to perform the decryption. In particular, the result after the XOR operation is parsed as $x||y$, where x has the same length of id, and y has the same length of each share p. Then the pair (x, y) is inserted to $\mathcal{M}$. If x exists in $\mathcal{M}$, it indicates that x could be a valid id and y could be one of the shares. In that case, *MB* performs $y \oplus y'$, where y' is the previous share. When all the necessary shares are XORed, the action act will be recovered and executed.

E. Secure Inspection for Other Rule Support

Cross-rule inspection: Normally, some rules may contain same suspicious strings, because some sensitive keywords appear in different contexts or activities. For example, "password" belongs to over ten different rules from Snort community rules [11]. To support cross-rule inspection for same suspicious strings, we propose a novel construction based on the design of encrypted filter. The high-level idea is to introduce another counter to differentiate different rules for a given string, and use a flag to indicate the last matching rule.

Recall that $P_1(t, 1)$ and $P_2(t, 2)$ are previously computed from a token t to seek available entries in the filter. Instead, $P_1(t, 1||c_r)$ and $P_2(t, 2||c_r)$ are computed, where c_r is a counter associated with each distinct suspicious string to differentiate the related rules. Besides, a new mask $P_3(s, c_r)$ is generated for encryption, where P_3 is PRF. By giving (t, s), *MB* can increment c_r for inspection, but it does not know when to stop. Accordingly, we introduce a flag and attach it to the share of action as well as the id. Therefore, when the flag in an entry is correctly recovered, *MB* will know whether it should continue or not. Each entry is constructed as follows:

$$P_3(s, c_r) \oplus (id||p||flag), flag \in \{valid, null\}$$

The flag is assigned as either $valid$ or $null$, where $valid$ means that more rules matching the token need to be checked, and $null$ means that this is the last rule that matches the token. For the rule with the unique string, the flag is set to $null$.

Algorithm 2 Inspect for encryption through secret sharing

Input: $\{t||s\}$: the streaming tokens; $\mathcal{M}$: an associative map.
Output: $\{act\}$: the triggered actions.

1: For each incoming token (t, s), compute $t_1 = P_1(t, 1)$ and $t_2 = P_2(t, 2)$;
2: Perform $e \oplus s$, $\forall e$ in $T_1[t_1]$ and $T_2[t_2]$, and parse the result as $x||y$, where $|x| = |id|$ and $|y| = |p|$;
3: Insert (x, y) to $\mathcal{M}$. If x exists, perform $y \oplus y'$, where y' is the previous value associated with x.
4: Once a valid action act is recovered via the XOR operation, the act is executed.

Cross-connection inspection: Cross-connection inspection is another common way for intrusion detection, e.g., detecting brute-force login shown as follows.

> Snort rule #1633: alert tcp $EXTERNAL_NET$ any -> $HOME_NET$ 110 (flow:to_server, established; content:"USER"; count 30, seconds 30;)

This rule will count the occurrences of "USER", and trigger an alert on the activity of suspicious login when "USER" appears over 30 times within 30 seconds. To enable the inspection across different connections, we further introduce a universal message authentication code mac attached with the action for equality checking, i.e., $mac = P_4(r_c, str)||time$, where P_4 is PRF, r_c is a randomness agreed by all the connections, and $time$ is the time interval. Because r_c is the same for all the connections, $P_4(r_c, str)$ will be matched for the same string str. When a mac is recovered, *MB* will start a timer, and record the occurrences of $P_4(r_c, str)$. Here, we assume that the throughput of token processing that *MB* handles can be comparable to the throughput of network traffic due to the unlimited power of cloud. Otherwise, $time$ should be adjusted to tolerate the delay introduced by inspection.

Discussion: Currently, our design does not cover the support of sophisticated inspection, i.e., regular expression and scripts. Similar to the treatment in BlindBox [5], such operations need post processing after the traffic is decrypted. But unlike BlindBox, which allows *MB* to decrypt the traffic, our design choice is to send the warning and the suspicious packets back to *AS* for the security consideration, who will enforce the endpoints to hand over the SSL key for decryption, or it will stop the connection. In particular, we define a new action called "pcre", and replace the previous action by it in the rules with regular expression. As long as "pcre" is recovered, the packet will be sent back. We do acknowledge that this is the limitation of our system, but we argue that only a very small

portion of packets are matched which require sophisticated inspection, e.g., less than 1% in instruction detection traffic dumps with over 30 million packets shown in our experiment.

F. Initialization Optimization

As mentioned, the initialization for each connection needs fresh private keys to build the encrypted filter so that the equality of tokens among different connections is hidden. Although it takes a short time, e.g., around 100 milliseconds for a ruleset with over 3000 rules, it is still an additional cost for endpoints to establish an encrypted connection. To eliminate the overhead during the initialization, we leverage *AS* to pre-build a set of encrypted filters for multiple incoming connections. Explicitly, the function Build is decoupled from the initialization, and the filters are built for *MB* in an asynchronous process periodically. For implementation, *AS* records the number of established connections, and prepares another set of new filters before the number of connections reach the number of engaged filters. As a result, the inspection on the incoming connections becomes seamless. After the connections are expired, *MB* will evict the outdated filters.

IV. Security Analysis

In this section, we will present rigorous security analysis to demonstrate that *MB* cannot learn the traffic payloads and the rule content when performing DPI over multiple different connections. Specifically, we will prove that the single string inspection protocol $\mathcal{P}$ as depicted in Section III-B is secure against adaptive adversaries, then show that the rest of designs can still guarantee the confidentiality of rules and payloads.

First, we formulate the simulation-based security definition verbatim from [12], [13]. Given the well-defined view of *MB*, i.e., a stateful function $\mathcal{L}$, we prove that $\mathcal{P}$ is $\mathcal{L}$-secure against adaptive chosen-keyword attacks; that is, $\mathcal{P}$ does not yield any semantic information about the payloads and the rules beyond the actions to the suspicious tokens, which are "naturally revealed" by the inspection. The security definition in Definition 1 is given in such a way that any probabilistic polynomial time adversary $\mathcal{A}$ cannot distinguish between the real encrypted filter and the simulated filter, the real randomized tokens and the simulated tokens in the real game $\mathbf{Real}_{\mathcal{A}}(k)$ and a simulation game $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ respectively.

***Definition* 1.** *$\mathbf{Real}_{\mathcal{A}}(k)$: a challenger calls $\mathsf{KGen}(1^k)$ to output private keys $\{K_1, K_2\}$. The adversary $\mathcal{A}$ chooses a ruleset $\mathcal{R}$ for the challenger to create an encrypted filter $\mathcal{F}$ via $\mathsf{Build}(K_1, K_2, \mathcal{R})$, and $\mathcal{A}$ adaptively sends a polynomial number of strings extracted from a set of packets. After that, the challenger responds to $\mathcal{A}$ with corresponding tokens, and $\mathcal{A}$ processes the tokens over $\mathcal{F}$. Finally, $\mathcal{A}$ outputs a bit.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$: $\mathcal{A}$ chooses $\mathbf{R}$, and a simulator $\mathcal{S}$ generates $\widetilde{\mathcal{F}}$ based on $\mathcal{L}(\mathbf{R})$. Then $\mathcal{A}$ adaptively sends a polynomial number of strings extracted from a set of packets. After that, $\mathcal{S}$ responds to $\mathcal{A}$ with simulated tokens, and $\mathcal{A}$ processes the tokens over $\widetilde{\mathcal{F}}$. Finally, $\mathcal{A}$ outputs a bit.

Before presenting the proof, we formalize the view of *MB*: $\mathcal{L}(\mathbf{R}) = (|\mathcal{F}|, |e|, d, \{t||s\}_q, \{act\}_m)$, where $|\mathcal{F}|$ is the size of the encrypted filter, $|e|$ is the bit length of the filter entry, d is the number of entries in a bucket, $\{t||s\}_q$ are q tokens which are adaptively generated, and $\{act\}_m$ are m actions recovered within q tokens. Then we have the following theorem.

***Theorem* 1.** *$\mathcal{P}$ is $\mathcal{L}$-secure against adaptive chosen-keyword attacks if F_1, F_2, P_1, and P_2 are PRF.*

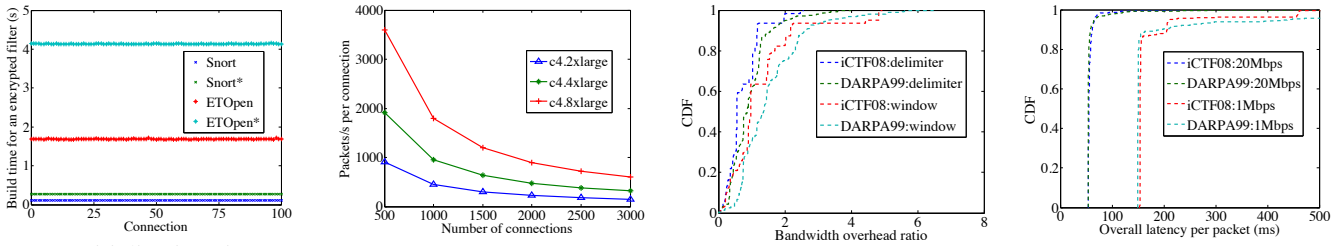
Proof. First of all, $\mathcal{S}$ simulates a filter $\widetilde{\mathcal{F}}$ with the same size of real $\mathcal{F}$ except that each entry in the filter stores a $|e|$-bit random string. In $\widetilde{\mathcal{F}}$, each bucket also contains d entries. Recall that each entry in $\mathcal{F}$ is either encrypted with a random mask or filled with random padding. Due to the pseudo-randomness of F_1, F_2, P_1, and P_2, $\mathcal{S}$ cannot differentiate $\mathcal{F}$ from $\widetilde{\mathcal{F}}$. Then $\mathcal{S}$ simulates the tokens and the inspection results, i.e., the actions. For the first token $t||s$, if there is no action recovered from the accessed 2 buckets from 2 hash tables, and the results are still the random strings, $\mathcal{S}$ will generate random strings $\widetilde{t}||\widetilde{s}$, and operate a random oracle $\mathcal{H}$ that replaces PRFs to find two buckets in $\widetilde{\mathcal{F}}$. $\widetilde{s}$ unmasks the entries in the buckets and no action reveals as well. Thus, $\mathcal{A}$ cannot differentiate the real tokens and results from the simulated ones. If an action act is recovered, $\mathcal{S}$ operates $\mathcal{H}$ to generate $\widetilde{s}$, which unmasks one of the randomly selected entries: $act = \widetilde{s} \oplus \widetilde{e}$. For the subsequent tokens, if the token appeared before, $\mathcal{S}$ uses the identical one simulated previously, or follows the same way of simulating the first token. Therefore, $\mathcal{A}$ cannot differentiate the simulated tokens and results from the real ones. After all, the outputs of $\mathbf{Real}_{\mathcal{A}}(k)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ are indistinguishable. □

Applying symmetric-key based searchable encryption will let *MB* know the repeated tokens, because they are generated via deterministic one-way functions. As proposed, our design improves the security by trading the space. Each rule string is duplicated for C copies associated with an counter. Then *MB* will see different tokens generated within C repeated strings appeared in the connection. We note that if the rule requires cross-rule inspection, the recovered universal mac to some suspicious string will reveal the equality even for different tokens. Yet, it is necessarily required for the functionality, e.g., counting the repeated suspicious strings across connections. And the equality of unmatched tokens is still under protection.

V. Experimental Evaluation

Experiment setup: For evaluation, we select two open-source rules, i.e., Snort default ruleset[2] and ETOpen ruleset[3], and two intrusion detection traffic dumps, i.e., DARPA99[4], and iCTF08[5] with total over 3×10^7 packets. We implement the middlebox module and the endpoint module in C++, and a rule parser with UI in C#. Then we deploy the middlebox module at different models of instances on Amazon Cloud, i.e., "c4.2xlarge", "c4.4xlarge", and "c4.8xlarge", and install

[2]Snort stable release: online at https://www.snort.org/downloads.
[3]ETOpen ruleset: online at https://www.snort.org/downloads.
[4]DARPA datasets: online at http://www.ll.mit.edu/ideval/data/.
[5]iCTF traffic: online at http://ictf.cs.ucsb.edu/pages/the-2008-ictf.html.



(a) Initialization time (b) Inspection throughput (c) Token overhead (d) Introduced latency

Fig. 3: The performance evaluation of our proposed system

Traffic	ETOpen			Snort		
	match	support	pcre	match	support	pcre
iCTF08	0.7%	93.8%	6.2%	0.2%	96.4%	3.6%
DARPA99	0.2%	15.3%	84.7%	0.3%	97.6%	2.4%

TABLE I: Statistics on the support of matched suspicious packets. "match": the percentage of matched packets in the traffic dump; "support": the percentage of packets in total matched packets directly supported by our designs; "pcre": the percentage of packets that needs regular expression.

the endpoint module on a Macbook Pro with Intel Core i7 CPU and 16GB RAM.

Effectiveness evaluation: The percentages of the rules directly supported by our designs are 69% and 52% for Snort and ETOpen respectively, excluding "pcre" rules. As shown in Table I, our designs can handle over 90% matched packets in most cases, where the rest are matched with the "pcre" rules which need to be decrypted for post processing. Yet, the total suspicious matched packets are less than 1% in selected traffic dumps. Even if some packets need to be decrypted at *AS*, they are only a small portion of the overall traffic. Note that the inspection accuracy depends on how DPI systems parse the packet payloads, which is studied intensively in the plaintext domain. Similar to BlindBox [5], we adopt the delimiter-based and window-based string segmentation.

Performance evaluation: First, we evaluate the cost introduced by the encrypted filter. In Table II, the total numbers of string-action pairs extracted from the rulesets are shown. To hide the inspection positions, multiple string-action pairs are generated for each suspicious string with several possible positions, so the number of pairs is much larger than the number of rules. Recall that each encrypted entry stores $id||act||flag||mac$, where each segment in our design is 4-byte long, and the entry is masked by a 16-byte random mask, so each entry is 16-byte long. For the load factor 95%, the encrypted filter for ETOpen with nearly $20K$ rules costs 1.9MB, while the one for Snort with around $3K$ rules only costs 129KB. It is also the bandwidth consumption from *AS* to *MB* to establish a connection. Figure 3-(a) shows that the initialization time for each connection, which is dominated by the build time of encrypted filter. Here, we report the cost in terms of different rulesets, i.e., around 0.1s for Snort and less than 2s for ETOpen. As mentioned, this procedure can be decoupled from the connection initialization since a number of filters can be pre-built. Therefore, *MB* can perform the inspection immediately after the connection is established.

Ruleset	#rules	#pairs	filter size
Snort	3240	7678	129 KB
ETOpen	19528	115503	1945 KB

TABLE II: The space and bandwidth consumption of an encrypted filter per connection. The load factor is set as 95%.

Recall that we propose to index duplicated string-action pairs to hide the repeated tokens. To understand the overhead, we analyze the packets from two traffic dumps with textual content, i.e., counting the maximum occurrences for distinct strings in individual packets. The result shows that 91% of total $10,268$ distinct strings appear no greater than 3 times in a packet. Then we set the number of duplicates for each suspicious string as 3, i.e., the threshold for the counter c defined in Sec. III-B. Accordingly, the size of filter becomes 2 time larger. In future, we will explore to minimize the overhead while protecting the token distribution as much as possible.

The throughput of packet processing is illustrated in Figure 3-(b). We measure the throughput at three models of AWS instances. As our encrypted filter achieves fast and concurrent lookup, the latency to process one token is less than 10μs, and the throughput reaches up to 63×10^6 tokens per second. In the traffic dumps, the average number of tokens per HTTP packet is 35. For a "c4.8xlarge" instance holding 500 connections, the throughput for one connection can reach up to $3,600$ packets per second. For $3,000$ connections, the throughput per connection still achieves 600 packets per second. And using multiple instances will further improve the throughput.

To enable *MB* to perform DPI over encrypted traffic, the endpoints are required to parse the packet content and generate tokens for inspection. Producing and transmitting those tokens indeed introduce computation and bandwidth overheads. Here, we implement delimiter-based and window-based string segmentation, and set the sliding window size as 6 bytes. We also truncate the outputs of HMAC-SHA1 as 10 bytes and 16 bytes for token t and mask s respectively. We also observe that the rules with attributes are usually used to inspect the headers of network protocols [11], so our client needs to generate three tokens for each string in the header for correctness, i.e., the token with the packet field, the token with the offset, and the token with the string only. For other strings, e.g., the strings in the HTTP body, one token with the string only is generated.

As shown in Figure 3-(c), the introduced bandwidth overhead for over 90% packets varies from 3 times to 6 times in

terms of original packet sizes. Figure 3-(d) reports the network latency, including the token generation time, transmission time, and processing time for a given packet. We evaluate the introduced latency per packet under two bandwidth settings, i.e., 20Mbps and 1Mbps with 100ms delay. For a high-speed network, the introduced latency for nearly all of packets is less than 100ms. Under a poor network condition, the latency for over 85% of packets is still less than 200ms, because a large portion of HTTP packets have a small payload size, outputting only dozens of tokens per packet. In a word, our system can perform secure DPI with practical performance.

VI. Related Work

Most of existing intrusion detection systems are unable to conduct full analysis over encrypted traffic [6]. Prior middleboxes decrypt the traffic in the middle of the paths [1], [22], which compromises the confidentiality of payloads, and may constitute man-in-the-middle attacks [10]. Another work [17] performs statistic analysis on encrypted traffic to extract the characteristics and the features of endpoint activities, but those mechanisms cannot detect sophisticated semantic attacks, and thus limit the ability of intrusion detection systems.

Very recently, a middlebox design named BlindBox [5] enables DPI services over HTTPS traffic. An improved design [23] extends BlindBox to support wider middlebox functionalities, and also considers to protect the privacy of enterprises that use cloud-based middlebox services. Different from these designs, our design ensures more delicate protection on the rule sets by handling different types of rules in a tailored manner. Auxiliary information like the inspection positions, fields and so on is also protected, which might be exploited to compromise the confidentiality of rules and payloads. On the other hand, a newly proposed firewall design [7] obfuscates the firewall rules when filtering non-encrypted traffic. And another secure middlebox design [15] also aims to protect both traffic and rules from the middlebox service provider. But the above designs use heavy cryptographic tools like multilinear map [7] and homomorphic encryption [15]. Therefore, it is not clear whether they can achieve the same level of practical performance as our design does.

Our proposed designs are also related to a large number of searchable encryption schemes (to list a few) [12], [13]. They study the problem on how to enable private keyword search over encrypted documents. But as mentioned, directly applying them does not provide comprehensive support of inspection rules or result in a secure design with high throughput and memory efficiency.

VII. Conclusion

In this paper, we design a system that enables outsourced middleboxes to conduct packet inspection while protecting the content of packets and inspection rules. We first formulate the problem as encrypted string matching, and then propose an encrypted filter that securely stores the encrypted string-action pairs extracted from rules. After that, the endpoints parse the packet content and generate randomized tokens so that the middleboxes can process them over the filter for inspection. Our designs support wide range of inspection rules, and the evaluation on real rulesets and traffic demonstrates that our system can efficiently detect most of suspicious packets. In future, we will study the way of handling the regular expression rules over encrypted traffic, and also investigate efficient mechanisms to verify the behavior of middlexboxes.

Acknowledgment

This work was supported in part by the Research Grants Council of Hong Kong (Project No. CityU 138513), the Natural Science Foundation of China (Project No. 61572412), and an AWS in Education Research Grant award.

References

[1] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," in *Proc. of ACM SIGCOMM*, 2012.

[2] A. Gember, R. Grandl, J. Khalid, and A. Akella, "Design and implementation of a framework for software-defined middlebox networking," in *Proc. of ACM SIGCOMM*, 2013.

[3] H. Jamjoom, D. Williams, and U. Sharma, "Don't call them middleboxes, call them middlepipes," in *Proc. of ACM HotSDN*, 2014.

[4] Google, "HTTPS as a ranking signal," http://googlewebmastercentral.blogspot.hk/2014/08/https-as-ranking-signal.html, 2014.

[5] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep packet inspection for encrypted traffic," in *Proc. of ACM SIGCOMM*, 2015.

[6] K. Skarfone and P. Mell, "Guide to intrusion detection and prevention systems," *National Institute of Standards and Technology, available at: http://csrc.nist.gov/publications/nistpubs/800-94/SP800-94.pdf*, 2007.

[7] J. Shi, Y. Zhang, and S. Zhong, "Privacy-preserving network functionality outsourcing," arXiv preprint arXiv:1502.00389, 2015.

[8] A. R. Khakpour and A. X. Liu, "First step toward cloud-based firewalling," in *Proc. of IEEE SRDS*, 2012.

[9] Z. Zhou and T. Benson, "Towards a safe playground for HTTPS and middleBoxes with QoS2," in *Proc. of ACM HotMiddlebox*, 2015.

[10] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson, "Analyzing forged SSL certificates in the wild," in *Proc. of IEEE S&P*, 2014.

[11] Snort, "An open source intrusion prevention system," https://www.snort.org/, 2015.

[12] R. Curtmola, J. A. 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.

[13] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very large databases: Data structures and implementation," in *Proc. of NDSS*, 2014.

[14] B. Fan, D. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. of ACM CoNEXT*, 2014.

[15] L. Melis, H. J. Asghar, E. D. Cristofaro, and M. A. Kaafar, "Private processing of outsourced network functions: Feasibility and constructions," Cryptology ePrint Archive, Report 2015/949, 2015.

[16] S. K. Fayazbakhsh, M. K. Reiter, and V. Sekar, "Verifiable network function outsourcing: requirements, challenges, and roadmap," in *Proc. of ACM HotMiddlebox*, 2013.

[17] A. Yamada, Y. Miyake, K. Takemori, A. Studer, and A. Perrig, "Intrusion detection for encrypted web accesses," in *Proc. of IEEE Advanced Information Networking and Applications Workshops*, 2007.

[18] E. Bertino and G. Ghinita, "Towards mechanisms for detection and prevention of data exfiltration by insiders: keynote talk paper," in *Proc. of ASIACCS*, 2011.

[19] R. Pagh and F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[20] A. Fiat and M. Naor, "Broadcast encryption," in *Proc. of CRYPTO*, 1994.

[21] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons, 2007.

[22] Squid, "Squid-cache feature: HTTPS (HTTP Secure or HTTP over SSL/TLS)," http://wiki.squid-cache.org/Features/HTTPS, 2015.

[23] C. Lan, J. Sherry, R. A. Popa, and S. Ratnasamy, "Mbark: Securely outsourcing middleboxes to the cloud," in *Proc. of USENIX NSDI*, 2016.