PrivDPI: Privacy-Preserving Encrypted Traffic Inspection with Reusable Obfuscated Rules

Jianting Ning*
Fujian Normal University & National University of Singapore
jtning88@gmail.com

Geong Sen Poh
Trustwave & NUS-Singtel Cyber Security Lab
geongsen.poh@trustwave.com

Jia-Ch'ng Loh
NUS-Singtel Cyber Security Lab
dcsljc@nus.edu.sg

Jason Chia
NUS-Singtel Cyber Security Lab
chia_jason96@live.com

Ee-Chien Chang
National University of Singapore
changec@comp.nus.edu.sg

ABSTRACT

Network middleboxes perform deep packet inspection (DPI) to detect anomalies and suspicious activities in network traffic. However, increasingly these traffic are encrypted and middleboxes can no longer make sense of them. A recent proposal by Sherry et al. (*SIGCOMM 2015*), named BlindBox, enables the middlebox to perform inspection in a privacy-preserving manner. BlindBox deploys garbled circuit to generate encrypted rules for the purpose of inspecting the encrypted traffic directly. However, the setup latency (which could be 97s on a ruleset of 3,000 as reported) and overhead size incurred by garbled circuit are high. Since communication can only be commenced after the encrypted rules being generated, such delay is intolerable in many real-time applications. In this work, we present *PrivDPI*, which reduces the setup delay while retaining similar privacy guarantee. Compared to BlindBox, for a ruleset of 3,000, our encrypted rule generation is 288x faster and requires 290,227x smaller overhead for the first session, and is even 1,036x faster and requires 3424,505x smaller overhead over 20 consecutive sessions. The performance gain is based on a new technique for generating encrypted rules as well as the idea of reusing intermediate results generated in previous sessions across subsequent sessions. This is in contrast to Blindbox which performs encrypted rule generation from scratch for every session. Nevertheless, PrivDPI is 6x slower in generating the encrypted traffic tokens, yet in our implementation, the token encryption rate of PrivDPI is more than 17,271 per second which is sufficient for many real-time applications. Moreover, the intermediate values generated in each session can be reused across subsequent sessions for repeated tokens, which could further speedup token encryption. Overall, our experiment shows that PrivDPI is practical and especially suitable for connections with short flows.

*The work was done while the author was at National University of Singapore.

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354204

CCS CONCEPTS

• **Security and privacy → Security protocols**; **Web protocol security**; **Cryptography**.

KEYWORDS

Network privacy; Encrypted traffic inspection; Middlebox privacy

ACM Reference Format:
Jianting Ning, Geong Sen Poh, Jia-Ch'ng Loh, Jason Chia, and Ee-Chien Chang. 2019. PrivDPI: Privacy-Preserving Encrypted Traffic Inspection with Reusable Obfuscated Rules. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3319535.3354204

1 INTRODUCTION

Deep packet inspection (DPI) has been deployed for many functionalities such as intrusion detection, network monitoring, and preventing data leakage. At the same time, it is expected that 80% of all network traffic on the web is encrypted (i.e. SSL/TLS) [11]. This means existing DPI techniques that inspect plain packets would be of limited usage. In order to inspect encrypted traffic, a technique widely used by enterprises is split-TLS using man-in-the-middle (MitM) attack. A middlebox serves as a MitM by establishing a session with the client and the destination server on behalf of the client. By doing so the middlebox is able to decrypt, inspect and re-encrypt the encrypted traffic originated from the client. This provides a practical solution since modification is not required on the underlying security protocol (i.e. TLS).

A system based on the above technique is secure as long as the root certificate required in the MitM approach is securely stored and the TLS protocol implemented is up-to-date. Unfortunately, deployments have been shown to be insecure due to weaknesses in their implementation, such as allowing deprecated cipher suites, as discussed by Jarmoc [18], Carnavalet and Mannan [12], Durumeric *et al.* [14] and Waked *et al.* [28]. Also, as was discussed by Sherry *et al.* [23], a large network with heterogeneous network devices would require many experienced administrators to administer them. This poses another privacy concern when MitM approach is used, since many of these devices may have access to the decrypted data. In addition, MitM violates end-to-end encryption and data privacy guarantee of the supposedly secure two-party communications. The various security issues prompted the US-Cert to issue an alert (TA17-075A) on interception of encrypted traffic [27].

New approaches have been proposed in order to address the issues stated. A recent approach is BlindBox proposed by Sherry *et al.* [24]. BlindBox performs deep packet inspection directly on the encrypted traffic, without the client being able to learn the rules used by the middlebox, and the middlebox not being able to gain any information of the underlying content of the encrypted payloads (except for the content that matches the rules). It introduced a technique called *obfuscated rule encryption* to achieve this, which is based on garbled circuit. Specifically, the client and the server generate a garbled circuit for a function F and perform oblivious transfer (OT) with the middlebox respectively to prepare the encrypted rules. After that, the client tokenizes its payloads and encrypts the payloads to obtain the corresponding encrypted tokens using the same key as was used in preparing the encrypted rules. The middlebox then compares the encrypted rules with the encrypted tokens in the traffic.

However, such garbled circuit approach incurs significant computation and communication overhead, where roughly 97 seconds are required to prepare 3,000 encrypted rules as reported in [24]. Furthermore, for every new session, this operation must be performed again. As noted in [24], BlindBox is not yet practical for short, independent flows with many rules.

1.1 Our Contributions

To address the limitations of BlindBox, we present PrivDPI, which provides a secure, practical solution whereby (a) the encrypted rule generation is relatively more efficient than BlindBox yet retains the same security and privacy guarantee, and (b) introducing a reusable technique so that a client and the middlebox only need to perform rule preparation (i.e., generation of obfuscated rules) once, which is during the first session. The obfuscated rules can then be reused to generate encrypted rules for every new session, thus effectively reducing the computation and communication overhead in rule preparation compared to BlindBox.

To realize our objective, we developed the following techniques:

- We present a new obfuscated rule generation technique that does not use garbled circuit, which achieves a better performance for encrypted rule generation.
- We introduce a *reusable obfuscated rule generation* technique, in which for every Q sessions, the obfuscated rules generated in the first session can be reused in subsequent sessions, which minimizes the computation and communication overhead during encrypted traffic inspection. The capability to reuse the rules further improves the performance of our proposal.

We demonstrate through extensive experiments the improved performance due to our proposed techniques as compared to BlindBox. For the first session on a ruleset of 3,000, the encrypted rule generation of PrivDPI is 288x faster, and requires 290,227x less bandwidth than that of BlindBox. For 20 consecutive sessions on a ruleset of 3,000, the encrypted rule generation is 1,036x faster compared to BlindBox, and requires 3424,505x less bandwidth. The token encryption phase of PrivDPI, however, is roughly 6x slower than that of BlindBox (with AES-NI hardware support). In order to reduce the computation overhead, we record the encrypted tokens generated so far and reuse them in subsequent sessions when the same tokens appear. It is shown in our experiments that the performance of our token encryption phase using this reuse mechanism is only 3.5x slower when all the tokens of the current session have already appeared in previous sessions. For a complete TLS connection on a ruleset of 3,000, our running time is less than BlindBox providing that the number of tokens in both client and server is less than around 3.78 million. This demonstrates that PrivDPI is practical and especially suitable for settings using connections with short flows.

1.2 Use Cases

The main objective of PrivDPI is to enable a middlebox in an enterprise or Internet Service Provider (ISP) to be able to inspect encrypted traffic, while at the same time preserve privacy of the underlying payloads. In this setting, a user can deploy PrivDPI so that he/she is assured that the middlebox has no access to the encrypted information transmitted except if the session has been compromised (i.e. encrypted malware traffic). For example, a subscriber to an ISP service may install PrivDPI so that he/she can be assured that his/her email or Facebook can never be viewed by the personnel that administers the middlebox.

An enterpise may also provide a guarantee to its employees in the scenario where personal data privacy becomes increasingly prevalent (i.e. General Data Protection Regulation (GDPR)). The inspection on encrypted payloads is performed in such a way that not even the administrators have the ability to exploit the existing middlebox setting. We note that an enterprise usually installs or subscribes to security services in order to secure its enterprise networks. Furthermore, the enterprise can deploy PrivDPI to detect potential data exfiltration through keyword matching yet maintaining privacy of the users.

2 OVERVIEW

PrivDPI has a similar architecture as in BlindBox [24] (Fig. 1). It consists of four entities: Rule Generator (**RG**), MiddleBox (**MB**), Client (**C**) and Server (**S**) described as follows.

- **RG**: It issues rule tuples that contain network rules. These rule tuples are used by **MB** to detect malicious network traffic. In particular, each rule contains one or more keywords describing malicious behaviours in the network communication. **RG** can be an organization that provides network security services.
- **MB**: It is a network appliance that monitors the encrypted traffic and try to find malicious data payload(s) in the traffic that match the rule issued by **RG**.
- **C**: It is the user (e.g. the web browser) that sends and receives network traffic. In this work, we focus on encrypted network traffic. In particular, TLS connections.
- **S**: It is the service provider that provides content to the client.

2.1 Threat Model

There are two types of adversaries in the system. The first type of adversaries involves either **C** or **S**. In this case, we assume either **C** or **S** is malicious but not both. Here, the main objective of an adversary is to escape detection of malicious behaviour, e.g., by generating encrypted token that does not correspond to the real

traffic. This is a similar threat setting of a standard intrusion detection system (IDS). We do not consider the case when both **C** and **S** are malicious, since they can just agree on a secret key and encrypt their traffic throughout the entire session. The assumption that either **C** or **S** is malicious is a standard setting for data exfiltration and parental filtering applications [24].

The second type of the adversaries involves **MB**. An adversary of the second type exploits the system in order to learn the content of the encrypted traffic between **C** and **S**. We assume the adversary to be semi-honest (i.e., honest-but-curious), in that it follows the protocol as it is, and only tries to learn the content of the encrypted payload(s). The goal of PrivDPI is to enable **MB** to perform deep packet inspection without exposing the content of the traffic to it.

2.2 System Flow

PrivDPI consists of the following phases.

- **Setup**. In this phase, **MB** receives a set of rule tuples from **RG**. Meanwhile, **C** and **S** establish a session key through the TLS handshake protocol. This session key can be used as a common randomness source for **C** and **S**.
- **Preprocessing**. In this phase, **MB** interacts with **C** and **S** to establish a set of reusable obfuscated rules in such a way that **C** and **S** do not learn the rules while **MB** does not learn the key used by **C** and **S**.
- **Session Rule Preparation**. In this phase, the reusable obfuscated rules generated in the preprocessing phase are used as "seeds" to establish the session rules for the TLS session. The session rules can then be used as a key to generate the encrypted rules for traffic detection (in the following token detection phase).
- **Token Encryption**. **C** first tokenizes the data to be transmitted through the TLS session. Each token is then encrypted, which will be used for matching during the token detection phase.
- **Token Detection**. **MB** first generates encrypted rules using the session rules generated in the session rule preparation phase. It then performs token detection based on the encrypted tokens received from **C** and the encrypted rules it generated. The main characteristic here is that **MB** is able to reuse the obfuscated rules, which is prepared in the first session, in the subsequent sessions.
- **Token Validation**. This phase is required only when either **C** or **S** is malicious in that the data sent through the TLS session and the tokenization session is different. Since **S** (resp. **C**) has the session key, **S** (resp. **C**) may perform the same tokenization and encryption algorithms on the data that it receives and compares the generated tokens to the tokens received from **C** (resp. **S**).

3 PRIVDPI

In this section, we present a basic version of our PrivDPI. This version uses bilinear map to ensure **C** and **S** use the same key to generate tokens, in order to guard against a malicious **C** (or **S**) generating tokens that are different from the TLS payloads to evade detection by **MB**. It is designed with minimal involvement of **S**. This reduces computation and communication requirements on **S** during preparation of obfuscated rules. However, use of bilinear map incurs high computation costs. In Section 4, we will present a variant of PrivDPI that does not need pairing with much better efficiency but require more interactions with **S**. In the following, we provide preliminaries before we describe PrivDPI.

3.1 Preliminaries

3.1.1 Notation. Let PPT be probabilistic polynomial-time. Let $\mathbb{N}$ be the natural numbers. We denote $N \in \mathbb{N}$ as the number of rules, and define $[N]$ to represent the set $\{1,\ldots,N\}$, $[i,j]$ to represent the set $\{i,i+1,\ldots,j\}$. Table 1 lists the notation we used in the system.

3.1.2 Bilinear Maps. Let G and G_T be two multiplicative cyclic groups of prime order p, and g be a generator $\in G$. Let $e : G \times G \rightarrow G_T$ be a bilinear map, which has the following properties: (1) Bilinearity: for all $g \in G$ and $a,b \in \mathbb{Z}_p$, we have $e(g^a,g^b) = e(g,g)^{ab}$; (2) Non-degeneracy: $e(g,g) \neq 1$. The group operation in G and the bilinear map $e : G \times G \rightarrow G_T$ are efficiently computable.

3.1.3 Computational Diffie-Hellman (CDH) Assumption. Given $(g, g^a, g^b) \in G$ for any $(a,b) \in \mathbb{Z}_p^*$, it is hard to compute $g^{ab} \in G$.

3.1.4 Decisional Diffie-Hellman (DDH) Assumption. Given $(g, g^a, g^b, Z) \in G$ for any $(a,b) \in \mathbb{Z}_p^*$, it is hard to decide $Z = g^{ab}$ or Z is a random value.

3.2 Setup

During the setup phase, for a rule set $\{r_i \in \mathcal{R}\}_{i\in[N]}$ (where $\mathcal{R}$ is the rule domain), **RG** chooses a random secret $\alpha \in \mathbb{Z}_p$, random $s_i \in \mathbb{Z}_p$ for each r_i, and sets $A = g^\alpha$, $R_i = g^{\alpha r_i + s_i}$ for $i \in [N]$. **RG** then signs $\{R_i\}_{i\in[N]}$ with its private key to obtain a set of signatures $\{\mathtt{sig}(R_i)\}_{i\in[N]}$. Finally, **RG** sends $(\{s_i, R_i, \mathtt{sig}(R_i)\}_{i\in[N]})$ to **MB**. These rule tuples, which contain keywords known to be used to formulate attacks, will enable **MB** to perform privacy-preserving deep packet inspection at the later stage. Note that this is the only time **RG** is involved in the protocol (except when there are updates on the rules). **RG** can be offline once the above steps are performed. Independently, **C** and **S** install a PrivDPI HTTPS configuration which includes A, the public key of the signature scheme used by **RG**, a hash function H and a value RS. Similar to BlindBox, H is implemented using AES, and RS is a parameter used to reduce the ciphertext size to reduce bandwidth overhead.

Let sk be the session key established from the TLS handshake protocol. As in BlindBox, **C** and **S** then derive the following three keys using sk (via a pseudorandom number generator):

- k_{rand}: will be used as a seed for generating randomness. Note that since **C** and **S** share the same seed, the randomnesses they later generate are the same;
- k: will be used for generating reusable obfuscated rules and session rules[1]. Without loss of generality, we assume $k \in \mathbb{Z}_p$;
- k_{TLS}: is the regular TLS key, and is used to encrypt the traffic.

3.3 Preprocessing

In this phase, a preprocessing protocol will be executed in the *first TLS session* right after the completion of the handshake protocol. The protocol is described in Fig. 2. The aim of this protocol is to establish a set of intermediary, reusable obfuscated rules for **MB**, which will be reused to generate session rules in subsequent

[1]The security requirement of k is that: given k, it is computationally infeasible to recover sk.

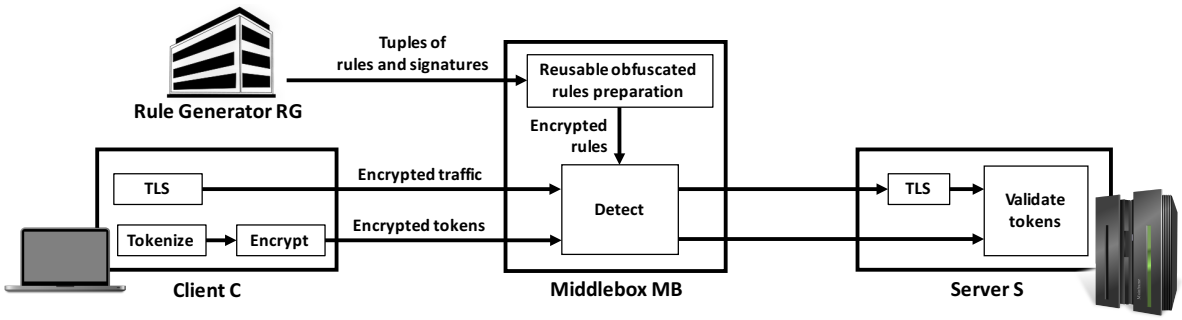

Figure 1 PrivDPI System Architecture: There are two data flows, one the TLS session and another the tokenized data. RG prepares rule tuples and generates signatures for the rule tuples. MB interacts with C and S to prepare the reusable obfuscated rules. MB inspects the tokenized data and forwards the TLS session to the server. The server is able to verify the two data flows are identical since it has the session key used to encrypt the TLS traffic and the tokenized data.

Table 1 Notation

Notation	Meaning	Description
$(s_i, R_i, \mathtt{sig}(R_i))$	Rule tuple	Generated by **RG** for rule r_i as the input to **MB** to generate reusable obfuscated rule
I_i	Reusable obfuscated rule	Generated by **MB** for rule r_i in the preprocessing phase in the first session, used as a source to generate session rule for any new session
S_i	Session rule	Generated by **MB** for rule r_i in the session rule preparation phase, used as the "seed" to generate encrypted rule
T_i	Session token	Generated by **C** (or **S**) for token t_i in the token encryption phase, used as the "seed" to generate encrypted token
C_{r_i}	Encrypted rule	Generated by **MB** for rule r_i in the token detection phase, used for matching
C_{t_i}	Encrypted token	Generated by **C** (or **S**) for token t_i in the token encryption phase, used for matching

sessions. The reusable obfuscated rule of a rule r_i is $I_i = g^{k\alpha r_i + k^2}$, where k, α, r_i are as defined in Section 3.2.

We now explain the rationale behind this protocol. In order to enable **MB** to detect the encrypted traffic, **MB** needs to obtain a function $F_k(r_i)$ which is jointly generated from both k and r_i. However, **MB** should not obtain k and **C** should not know r_i. This is because if **MB** knows k, **MB** will be able to arbitrarily generate encrypted rules to learn the content of the encrypted payloads. Due to the rules r_i being proprietary knowledge that should only be known to **MB** and **RG**, **C** is not supposed to know r_i. To address the above issue, we introduce a new technique called *oblivious reusable obfuscated rule generation*, which is inspired by a recent protocol for oblivious transfer [10]. Our main intuition is for **C** to obfuscate the key k and for the **RG** and **MB** to obfuscate the rules such that an obfuscated rule can be derived without revealing the k or r_i. We did it in a way that enables the obfuscated rules to be reusuable.

In more details, given a group G and its generator g, **C** calculates $K_c = g^k$ using her k and sends K_c to **MB**. Symmetrically, **MB** sends a set $\{R_i = g^{\alpha r_i + s_i}\}_{i\in[N]}$ to **C**. The key observation is that **C** can derive a value $K_i = (R_i \cdot K_c)^k = g^{k\alpha r_i + k s_i + k^2}$ for a rule r_i, and that **MB** is not able to calculate this value assuming CDH assumption holds. This ensures that **MB** can only obtain the set $\{K_i\}_{i\in[N]}$ corresponding to $\{R_i\}_{i\in[N]}$. The set $\{K_i\}_{i\in[N]}$ will be used to generate the reusable obfuscated rules $\{I_i\}_{i\in[N]}$, which are the key components for inspecting the encrypted traffic. However, **MB** cannot inject additional rules $\widehat{R}$ not provided by **RG** and forge $\widehat{K_i} = (\widehat{R_i} \cdot K_c)^k$ for $\widehat{R_i} \notin \{R_i\}_{i\in[N]}$. As a result, **MB** does not have the ability to learn more information from the encrypted traffic except for $\{R_i\}_{i\in[N]}$. There are further, subtle issues that need to be addressed. We explain them in the followings:

1. In order to prevent **MB** from gaining more information on the encrypted traffic by creating $\widehat{R_i} \notin \{R_i\}_{i\in[N]}$ to obtain $\widehat{K_i} = (\widehat{R_i} \cdot K_c)^k$, **MB** needs to send the signatures of $\{R_i\}_{i\in[N]}$ along with $\{R_i\}_{i\in[N]}$. **C** will verify the received $\{R_i\}_{i\in[N]}$ using the signatures received. Since **MB** cannot forge the signature for $\widehat{R_i}$, any additional $\widehat{R_i}$ will be detected by **C**.
2. In order to ensure k in K_c (sent by **C**) and k in K_i (generated by **C**) for rule r_i is identical, **S** needs to calculate $K_s = g^k$ using her k and sends K_s to **MB**. Since **C** and **S** share the same k and one of them is honest, for the case where **C** uses a different k', **MB** can notice this by checking whether K_c equals K_s. For the case where K_i is computed by **C** using a different k', **MB** can notice this by checking whether the equation $e(K_i, g) = e(R_i \cdot K_c, K_c)$ holds. We present a more efficient mechanism to check the correctness of K_c sent by **C** without using bilinear map in section 4, which requires more interactions with **S**.
3. In order to prevent **C** from learning the rules, the rule is blinded by α and s_i chosen by **RG**. In particular, for a rule r_i, **C** receives $R_i = g^{\alpha r_i + s_i}$, where $\alpha, s_i \in \mathbb{Z}_p$. For the case where the rule domain is small, since r_i is blinded by α and s_i, **C** cannot obtain the value of r_i even by launching brute-force attack, even if **C** is computational unbounded.

Preprocessing Protocol

Input: **MB** has inputs $(\{s_i, R_i, \mathtt{sig}(R_i)\}_{i\in[N]})$ (from **RG**), where $R_i = g^{\alpha r_i + s_i}$; **C** and **S** have common inputs k.

Protocol:

1. **C** computes $K_c = g^k$ (using her k), and sends K_c to **MB**. Similarly, **S** computes $K_s = g^k$ (using her k) and sends K_s to **MB**.
2. **MB** checks whether K_c equals K_s. If not, it halts and outputs $\perp$. Otherwise, it sends $(\{R_i, \mathtt{Sig}(R_i)\}_{i\in[N]})$ to **C**.
3. **C** does as follows:
   (a) For $i \in [N]$, check if $\mathtt{Sig}(R_i)$ is a valid signature on R_i using the public key of the signature scheme used by **RG**. If not, halt and output $\perp$.
   (b) Compute $K_i = (R_i \cdot K_c)^k = g^{k\alpha r_i + k s_i + k^2}$ for $i \in [N]$. Finally, send $\{K_i\}_{i\in[N]}$ to **MB**.
4. For $i \in [N]$, **MB** verifies whether the equation $e(K_i, g) = e(R_i \cdot K_c, K_c)$ holds. If not, it halts and outputs $\perp$. Otherwise, for $i \in [N]$, it calculates the reusable obfuscated rule $I_i = K_i/(K_c)^{s_i} = g^{k\alpha r_i + k^2}$ for future use.

Figure 2 Preprocessing Protocol

3.4 Session Rule Preparation

In this phase, a session rule preparation protocol is executed to generate session rule. The session rule is derived from the reusable obfuscated rule generated in the preprocessing protocol. The protocol is described in Fig. 3. The main objective is so that for subsequent sessions, the protocol does not need to re-compute the obfuscated rule. This improves the computation and communication costs as compared to BlindBox, which we demonstrate in our experiment.

In the preprocessing protocol discussed in the previous section, **C** and **S** establish a session key sk and derive (k_{rand}, k, k_{TLS}) in the first TLS session. For a new session, we denote the session key that **C** and **S** established as sk_{new}. Let $(\widehat{k}_{rand}, \widehat{k}, \widehat{k}_{TLS})$ be the new keys derived from sk_{new}.

For the first session, the reusable obfuscated rules $\{I_i\}_{i\in[N]}$ generated in the preprocessing protocol can be used directly as the session rules. For any subsequent sessions, **C** will send $\widehat{K}_c = g^{\widehat{k}}$ specific to the current session to **MB**. **MB** generates the session rules specific to the current session by calculating $S_i = I_i \cdot \widehat{K}_c$ using the reusable obfuscated rules $\{I_i\}_{i\in[N]}$ generated in the preprocessing protocol. In order to prevent **C** from sending a different $\widehat{K}_c$, **S** needs to send $\widehat{K}_s = g^{\widehat{k}}$ to **MB** as well. If a different $\widehat{K}'_c$ is sent by **C**, **MB** will be able to notice this fact by checking whether the messages received from **C** and **S** are the same.

Remark: We note that as the number of session increases, more and more group elements will be consumed. In order to prevent brute-force attack, the preprocessing protocol (Fig. 2) will be run every Q sessions (the value of Q depends on the group we chosen). The session when the preprocessing is performed is treated as the new first session.

Session Rule Preparation Protocol

Input: **MB** has inputs $\{I_i\}_{i\in[N]}$ (from the preprocessing protocol). **C** and **S** have common inputs k (for the first session) or $\widehat{k}$ (for a subsequent session).

Protocol:

Case 1: For the first session, **MB** sets $\{S_i = I_i\}_{i\in[N]}$ as the session rules.

Case 2: For a subsequent session, the protocol operates as follows:

1. **C** sends $\widehat{K}_c = g^{\widehat{k}}$ to **MB**. Similarly, **S** sends $\widehat{K}_s = g^{\widehat{k}}$ to **MB**.
2. **MB** checks whether $\widehat{K}_c$ equals $\widehat{K}_s$. If not, it halts and outputs $\perp$. Otherwise, for $i \in [N]$, it calculates $S_i = I_i \cdot \widehat{K}_c$ as the session rule.

Figure 3 Session Rule Preparation Protocol

3.5 Token Encryption

Sliding window-based tokenization was used by Blindbox and we follow the same approach in our protocol, in which each token consists of 8 bytes. As an example, given a keyword "confidential" which consists of 12 bytes of character, we can generate five tokens in 8 bytes, such that "confiden", "onfident", "nfidenti", "fidentia", and "idential". In order to reduce storage size (as there exists overlay tokens), given a keyword that is more than 8 bytes, **MB** needs only to generate two tokens, a prefix "confiden" and a suffix "idential". This would be sufficient to detect the keyword. We note that an optimization was also proposed in Blindbox, namely delimiter-based tokenization, which is applicable to the HTTP realm by observing how the keywords from rules for these protocols are structured. For example, suppose that given a payload "submit.php?keyword=secret", the possible keywords in rules are "submit", "submit.php", "?keyword=", "keyword=secret", but not "subm" or "submit.p". In practice, we can generate tokens that match keywords that start and end on delimiter-based offsets.

After tokenization, for every token t, token encryption is performed based on the algorithm described in Fig. 4. The intuition is that, since **MB** holds the session rules $(\{S_i = I_i\}_{i\in[N]}$ for the first session or $\{S_i = I_i \cdot \widehat{K}_c\}_{i\in[N]}$ for other session), to perform an equality check, the encrypted token should be derived from $g^{k\alpha t + k^2}$ for the first session or $g^{k\alpha t + k^2 + \widehat{k}_c}$ for the other session. Hence, we require **C** to compute $T_t = A^{kt} \cdot g^{k^2} = g^{k\alpha t + k^2}$ for the first session or $T_t = A^{kt} \cdot g^{k^2} \cdot \widehat{K}_c = g^{k\alpha t + k^2 + \widehat{k}}$ for a subsequent session with key $\widehat{k}$. We call T_t the *session token* for token t.

The security requirement here is that **MB** should be able to match a token t if t equals a rule r hold by **MB**, otherwise, no information is revealed except for the fact that there is no match. To achieve this with high efficiency, a simple approach is to encrypt T_t using a deterministic encryption scheme. However, simply encrypting T_t using a deterministic encryption scheme leaks information that may lead to frequency analysis attack. This is because identical tokens in the encrypted packet stream will have the same session tokens, which will result in identical ciphertexts. Here, we adopt the

approach in BlindBox. We add a random salt to prevent frequency analysis attack. However, naively adding a random salt to each token requires one salt per encrypted token. The salt needs to be sent to **MB** in order to perform equality check. In order to avoid sending an independent salt for each encrypted token, **C** initializes a counter table $\mathtt{CT}_c$ that will record a tuple (t, T_t, ct_t) for every token t, where T_t is the session token corresponding to t and ct_t is the times t appeared in the stream per session so far. In addition, **C** derives a value $\mathtt{salt}$ as the initial salt using k_{rand} and sends it to **MB**, which **MB** will record. For every token t, we consider the following two cases:

1. For the first session, if there does not exist a tuple corresponding to t, calculate $T_t = A^{kt} \cdot g^{k^2}$, set $ct_t \leftarrow 0$, add the tuple (t, T_t, ct_t) into $\mathtt{CT}_c$, and compute the encrypted token as $H(\mathtt{salt} + ct_t, T_t)$. Otherwise, i.e., there exists a tuple $(t', T_{t'}, ct_{t'})$ in $\mathtt{CT}_c$ such that $t' = t$, set $ct_{t'} \leftarrow ct_{t'} + 1$, and compute the encrypted token as $H(\mathtt{salt} + ct_{t'}, T_{t'})$.
2. For other session, first compute $T_t = A^{kt} \cdot g^{k^2} \cdot \widehat{K}_c$. If there does not exist a tuple corresponding to t in $\mathtt{CT}_c$, set $ct_t \leftarrow 0$ and add tuple (t, T_t, ct_t) into $\mathtt{CT}_c$, compute the encrypted token as $H(\mathtt{salt} + ct_t, T_t)$. If there exists a tuple $(t', T_{t'}, ct_{t'})$ in $\mathtt{CT}_c$ such that $t' = t$ and $T_{t'} \neq T_t$, set $T_{t'} \leftarrow T_t$, $ct_{t'} \leftarrow 0$, compute the encrypted token as $H(\mathtt{salt} + ct_{t'}, T_{t'})$. If there exists a tuple $(t', T_{t'}, ct_{t'})$ in $\mathtt{CT}_c$ such that $t' = t$ and $T_{t'} = T_t$, set $ct_{t'} \leftarrow ct_{t'} + 1$, the encrypted token as $H(\mathtt{salt} + ct_{t'}, T_{t'})$.

Token Encryption Algorithm

Input: A counter table $\mathtt{CT}_c$, a token t, and k (for the first session) and $\widehat{K}_c$ (for a subsequent session), $A = g^{\alpha}$.

Algorithm:

Case 1: For the first session, the algorithm operates as follows:

1. For every token t, there are two cases:
   - If there does not exist a tuple corresponding to t in $\mathtt{CT}_c$: compute $T_t = A^{kt} \cdot g^{k^2} = g^{k\alpha t + k^2}$, set $ct_t \leftarrow 0$, $ct \leftarrow ct_t$, and insert tuple (t, T_t, ct_t) into $\mathtt{CT}_c$.
   - If there exists a tuple $(t', T_{t'}, ct_{t'})$ in $\mathtt{CT}_c$ satisfying $t' = t$: set $T_t \leftarrow T_{t'}$, $ct_{t'} \leftarrow ct_{t'} + 1$ and $ct \leftarrow ct_{t'}$.
2. Compute the encryption of t as $C_t = H(\mathtt{salt} + ct, T_t)$.

Case 2: For a new session (different from the first session), the algorithm operates as follows:

1. Compute $T_t = A^{kt} \cdot g^{k^2} \cdot \widehat{K}_c = g^{k\alpha t + k^2 + \widehat{k}}$.
2. For every token t, we have the following cases:
   - If there does not exists a tuple corresponding to t in $\mathtt{CT}_c$: set $ct_t \leftarrow 0$, $ct \leftarrow ct_t$ and insert tuple (t, T_t, ct_t) into $\mathtt{CT}_c$.
   - If there exists a tuple $(t', T_{t'}, ct_{t'})$ in $\mathtt{CT}_c$ satisfying $t' = t$ and $T_{t'} \neq T_t$: set $T_{t'} \leftarrow T_t$, $ct_{t'} \leftarrow 0$ and $ct \leftarrow ct_{t'}$.
   - If there exists a tuple $(t', T_{t'}, ct_{t'})$ in $\mathtt{CT}_c$ satisfying $t' = t$ and $T_{t'} = T_t$: set $ct_{t'} \leftarrow ct_{t'} + 1$ and $ct \leftarrow ct_{t'}$.
3. Compute the encryption of t as $C_t = H(\mathtt{salt} + ct, T_t)$.

Figure 4 Token Encryption algorithm

Similar to BlindBox, in order to prevent $\mathtt{CT}_c$ from growing too large, **C** can reset $\mathtt{CT}_c$ and sends a new salt $\mathtt{salt}'$ to **MB**, where $\mathtt{salt}' = \mathtt{salt} + max_t ct_t + 1$.

3.6 Token Detection

To perform equality check, **MB** only needs to compute the encrypted rule $C_{r_i} = H(\mathtt{salt} + ct_{r_i}, S_i)$ and check whether C_t equals C_{r_i}. This is possible because as shown in Step 2 of Fig. 4, the encrypted token for token t generated by **C** is $C_t = H(\mathtt{salt} + ct_t, T_t)$, where $T_t = g^{k\alpha t + k^2}$ for the first session and $T_t = g^{k\alpha t + k^2 + \widehat{k}}$ for other session with $\widehat{k}$. **MB**, as shown in Fig. 3, holds the session rule set $\{S_i = I_i\}_{i \in [N]}$ for the first session and $\{S_i = I_i \cdot \widehat{K}_c\}_{i \in [N]}$ for the other session. To reduce the bandwidth and detection overhead, we use a counter table and search tree as used in BlindBox. Specifically, for each session, **MB** initializes a count table $\mathtt{CT}_{mb}$ that contains a tuple (ct_{r_i}, C_{r_i}) for every rule r_i, where ct_{r_i} is the count of r_i and is set to be 0, and $C_{r_i} = H(\mathtt{salt} + ct_{r_i}, S_i)$. **MB** also initializes a fast search tree that contains $\{C_{r_i}\}_{i \in [N]}$. The token detection algorithm is described in Fig. 5.

Token Detection Algorithm

Input: A counter table $\mathtt{CT}_{mb}$ and a fast search tree.

Scheme: For each encrypted token C_t in the traffic stream, if there exists a C_{r_i} equals C_t for some $i \in [N]$, do

1. Take the corresponding action dictated by the security policy.
2. Delete the node in tree corresponding to r_i and insert $C_{r_i} = H(\mathtt{salt} + ct_{r_i} + 1, S_i)$.
3. Set $ct_{r_i} \leftarrow ct_{r_i} + 1$.

Figure 5 Token Detection Algorithm

3.7 Token Validation

S runs the same tokenization and token encryption algorithm on the decrypted traffic from TLS as **C** does. **S** then checks the resulting encrypted tokens are the same as the encrypted tokens received from **MB**. If not, it concludes that **C** is malicious.

4 VARIANT OF PRIVDPI

In order to improve the efficiency of PrivDPI, we introduce the following methods by modifying the token encryption algorithm and the preprocessing protocol, respectively.

4.1 Enhanced Token Encryption Algorithm

We introduce two approaches to reduce the computational overhead of the token encryption algorithm in Section 3.5, which are based on the following observations:

1. For a token t, the main task in this algorithm is to compute $T_t = A^{kt} \cdot g^{k^2}$ for the first session or $T_t = A^{kt} \cdot g^{k^2} \cdot \widehat{K}_c$ for other subsequent session. Our first observation is described as follows. For the first session, $T_t = A^{kt} \cdot g^{k^2}$ can be written as $T_t = A^{kt} \cdot V$, where $V = g^{k^2}$. For any subsequent token t' in the stream within

the same session (i.e., in the first session), the detection token $T_{t'}$ can be computed as $A^{kt'} \cdot V'$, where $V' = V$. That is, we can reuse V for any subsequent tokens, the main computation cost is reduced to one exponentiation in G per token. Similarly, for any new token t' within the same session or in any subsequent session, we can reuse V to compute T_t, which only takes one exponentiation in G per token.

2. Our second observation is that in addition to the fact where one token may appear for many times within one session, the same token may appear in subsequent sessions. Hence, we can reuse the session token generated in previous session(s) to reduce the computation overhead of token encryption algorithm.

Based on the first observation, we modify the token encryption algorithm in Section 3.5 as follows. We add one more step prior to any operation, where $V = g^{k^2}$ (for the first session) is computed. The value V is then reused for generating all tokens within the same session or any subsequent session.

Based on the second observation, we modify the token encryption algorithm in Section 3.5 as follows. Before encryption of a token, **C** initializes a counter table $\mathtt{CT}'_c$ that records a tuple $(t, T_t, T_{t,0}, ct_t, ct_s)$ for each token t, where T_t is the session token corresponding to t, $T_{t,0}$ is the session token generated using key k (where k is the key in the first session), ct_t is the number of times t appeared in the stream so far per session and ct_s is the index of the latest session where t appears. In addition, **C** derives a value $\mathtt{salt}$ as the initial salt using k_{rand} and sends it to **MB**, which **MB** will record. For each token t, the modified token encryption algorithm is described in Fig. 6.

Remark: Note that as the table $\mathtt{CT}'_c$ grows, the search time will increase accordingly. Hence, $\mathtt{CT}'_c$ will be reset for every Q sessions. The value of Q will depend on the number of tokens.

4.2 Enhanced Preprocessing protocol

Note that the fourth step of the preprocessing protocol in Section 3.3 requires the pairing operation. In general, the elliptic curve supporting pairing is less efficient than other normal elliptic curves. Hence, if we can remove the need of pairing, we can use a more efficient elliptic curve to reduce the computation overhead. In addition, for $i \in [N]$, $K_i = (R_i \cdot K_c)^k$ can be written as $K_i = (R_i)^k \cdot K$, where $K = (K_c)^k$. Hence, we can (pre-)compute K for once, and reuse K to compute $K_i = (R_i)^k \cdot K$ for $i \in [2, N]$. Thus, the computational cost is reduced to one exponentiation in G per token for $i \in [2, N]$. Based on the above observations, the modified preprocessing protocol is shown in Fig. 7.

5 SECURITY

We first introduce the syntax for our protocol, and then provide the security definition. Finally we prove our protocol is secure based on the defined security definition.

5.1 Syntax

We first introduce the syntax for the class of encryption schemes called middlebox searchable encryption scheme (MBSE for short), as introduced in BlindBox. We then define the syntax for the preprocessing protocol.

Enhanced Token Encryption Algorithm

Input: A counter table $\mathtt{CT}'_c$, a token t, and k (for the first session) or $\widehat{k}$ (for a new session), $A = g^{\alpha}$.

Algorithm:

Case 1: For the first session, the algorithm operates as follows:

1. First compute $V = g^{k^2}$ and store it.
2. For every token t, there are two cases:
   - If there does not exist a tuple corresponding to t in $\mathtt{CT}'_c$: compute $T_t = A^{kt} \cdot V = g^{k\alpha t + k^2}$, set $T_{t,0} \leftarrow T_t$, $ct_t \leftarrow 0$, $ct \leftarrow ct_t$, $ct_s \leftarrow 1$, and insert tuple $(t, T_t, T_{t,0}, ct_t, ct_s)$ into $\mathtt{CT}'_c$.
   - If there exists a tuple $(t', T_{t'}, T_{t',0}, ct_{t'}, ct_s)$ in $\mathtt{CT}'_c$: set $T_t \leftarrow T_{t'}$, $ct_{t'} \leftarrow ct_{t'} + 1$ and $ct \leftarrow ct_{t'}$.
3. Compute the encryption of t as $C_t = H(\mathtt{salt} + ct, T_t)$.

Case 2: For the i-th session where $i > 1$, the algorithm operates as follows:

1. For every token t, there are three cases:
   - If there does not exists a tuple corresponding to t in $\mathtt{CT}'_c$: compute $T_{t,0} = A^{kt} \cdot V = g^{k\alpha t + k^2}$, $T_t = T_{t,0} \cdot \widehat{K}_c = g^{k\alpha t + k^2 + \widehat{k}}$, set $ct_t \leftarrow 0$, $ct \leftarrow ct_t$, $ct_s \leftarrow i$, and insert tuple $(t, T_t, T_{t,0}, ct_t, ct_s)$ into $\mathtt{CT}'_c$.
   - If there exists a tuple $(t', T_{t'}, T_{t',0}, ct_{t'}, ct_s)$ in $\mathtt{CT}'_c$ such that $t' = t$ and $ct_s < i$: compute $T_t = T_{t',0} \cdot \widehat{K}_c$, set $T_{t'} \leftarrow T_t$, $ct_{t'} \leftarrow 0$, $ct \leftarrow ct_{t'}$, $ct_s \leftarrow i$.
   - If there exists a tuple $(t', T_{t'}, T_{t',0}, ct_{t'}, ct_s)$ in $\mathtt{CT}'_c$ such that $t' = t$ and $ct_s = i$: set $T_t \leftarrow T_{t'}$, $ct_{t'} = ct_{t'} + 1$ and $ct \leftarrow ct_{t'}$.
2. Compute the encryption of t as $C_t = H(\mathtt{salt} + ct, T_t)$.

Figure 6 Enhanced Token Encryption algorithm

5.1.1 Definition of MBSE. The syntax of MBSE is defined as follows.

Definition 5.1. An MBSE scheme with message space $\mathcal{M}$ consists of four PPT algorithms (**Setup**, **TEnc**, **REnc**, **Match**) as follows:

- **Setup**(1^λ): The setup algorithm takes as input the security parameter 1^λ, outputs a key k and the public parameter pk.
- **TEnc**$((t_1, ..., t_n), k, pk)$: The token encryption algorithm takes as input a set of n tokens $(t_1, ..., t_n) \in \mathcal{M}^n$, a key k and the public parameter pk, outputs a salt $\mathtt{salt}$ and a set of ciphertexts $(ct_1, ..., ct_n)$.
- **REnc**(r, k, pk): The rule encryption algorithm takes as input a rule $r \in \mathcal{M}$, a key k and the public parameter pk, and outputs an encrypted rule ct_r.
- **Match**$(\mathtt{salt}, (ct_1, ..., ct_n), ct_r)$: The match algorithm takes as input a salt $\mathtt{salt}$, a set of ciphertexts $(ct_1, ..., ct_n)$ and an encrypted rule ct_r corresponding to a rule r, and outputs a set of indexes $I = \{id_1, ..., id_m\}$, where $id_i \in [n]$ for $i \in [m]$.

Correctness. An MBSE scheme ensures that: (1) for every token that matches a given rule, the probability of a match is 1; (2) for every token that does not match a given rule, a match is detected with only negligibly small probability. Specifically, for any sufficiently

Enhanced Preprocessing Protocol

Input: **MB** has inputs $(\{s_i, R_i, \texttt{sig}(R_i)\}_{i\in[N]})$ (from **RG**); **C** and **S** have common inputs k.

Protocol:

1. **C** computes $K_c = g^k$ (using her k), and sends K_c to **MB**. Similarly, **S** computes $K_s = g^k$ (using her k) and sends K_s to **MB**.
2. **MB** checks whether K_c equals K_s. If not, it halts and outputs $\perp$. Otherwise, it sends $(\{R_i, \texttt{Sig}(R_i)\}_{i\in[N]})$ to **C** and **S**.
3. **C** and **S** do as follows respectively:
   (a) For $i \in [N]$, check if $\texttt{Sig}(R_i)$ is a valid signature on R_i using the public key of the signature scheme used by **RG**. If not, halt and output $\perp$.
   (b) Compute $K = (K_c)^k$ (resp. $K = (K_s)^k$ for **S**), which will be reused in the next step.
   (c) Compute $K_i = (R_i)^k \cdot K = g^{k\alpha r_i + k s_i + k^2}$ for $i \in [N]$. Finally, send $\{K_i\}_{i\in[N]}$ to **MB**.
4. For $i \in [N]$, **MB** verifies whether the K_i from **C** and **S** are the same. If not, it halts and outputs $\perp$. Otherwise, for $i \in [N]$, it calculates reusable obfuscated rule $I_i = K_i/(K_c)^{s_i} = g^{k\alpha r_i + k^2}$ for future use.

Figure 7 Enhanced Preprocessing Protocol

large security parameter λ, for any polynomial $n(\cdot)$, if $n = n(\lambda)$, for all $(t_1, ..., t_n) \in \mathcal{M}^n$, for every rule $r \in \mathcal{M}$, for every index id_1 such that $r = t_{id_1}$ and for every id_2 such that $r \neq t_{id_2}$, we have:

$$\Pr\left[\begin{array}{l} k, pk \leftarrow \textbf{Setup}(1^\lambda); \\ \texttt{salt}, (ct_1, ..., ct_n) \leftarrow \textbf{TEnc}((t_1, ..., t_n), k, pk); \\ ct_r \leftarrow \textbf{REnc}(r, k, pk); \\ I \leftarrow \textbf{Match}(\texttt{salt}, (ct_1, ..., ct_n), ct_r) : \\ id_1 \in I \end{array}\right] = 1.$$

and

$$\Pr\left[\begin{array}{l} k, pk \leftarrow \textbf{Setup}(1^\lambda); \\ \texttt{salt}, (ct_1, ..., ct_n) \leftarrow \textbf{TEnc}((t_1, ..., t_n), k, pk); \\ ct_r \leftarrow \textbf{REnc}(r, k, pk); \\ I \leftarrow \textbf{Match}(\texttt{salt}, (ct_1, ..., ct_n), ct_r) : \\ id_2 \in I \end{array}\right] = \texttt{negl}(\lambda).$$

5.1.2 Definition of the preprocessing protocol. The preprocessing protocol is a two-party computation that maps pairs of inputs to pairs of outputs. We refer to the process of the computation as a functionality $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$, where for every pair inputs x_1, x_2, the output pair is $(f_1(x_1, x_2), f_2(x_1, x_2))$. For our preprocessing protocol, the two parties are **C** with input k (the first party) and **MB** with input r (the second party) (the actual protocol is a form of r), and the output is only given to **MB**.

5.2 Security Definition

5.2.1 MBSE Security. The security definition follows that of BlindBox, which is also similar to the security definition of Song et al. [25]. As with BlindBox, our security definition is indistinguishability-based. Specifically, given two sets of tokens, no PPT adversary can distinguish an encryption of one of these two sets of tokens with chance better than half. In addition, for an MBSE scheme, given an encrypted rule, the middlebox is able to tell which encrypted token this rule matches. Hence, in the security definition, we allow the adversary to choose any number of rules and any two sets of tokens of the same length, with the restriction that the two sets of tokens match the set of rules at the same tokens. For an MBSE scheme defined in Section 5.1, its security is defined by the following games between a challenger C and an adversary $\mathcal{A}$:

- **Setup**: C calls the $\textbf{Setup}(1^\lambda)$ algorithm and sends the public parameter pk to $\mathcal{A}$.
- **Challenge**: In this phase, $\mathcal{A}$ chooses two set of tokens $T_0 = (t_1^0, ..., t_n^0)$, $T_1 = (t_1^1, ..., t_n^1)$ and forwards them to C. C flips a random coin $b \in \{0,1\}$ and calls the **TEnc** algorithm with T_b as input to obtain a salt $\texttt{salt}$ and a set of ciphertexts $(c_1, ..., c_n)$. Finally, C sends $\texttt{salt}$ and $(c_1, ..., c_n)$ to $\mathcal{A}$.
- **Query Phase**: $\mathcal{A}$ randomly chooses a set of rules $(r_1, ..., r_l)$ and sends them to C. For $i \in [l]$, C calls the **REnc** algorithm to obtain the encrypted rule ct_{r_i}. Finally, C sends $(ct_{r_1}, ..., ct_{r_l})$ to $\mathcal{A}$.
- **Guess**: $\mathcal{A}$ outputs his guess $b' \in \{0,1\}$ for b.

Let I_0, I_1 be the sets of indexes that match any rule $r_i \in (r_1, ..., r_l)$, respectively. We say that the adversary wins the above game if $I_0 = I_1$ and $b' = b$ for all $i \in [l]$.

Definition 5.2. The MBSE scheme is secure if all PPT adversaries have at most a negligible advantage in λ in the above security game, where the advantage of an adversary $\mathcal{A}$ is defined as $Adv_{\mathcal{A}} = \Pr[(b' = b)] - \frac{1}{2}$.

5.2.2 Preprocessing Security. PrivDPI employs a preprocessing protocol, where **MB** obtains a set of reusable obfuscated rules that will be used in subsequent sessions. There are two security requirements for the preprocessing protocol: (1) **MB** should not be able to compute the reusable obfuscated rule of any new rule that is different from the rules being processed in the preprocessing protocol; (2) **C** cannot obtain what the rules are. Intuitively, after the execution of the protocol, if **MB** cannot obtain any information about k, we can conclude that the first security requirement is satisfied. Similarly, if **C** cannot obtain any information about r, then the second security requirement is satisfied. In the threat model described in Section 2.1, **MB** is semi-honest in the sense that it will follows the protocol. In the preprocessing protocol described in Section 3.3, all malicious behaviors of **C** will be detected by **MB** using the messages received from **S**. Hence, we can treat **C** as semi-honest in the two-party computation with **MB**. We here use the definition of security in present of static semi-honest adversaries in [9, 16]. Let π be a two-party protocol for computing the functionality f defined in Section 5.1. Let $\texttt{view}_i^\pi$ be the view of the ith party ($i \in \{1,2\}$) during an execution of a protocol π on input (x_1, x_2) and security parameter λ, which consists of its input x_i, its internal random coins c_i and the messages that it received. Let $\texttt{Output}^\pi$ be the joint output of the two parties from an execution of π.

Definition 5.3. Let $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ be a functionality. We say that π securely computes f in the presence

of static semi-honest adversaries if there exist PPT algorithms $\mathcal{S}_1$ and $\mathcal{S}_2$ such that

$$\{\mathcal{S}_1(x_1, f_1(x_1,x_2)), f(x_1,x_2)\} \stackrel{c}{\equiv} \{\mathtt{view}_1^{\pi}(x_1,x_2), \mathtt{output}^{\pi}(x_1,x_2)\},$$

$$\{\mathcal{S}_2(x_2, f_2(x_1,x_2)), f(x_1,x_2)\} \stackrel{c}{\equiv} \{\mathtt{view}_2^{\pi}(x_1,x_2), \mathtt{output}^{\pi}(x_1,x_2)\}.$$

where $x_1, x_2 \in \{0,1\}^*$ such that $|x_1| = |x_2|$.

Note that the above definition considers the joint distribution of the output of $\mathcal{S}_1$, $\mathcal{S}_2$ and the parties, it works for the general case of probabilities functionalities. For the deterministic functionalities, we separate the correctness and privacy requirements. The security definition for deterministic functionalities is shown below.

Definition 5.4. Let $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ be a deterministic functionality. We say that π securely computes f in the presence of static semi-honest adversaries if (1) $\mathtt{Output}^{\pi}(x_1,x_2) = f(x_1,x_2)$; (2) there exist PPT algorithms $\mathcal{S}_1$ and $\mathcal{S}_2$ such that

$$\{\mathcal{S}_1(x_1, f_1(x_1,x_2))\} \stackrel{c}{\equiv} \{\mathtt{view}_1^{\pi}(x_1,x_2)\},$$

$$\{\mathcal{S}_2(x_2, f_2(x_1,x_2))\} \stackrel{c}{\equiv} \{\mathtt{view}_2^{\pi}(x_1,x_2)\}.$$

where $x_1, x_2 \in \{0,1\}^*$ such that $|x_1| = |x_2|$.

5.3 Construction

5.3.1 Construction of MBSE scheme. We provide a construction that outlines the main structure below from the security context, in a similar way as BlindBox. In this treatment, we do not include the underlying data structure since it does not affect security. We also do not include the preprocessing protocol that generates the set of reusable obfuscated rules and the session rule preparation phase. The security of preprocessing protocol will be provided independently. In addition, since session reuse does not affect security, we only consider the scenario of the first session.

- **Setup**(1^{λ}): Generate $k, \alpha \in \mathbb{Z}_p$, compute $A = g^{\alpha}$, and set k as the key and A as pk.
- **TEnc**$((t_1,...,t_n), k, pk)$:
  1. Let salt be a random salt and set $ct = 0$.
  2. For each $i \in [n]$, do:
     (1) Set ct be the number of times t_i appeared in the sequence $t_1,...,t_{i-1}$.
     (2) Compute $T_{t_i} = A^{kt} \cdot g^{k^2}$, $C_{t_i} = H(\mathtt{salt} + ct, T_{t_i})$.
  3. Output salt, $\{C_{t_i}\}_{i\in[n]}$.
- **REnc**(r, k, pk): Output $T_r = g^{k\alpha r + k^2}$.

5.3.2 The preprocessing protocol. Any malicious behaviors of **C** can be verified by the messages sent from **S**, hence, in this simplified version of preprocessing protocol, we remove the messages sent for the aim of verification and assume **C** is semi-honest. As introduced in Section 2.1, **MB** is semi-honest, the only malicious act within the semi-honest model is to send additional $R_i^* \notin \{R_i\}_{i\in[N]}$, however, this can be detected and verified using the signatures sent from **MB** trivially. Hence, we do not include the signatures here. The simplified preprocessing protocol is two-party computation in the presence of static semi-honest adversaries. The protocol is described in Fig. 8.

Simplified preprocessing protocol

Inputs: **C** has $k \in \mathbb{Z}_p$, **MB** has $\{s_i, R_i,\}_{i\in[N]}$, where $R_i = g^{\alpha r_i + s_i}$, $s_i \in \mathbb{Z}_p$, $\{r_i \in \mathcal{R}\}_{i\in[N]}$, $\mathcal{R}$ is the domain of rules.

Protocol:

1. **C** computes $K_c = g^k$ and sends K_c to **MB**.
2. **MB** sends $\{R_i\}_{i\in[N]}$ to **C**.
3. **C** computes $K_i = (R_i \cdot K_c)^k = g^{k\alpha r_i + k s_i + k^2}$ for $i \in [N]$, and sends $\{K_i\}_{i\in[N]}$ to **MB**.
4. For $i \in [N]$, **MB** calculates the reusable obfuscated rule $I_i = K_i/(K_c)^{s_i} = g^{k\alpha r_i + k^2}$ for future use.

Figure 8 Simplified preprocessing protocol

5.4 Security Proof

5.4.1 Security of MBSE Scheme.

Theorem 5.5. *Assuming that H is a programmable random oracle, the construction of MBSE scheme in Section 5.3 is a secure MBSE scheme.*

The proof of this theorem is given in Appendix A.1.

5.4.2 Security of Preprocessing Protocol.

Lemma 5.6. *No computationally unbounded algorithm $\mathcal{A}$, engaging in the role of **C** in the execution of the simplified preprocessing protocol, can guess r_i with probability greater than $1/|\mathcal{R}|$ with input R_i.*

The proof of this lemma is given in Appendix A.2.

Theorem 5.7. *Assume that DDH problem is hard. Then, the preprocessing protocol securely computes the functionality f in the presence of static semi-honest adversaries.*

The proof of this theorem is given in Appendix A.3.

6 EXTENSIONS OF PRIVDPI

The protocol proposed in Section 3 supports single keyword inspection, which is the simplest case. In this section, we discuss briefly how our protocol can be extended, just as in BlindBox, to support (1) a limited form of IDS, (2) a full IDS based on retrieval of the session key from a matched suspicious keyword (known as probable cause privacy in BlindBox). This reflects the flexibility of our protocol, in that PrivDPI does not sacrifice the practical and useful features in order to achieve marked improvement in efficiency and reusability. We refer readers who are interested in the details to BlindBox [24, Section 4].

6.1 Supporting limited form of IDS

PrivDPI can be extended to support matching of multiple keywords as well as absolute and relative offset information within the encrypted packet. For example, consider a rule with three keywords, then it is a match if all three keywords appeared in the flow. Industrial rules also contain offset information (e.g. between 10 and 13) and, for example, a rule can be triggered if the content contains certain keywords at the offset of 10-13. The window-based tokenization approach that we deployed generates token at each offset and hence can be used to support the above feature.

6.2 Supporting full IDS

This extension enables full IDS functionality by allowing **MB** to decrypt the TLS traffic when a token matches a keyword in the rule. This allows inspection such as regular expression, that cannot be achieved based solely on token matching. The basic idea is as follow.

Insight. For token t, replace the encrypted token $C_t = H(\texttt{salt} + ct, T_t)$ in PrivDPI with $C_t = H(\texttt{salt}+ct, T_t) \oplus k_{TLS}$. In this manner, it is possible to retrieve the session simply by XORing the matched encrypted token with the encrypted rule. However, in this case, one cannot use the rule tree to do the simple tree lookup. In contrast, it will need a linear scan of the rules in order to perform the XOR operation.

Construction. To maintain efficiency during detection, we use a similar method as BlindBox. In particular, for a token whose encrypted token is $C_t = H(\texttt{salt}+ct, T_t)$, we generate an additional parameter as $C'_t = H(\texttt{salt} + ct + 1, T_t) \oplus k_{TLS}$. Now one can use the rule tree to do the tree lookup for matching. If a match is found, one can compute $C''_t = H(\texttt{salt} + ct + 1, T_t)$ and compute $C''_t \oplus C'_t$ to obtain k_{TLS}.

7 PERFORMANCE EVALUATION

Our experiments were performed by implementing the variant of PrivDPI (Section 4) on a Intel(R) Core(TM) i7-8750H CPU with 6 cores running at 2.20GHz under 64-bit Linux OS. The CPU supports AES-NI instructions. PrivDPI is built on Charm-crypto, a python library that was used to prototype cryptosystems. We also utilize pyOpenSSL library for establishing TLS connection between **C** and **S**. Our system was constructed based on prime256v1 curve (known as NIST Curve P-256), which is widely adopted in most of real-world applications nowadays. We use the rule sets from Snort Emerging Threats with around 3,000 rules. Each rule is tokenized to a size of **8** bytes in our evaluation. We measure the time taken by repeating each instance 10,000 times and eventually take the average of the results. In order to compare the performance of PrivDPI and BlindBox, we reconstruct BlindBox with JustGable [6] and OTExtention [1] to simulate **C** (and **S**) and **MB** in performing the preprocessing protocol for the preparation of encrypted rules.

7.1 Middlebox

The main computation and communication overhead for the middlebox is in the execution of the pre-processing protocol and token detection, which we discuss in the followings.

7.1.1 Performance (first session). Here we consider the time and bandwidth costs for the first session established between the client, the middlebox and the server.

Time. Table 2 shows the total time in the pre-processing phase for the preparation of encrypted rules. PrivDPI is more efficient computation-wise than BlindBox. In particular, PrivDPI takes 0.64 seconds with 3,000 rules compared to BlindBox 183.83 seconds. In other words, for 3,000 rules, PrivDPI is 288x more efficient. The reason behind this is that BlindBox requires one garbled circuit per rule while PrivDPI only requires one exponentiation in G per rule.

Bandwidth. As shown in Table 2, BlindBox requires 50.16GB for generating 3,000 encrypted rules. PrivDPI achieves better result with 172.83KB. This is because Blindbox sends the messages of garbled circuit per rule, while PrivDPI only need to send a few group elements per rule, which only incurs very low bandwidth.

Token detection. The time for token detection is shown in Table 3. The mechanisms deployed are identical between BlindBox and PrivDPI, hence, the running time are the same.

Table 2 MB: Time and bandwidth (first session)

No. of Rules (8 bytes)	Time		Bandwidth	
	BlindBox	PrivDPI	BlindBox	PrivDPI
1	0.595 s	0.1392 s	16.72 MB	57.61 B
3000	183.832 s	0.6371 s	50.16 GB	172.83 KB

Table 3 MB: Time required in token detection

Detection over No. of Rules	BlindBox	PrivDPI
1 rule, 1 token	7.9 μs	
3000 rules, 1 token	30.2 μs	

Table 4 MB: Time and bandwidth (subsequent session)

No. of Sessions	Time		Bandwidth	
	BlindBox	PrivDPI	BlindBox	PrivDPI
1	183.832 s	0.1586 s	50.16 GB	49 B
5	918.14 s	1.271 s	250.845 GB	0.291 MB
10	1838.33 s	2.074 s	501.69 GB	0.292 MB
20	3674.39 s	3.547 s	1003.38 GB	0.293 MB

7.1.2 Performance (subsequent session). We now consider the time and bandwidth for the subsequent session.

Time. Table 4 shows the running time for 5, 10, 20 consecutive sessions with the generation of 3,000 rules for comparison. For 20 consecutive sessions, PrivDPI takes only 3.547 seconds, while BlindBox requires 3674.39 seconds. Hence, PrivDPI is 1,035x faster than BlindBox. This is due to the fact that BlindBox needs to evaluate a garbled circuit for each rule in each session. In contrast, PrivDPI can reuse the reusable obfuscated rules generated during the preprocessing protocol in the first session, eliminating the need to re-generate the session rules from scratch.

Bandwidth. Table 4 shows the bandwidth for 5, 10, 20 consecutive sessions in the similar settings as before. For 20 consecutive sessions, the bandwidth of PrivDPI is 0.293 MB, while the bandwidth of BlindBox is 1003.38 GB, meaning PrivDPI requires 3424,500x less bandwidth than BlindBox. As in the previous case, this is because BlindBox sends the garbled tables for each garbled circuit for each rule in each session. In contrast, PrivDPI can reuse the reusable obfuscated rules stored after the preprocessing protocol in the first session. For each subsequent session, PrivDPI only has to send one group element per session.

Token detection. As before, the time taken for token detection is comparable with BlindBox due to the use of similar mechanism.

7.2 Client (or Server)

The main computation and communication overhead is in the execution of the token encryption protocol.

Table 5 C (or S): Time and Bandwidth

No. of Rules (8 bytes)	Time	
	BlindBox	PrivDPI
1	0.05110 s	0.00110 s
3000	143.547 s	0.19909 s

7.2.1 Performance (first session). Here we consider time and bandwidth for token encryption in the first session.

Time. The client and server in Blindbox performs garbling of the AES circuit as in the case of the middlebox described above. However, the time required to perform this operation is lesser than the time required to perform the operation in middlebox (cf. Table 2) as the sender do not have to wait for the middlebox to evaluate the garbled circuit. For PrivDPI, although it seems that the workload is the same, the fact that without the need to perform circuit garbling significantly improves the performance for **C** (or **S**). Table 5 shows a comparison between Blindbox and PrivDPI. We observe that more load is on the middlebox as compared to **C** (or **S**) on PrivDPI as the performance is faster by 721x compared to the previous score of 288x during the full setup. As network operators are more incentivized to invest in middleboxes with powerful hardware than client computers, shifting the load from **C** to the middleboxes is highly desirable.

Bandwidth. There is no discernible difference between the bandwidth usage of the middlebox (cf. Table 2) and **C** (or **S**) between both protocols, they have equal bandwith requirements as whatever sent by **C** (or **S**) needs to be received by the middlebox.

Token Encryption. Table 6 shows the running time we measured for 1,000 token in the first session. The running time on the **C** (or **S**) side of PrivDPI is around 6x slower than BlindBox. This is mainly due to the mechanism in BlindBox requiring two AES encryptions, while PrivDPI requires one exponentiation in G, one multiplication in G, and one AES encryption. But we note that in actual client-server communication, repeating tokens are likely to occur, especially for $\mathbf{S}$[2] (e.g., **S** having similar content each time it is accessed by the **C**). The repeating tokens are key to increase the encryption performance of PrivDPI, by storing some parts of the computation in a lookup table. The exponentiation in G and multiplication in G can be eliminated by storing them for future lookup. This means that for every repeating token, only one AES encryption is required. Fig 9 shows the improved performances as the number of tokens repeating increases. Likewise, for any token that repeats itself a number of times, PrivDPI also shows an improvement by storing the result of the corresponding exponentiation and multiplication computation. As shown in Fig 10, as the number of times a token repeating itself increases, the time taken to encrypt the token drops proportionally. This is also a likely real world scenario given that keywords in articles or documents tend to have a high number of ocurrences or the high frequency of common english words like 'something', 'between', 'different', 'together' and 'important'.

7.2.2 Performance (subsequent session). Building on the same philosophy in the previous section, subsequent session(s) between **C** and **S** must also consist some degree of similarity, and we ask the question whether if the same technique will work. We evaluated the token encryption time against the percentage of repeating token. We argue that the scenario is also applicable in the real world as **C** may use **S** as a source of information, while the content of **S** remain the same over multiple sessions (i.e looking up recipes, tutorial or specifications online). The results of our evaluation are shown in Fig 11. One can observe that for a high percentage of tokens being repeated from the previous session, the performance is comparable to that of blindbox despite being relatively slow initially. The speed up is due to the fact that the tokens found in the re-used table only require one multiplication in G and one AES encryption.

[2]This is the case when the traffic is sent from **S** to **C**.

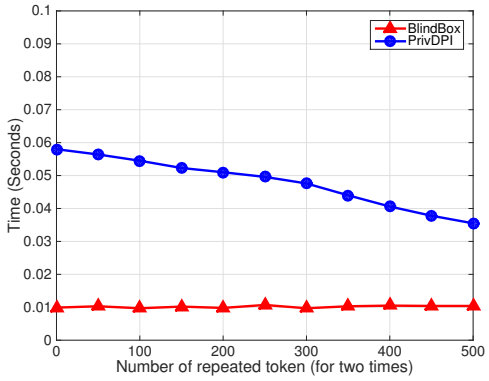

Figure 9 Token encryption time

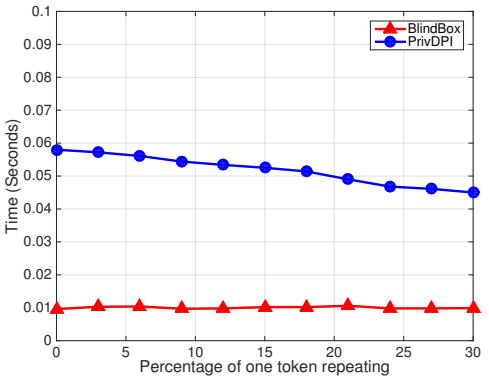

Figure 10 Token encryption time

Table 6 Running time in token encryption

No. of Tokens	BlindBox	PrivBox
1	0.0096 ms	0.0579 ms
500	4.9026 ms	29.0646 ms
1000	10.1182 ms	58.5263 ms

7.3 Performance of a complete TLS session

We also perform a full experiment to evaluate the trade off between pre-processing and token encryption. For this setup, we consider

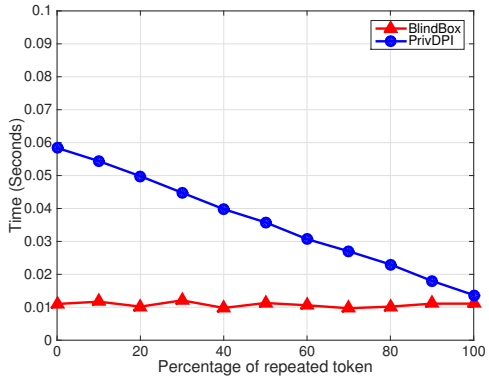

Figure 11 Token encryption time

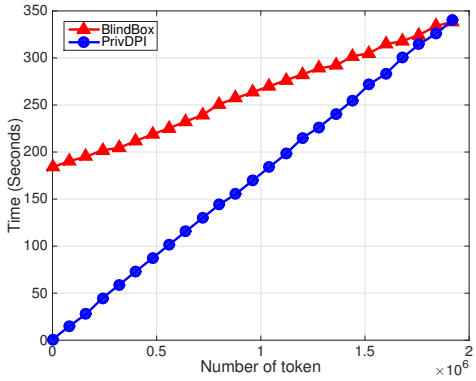

Figure 12 Round Trip Total Time

that no tokens will repeat and only one session (no reuse) occurs between a client **C** and a server **S**. We recorded the time it takes to perform the pre-processing protocol and send a 8 bytes (or 1 token) from **C** to **S** and have **S** reply with a message of the same length. The whole process includes the pre-processing, token encryption and detection from a rule set of 3000 rules. In other words, a round trip of sending 80 bytes (or 10 token) requires 1 setup, 20 token encryptions and 20 token detections to realize. Bearing in mind that usually the client-server request to response bandwidth ratio is generally heavier for the server for general use cases, we chose this setup to reflect on the most extreme of a client-server architecture, namely an echo server to show in what scenario will PrivDPI be advantageous compared to Blindbox.

7.3.1 Analysis. From our experiment, we found that Blindbox will surpass PrivDPI after transmission of 1.89 million tokens from both **C** and **S** that is graphed on Fig 12. That's a total of roughly 3.78 million tokens. To put that into perspective, the required transfer bandwidth for a google search is roughly 0.6-0.8 KB based on different browser tools. Visiting a heavy weight page like youtube.com requires roughly 2 MB per session of transfer bandwidth. This is to say that visiting or performing short-lived queries and lookups to these pages that requires loading only once or twice can be better with PrivDPI. We note that even with a large number of inbound/outbound rules, PrivDPI does not jeopardize the performance of the network traffic and is able to allow a client to quickly establish a session with the middlebox and a server.

8 RELATED WORK

As was stated in our discussion in the introduction, Sherry *et al.* [24] introduced BlindBox, one of the early privacy-preserving deep packet inspection scheme that inspects encrypted traffic directly. The client establishes an encrypted session with the server. A separate session is also established for token matching. Both the sessions route through **MB**. The idea is for **MB** to host encrypted rulesets derived from the session key of the TLS session. For the second session, the client tokenizes and encrypts its payloads using identical key. The tokenized traffic is route through this second session to **MB**. **MB** then tries to match the tokenized traffic with the encrypted rulesets. If matched, the traffic is considered malicious. BlindBox preserves privacy of rules (from **C**) and the secret key derived from session key (from **MB**) through garbled circuit and oblivious transfer for every session, which is computationally expensive.

The tokenization mechanism in BlindBox is extended by Lan *et al.* [19]. The scheme is termed Embark, which is focused on the setting of outsourced **MB**. It proposes new token matching technique that include prefix matching, and caters for different middlebox services such as IP firewall, NAT, HTTP Proxy, data exfiltration and intrusion detection. In order to address the performance bottleneck of BlindBox, Canard *et al.* [8] proposed BlindIDS. It proposes a token-matching protocol based on pairing-based public key operation, and is not compatible with existing TLS protocol.

Another scheme that uses public key operation is SPABox by Fan *et al.* [15]. The scheme uses oblivious pseudo-random function for encrypted rule preparation. For regular expression matching, the scheme deploys a variant of garbled circuit. Yuan *et al.* [29] proposed a scheme that is more efficient than BlindBox but requires the server to first register with the administration service of the enterprise hosting the client. Another scheme, with the cloud setting is SplitBox proposed by Asghar *et al.* [5]. It is based on a two cloud system, in which every rule is XOR with a random string and then split into many blocks to the various **MB**s resided in one of the cloud systems. Both cloud systems then collaboratively compute the blocks to perform traffic inspection.

There are also proposals based on client-server accountable model, where both client and server are aware of and can authenticate all the **MB**s deployed between the two of them. Naylor *et al.* [21] introduced a scheme of this type, termed mcTLS. It modifies existing TLS protocol to allow the client, the **MB**s and the server to establish authenticated and secure channel, and exchange read and write secret keys in addition to the session key. The main issue with mcTLS is that it is a new protocol. For adoptions, existing TLS protocol that is widely used must all be replaced with mcTLS. Acknowledging this issue, Naylor *et al.* [20] further proposed another scheme, termed mbTLS. It does not change the underlying TLS protocol, except in introducing extensions that can be readily adopted using existing protocol. More recently, Bhargavan *et al.* [7]

demonstrated attacks on mcTLS, and proposed a formal model on analyzing the protocol.

Other related work include proposal to analyse encrypted traffic based on machine learning, without inspecting the encrypted payloads. Anderson *et al.* [2–4] proposed techniques for malware detection that uses various header information or metadata. Trusted hardware has also been deployed for privacy-preserving deep packet inspection. Most of the proposal utilizes the secure enclave of Intel SGX. The main idea is to give the trusted hardware, resided in the **MB**, the session key. The decryption, inspection and re-encryption is performed in the enclave. Han *et al.* [17] proposed a scheme, SGX-Box, using this technique. There are also proposals for secure Network Function Virtualization (NFV) systems that use trusted hardware to provide deep packet inspection. These includes SafeBricks by Poddar *et al.* [22], ShieldBox by Trach *et al.* [26] and LightBox by Duan *et al.* [13].

9 CONCLUSION

In this work we proposed a privacy-preserving deep packet inspection system, PrivDPI, which directly inspects encrypted traffic using a similar detection mechanism as in BlindBox. BlindBox incurs high performance overhead for the preparation of encrypted rules due to the use of garbled circuit. Our first key contribution is in minimizing such computation and communication overhead while at the same time preserves the security and privacy requirements as in BlindBox. We demonstrated that our encrypted rule generation on a ruleset of 3,000 is 288x times faster. We further introduce the notion of reusable obfuscated rules, which enables a client and the middlebox to reuse them in subsequent sessions. Such reusable obfuscated rules only need to be generated in the first session, which greatly reduce the performance overhead. We demonstrated that our encrypted rule generation over 20 consecutive sessions on a ruleset of 3,000 is 1036x faster. We note that, the limitation of our system is that the token encryption of payloads is 6x slower than BlindBox. By reusing of encrypted token generated previously, the token encryption is only 3.5x slower in the ideal case that all the tokens of the current session have appeared before. Overall, the computation time for a full session is faster than BlindBox on a ruleset of 3,000 providing that there are less than roughly 3.78 million tokens.

ACKNOWLEDGMENTS

We thank anonymous reviewers for helpful comments. We would like to thank Fuchun Guo for useful discussion on the security proof. Research supported in part by the National Research Foundation, Prime Minister's Office, Singapore, under its Corporate Laboratory@University Scheme, National University of Singapore, and Singapore Telecommunications Ltd., and in part by NSFC 61972094, 61822202, 61872089, 61632012, 61672239.

REFERENCES

[1] OT Extension library. https://github.com/encryptogroup/OTExtension.
[2] Blake Anderson and David A. McGrew. 2016. Identifying Encrypted Malware Traffic with Contextual Flow Data. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security, AISec@CCS 2016, Vienna, Austria, October 28, 2016*, David Mandell Freeman, Aikaterini Mitrokotsa, and Arunesh Sinha (Eds.). ACM, 35–46. https://doi.org/10.1145/2996758.2996768
[3] Blake Anderson and David A. McGrew. 2017. Machine Learning for Encrypted Malware Traffic Classification: Accounting for Noisy Labels and Non-Stationarity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 1723–1732. https://doi.org/10.1145/3097983.3098163
[4] Blake Anderson, Subharthi Paul, and David A. McGrew. 2018. Deciphering malware's use of TLS (without decryption). *J. Computer Virology and Hacking Techniques* 14, 3 (2018), 195–211. https://doi.org/10.1007/s11416-017-0306-6
[5] Hassan Jameel Asghar, Luca Melis, Cyril Soldani, Emiliano De Cristofaro, Mohamed Ali Kâafar, and Laurent Mathy. 2016. SplitBox: Toward Efficient Private Network Function Virtualization. In *Proceedings of the ACM SIGCOMM Workshop on Hot topics in Middleboxes and Network Function Virtualization, HotMiddlebox@SIGCOMM 2016, Florianopolis, Brazil, August, 2016*, Dongsu Han and Danny Raz (Eds.). ACM, 7–13. https://doi.org/10.1145/2940147.2940150
[6] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. 2013. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 478–492.
[7] Karthikeyan Bhargavan, Ioana Boureanu, Antoine Delignat-Lavaud, Pierre-Alain Fouque, and Cristina Onete. 2018. A Formal Treatment of Accountable Proxying over TLS. In *2018 IEEE Symposium on Security and Privacy, SP 2018, San Francisco, CA, USA, May 21-23, 2018*. IEEE Computer Society, 339–356.
[8] Sébastien Canard, Aïda Diop, Nizar Kheir, Marie Paindavoine, and Mohamed Sabt. 2017. BlindIDS: Market-Compliant and Privacy-Friendly Intrusion Detection System over Encrypted Traffic. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi (Eds.). ACM, 561–574. https://doi.org/10.1145/3052973.3053013
[9] Ran Canetti. 2000. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY* 13, 1 (2000), 143–202.
[10] Tung Chou and Claudio Orlandi. 2015. The simplest protocol for oblivious transfer. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 40–58.
[11] Cisco. 2018. Encrypted Traffic Analytics. (2018). https://www.cisco.com/c/dam/en/us/solutions/collateral/enterprise-networks/enterprise-network-security/nb-09-encrytd-traf-anlytcs-wp-cte-en.pdf.
[12] Xavier de Carné de Carnavalet and Mohammad Mannan. 2016. Killed by Proxy: Analyzing Client-end TLS Interception Software. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/killed-proxy-analyzing-client-end-tls-interception-software.pdf
[13] Huayi Duan, Xingliang Yuan, and Cong Wang. 2017. LightBox: SGX-assisted Secure Network Functions at Near-native Speed. *CoRR* abs/1706.06261 (2017). arXiv:1706.06261 http://arxiv.org/abs/1706.06261
[14] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J. Alex Halderman, and Vern Paxson. 2017. The Security Impact of HTTPS Interception. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/security-impact-https-interception/
[15] Jingyuan Fan, Chaowen Guan, Kui Ren, Yong Cui, and Chunming Qiao. 2017. SPABox: Safeguarding Privacy During Deep Packet Inspection at a MiddleBox. *IEEE/ACM Trans. Netw.* 25, 6 (2017), 3753–3766. https://doi.org/10.1109/TNET.2017.2753044
[16] Oded Goldreich. 2009. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press.
[17] Juhyeng Han, Seong Min Kim, Jaehyeong Ha, and Dongsu Han. 2017. SGX-Box: Enabling Visibility on Encrypted Traffic using a Secure Middlebox Module. In *Proceedings of the First Asia-Pacific Workshop on Networking, APNet 2017, Hong Kong, China, August 3-4, 2017*, Kai Chen and Jitendra Padhye (Eds.). ACM, 99–105. https://doi.org/10.1145/3106989.3106994
[18] Jeff Jarmoc. 2012. Transitive Trust: SSL/TLS Interception Proxies. (March 2012). https://www.secureworks.com/research/transitive-trust.
[19] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. 2016. Embark: Securely Outsourcing Middleboxes to the Cloud. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, Katerina J. Argyraki and Rebecca Isaacs (Eds.). USENIX Association, 255–273. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/lan
[20] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. 2017. And Then There Were More: Secure Communication for More Than Two Parties. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2017, Incheon, Republic of Korea, December 12 - 15, 2017*. ACM, 88–100. https://doi.org/10.1145/3143361.3143383
[21] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodríguez Rodríguez, and Peter Steenkiste. 2015. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August*

17-21, 2015, Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye (Eds.). ACM, 199–212. https://doi.org/10.1145/2785956.2787482

[22] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, Sujata Banerjee and Srinivasan Seshan (Eds.). USENIX Association, 201–216. https://www.usenix.org/conference/nsdi18/presentation/poddar

[23] Justine Sherry. 2016. *Middleboxes as a Cloud Service.* Ph.D. Dissertation. Electrical Engineering and Computer Sciences, University of California at Berkeley.

[24] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2015. Blind-Box: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015.* 213–226.

[25] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000.* IEEE, 44–55.

[26] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2018. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the Symposium on SDN Research, SOSR 2018, Los Angeles, CA, USA, March 28-29, 2018.* ACM, 2:1–2:14. https://doi.org/10.1145/3185467.3185469

[27] US-CERT. 2017. HTTPS Interception Weakens TLS Security (Alert TA17-075A). (2017). https://www.us-cert.gov/ncas/alerts/TA17-075A.

[28] Louis Waked, Mohammad Mannan, and Amr M. Youssef. 2018. The Sorry State of TLS Security in Enterprise Interception Appliances. *CoRR* abs/1809.08729 (2018). arXiv:1809.08729 http://arxiv.org/abs/1809.08729

[29] Xingliang Yuan, Xinyu Wang, Jianxiong Lin, and Cong Wang. 2016. Privacy-preserving deep packet inspection in outsourced middleboxes. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016.* IEEE, 1–9.

A PROOF SKETCH

A.1 Proof of Theorem 5.5

We prove security through one hybrid. In particular, the hybrid replaces the random oracle with deterministic random values, which is based on the property of random oracle. The **TEnc** algorithm is modified to replace H with deterministic random values in the hybrid, which is shown as follows.

$Hybrid.\mathbf{TEnc}((t_1,...,t_n),k,pk)$:

1. Let $\texttt{salt}$ be a random salt and set $ct = 0$.
2. For each $i \in [n]$, chooses a random value U_i in the ciphertext space and set $C_i = U_i$.
3. Output $\texttt{salt}, \{C_i\}_{i\in[n]}$.

Also, the **TEnc** algorithm is modified to output a random value U for each rule r with the restriction that: for any r' such that $r' = r$, the algorithm always output U.

In addition, the random oracle H is programmed as follows. For any hash query with input $\texttt{salt}^*$, $Hybrid.\mathbf{TEnc}(r,k,pk)$ for rule r satisfying $\texttt{salt}^* = \texttt{salt} + ct_i$ and $r = t_i$ for some i, U_i is returned as the response.

Clearly, all ciphertexts output by the **TEnc** algorithm are random given that the pattern of matching between tokens and rules are preserved. For the two sets T_0 and T_1 submitted by the adversary, their patterns are the same. We can conclude that any PPT adversary can only distinguish these two sets by exactly half.

A.2 Proof of Lemma 5.6

Fix any $R_0 = g^{r_0}$ for $r_0 \in \mathbb{Z}_p$, the probability that $R_0 = R_i$ is the probability that $s_i = r_0 - \alpha r_i$. Therefore, $\forall R_0 \in G$, $\Pr[R_0 = R_i] = 1/p$.

A.3 Proof of Theorem 5.7

Since the functionality is deterministic, we use Definition 5.4. We construct a separate simulator for each party ($\mathcal{S}_1$ for **C**'s view and $\mathcal{S}_2$ for **MB**'s view).

We first consider the case that **C** is corrupted. In the protocol, **C** receives no output. $\mathcal{S}_1$ merely needs to generate the view of the incoming messages received by **C**. In the protocol, for a rule r_i, the message that **C** received is R_i. $\mathcal{S}_1$ works as follows: $\mathcal{S}_1$ chooses a random $r_i' \in \mathbb{Z}_p$, compute $R_i' = g^{r_i'}$, and outputs $(k; R_i')$, where R_i' simulates the incoming message from **MB** to **C** in the protocol. By Lemma 5.6, we have that the distribution of R_i (i.e., R_i') sent by $\mathcal{S}_1$ is indistinguishable from a real execution of the protocol.

We next proceed to the case that **MB** is corrupted. In the protocol, for a rule r_i, **MB**'s input is $(s_i, R_i = g^{\alpha r_i + s_i})$ and receives output $I_i = K_i/(K_c)^{s_i} = g^{k\alpha r_i + k^2}$. The first and the third messages in the protocol are the messages that **MB** received. For the first message in the protocol, $\mathcal{S}_2$ chooses a random $k' \in \mathbb{Z}_p$, computes $K_c' = g^{k'}$ and set K_c' as the first message. For the third message in the protocol, for a rule r_i, $\mathcal{S}_2$ chooses a random $k'' \in \mathbb{Z}_p$, computes $K_i' = g^{k''}$ and set K_i' as the third message. For a rule r_i, $\mathcal{S}_2$ outputs $(s_i, R_i; K_c', K_i')$, where K_c', K_i' simulate the incoming message from **C** to **MB** in the protocol. The real view of **MB** in an execution can be written as $(s_i, R_i; K_c, K_i)$. The two distributions are $(s_i, g^{\alpha r_i + s_i}; g^k, g^{k\alpha r_i + k s_i + k^2})$ and $(s_i, g^{\alpha r_i + s_i}; g^{k'}, g^{k''})$. Clearly, if a PPT algorithm $\mathcal{A}$ can distinguish these two distributions, it can breaks DDH assumption.