BlindBox: Deep Packet Inspection over Encrypted Traffic

Justine Sherry Chang Lan Raluca Ada Popa Sylvia Ratnasamy
University of California, Berkeley

ABSTRACT

Many network middleboxes perform *deep packet inspection*, a set of useful tasks which examine packet payloads. These tasks include intrusion detection (IDS), exfiltration detection, and parental filtering. However, a long-standing issue is that once packets are sent over https, the middleboxes can no longer accomplish their tasks because the payloads are encrypted. Hence, one is faced with choosing at most one of two desirable properties: the functionality of the middleboxes and the privacy of encryption.

We propose *BlindBox*, a novel system that for the first time enables both properties together. The approach of BlindBox is to perform the deep-packet inspection *directly on the encrypted traffic*. We demonstrate how BlindBox enables applications such as IDS, exfiltration detection and parental filtering; BlindBox supports real rulesets from both open source (Snort) DPI systems as well as rulesets from industrial DPI systems developed by XYZ Co. and ABC Co..[1] While BlindBox's performance is not yet ready for real deployment, BlindBox is nearly practical and improves performance by more than 10^6 times as compared to a direct application of cryptography.

1 Introduction

Network middleboxes perform a wide range of services on packet payloads, which benefit both end users and network operators. For example, middleboxes run network intrusion detection or IDS (e.g., using Snort [2] or Bro [28]) to detect if packets from a compromised sender contain an attack; they perform data loss/ exfiltration prevention such as searching for watermarks in documents [35] to detect if an insider outsources confidential information; and they perform other tasks such as parental filtering [5].

However, a long-standing problem is that when the traffic is sent over HTTPS, network middleboxes can no longer run these tasks [27] because the payload is encrypted. The encryption provided by HTTPS is useful because it protects private user data or confidential company data from an attacker at the middlebox. Unfortunately, currently deployed middlebox systems support HTTPS in an *insecure* way: some middleboxes mount a man-in-the-middle attack on SSL and decrypt the traffic at the middlebox [18, 16]. This approach violates the end-to-end security guarantees of SSL and opens the door to a set of issues as surveyed in [18]. Moreover, users and clients have expressed criticism and various concerns over this approach [42, 21, 33, 38], including worries that the private data logged at the middlebox is given to marketers or to the government.

Therefore, one is faced with an unfortunate choice of *at most one* of two desirable properties: the functionality of the middleboxes and the privacy afforded by encryption.

It would be ideal to have a system that provides both the privacy of encryption and the functionality of middleboxes. Unfortunately, there is a strong tension between these two properties, making such an ideal solution seem impossible.

In this paper, we demonstrate that it is possible to build such a system. We propose the first system that provides *both* the benefits of encryption and the functionality at the middlebox. The system is called *BlindBox* to denote that the middlebox cannot see the private content of the traffic, despite being able to operate on it. BlindBox can support the functionality of a wide range of applications over HTTP traffic, such as data watermarking [35], parental filtering, and real intrusion detection signatures including signatures from Snort, XYZ Co. IDS, and ABC Co..

Our approach is to perform *the inspection directly on the encrypted payload*, without decrypting the payload at the middlebox. The traffic is encrypted with a special encryption scheme and the middlebox receives some capabilities enabling it to detect matches with rules of interest. BlindBox protects the data with a strong encryption scheme that is randomized (and formalized using indistinguishability-based security definitions, which we present in the Appendix of this document): for example, the middlebox cannot even tell if certain parts of the packet are equal to each other. Nevertheless, the middlebox learns a small amount of information about the traffic necessary to detect matching of rules efficiently: if a *suspicious* string matches at some offset in the traffic, the middlebox learns that the suspicious string matched at that offset, but it does not learn the content of the traffic that does not match suspicious strings.

Although BlindBox's performance is not yet ready for real deployment, BlindBox makes a major step towards practicality by improving performance by more than 10^6 times as compared to a direct application of existing cryptography. The key overhead with BlindBox is the session handshake. Setting up a connection in BlindBox is slowed down by minutes; although this is a substantial improvement over the hours or even days as traditional fully homomorphic or functional encryption schemes would lead to. However, once

[1] Company names anonymized for public release.

the handshake completes, page load times increase by only $2\times$ relative to normal SSL, making BlindBox practical for long-lived connections, but not short/frequent ones. These are the overheads at the client only: at the middlebox itself, BlindBox is actually faster at signature detection than Snort, a standard IDS.

1.1 Challenges and techniques

Building a practical system that provides both encryption and detection is a challenging task for three reasons: performance, security and functionality. BlindBox addresses each of these challenges with a combination of novel networking and cryptographic techniques: a new encryption scheme DPIEnc, a fast detection algorithm BlindBox Detect, enhancing packets with certain offset information, and a new model for traffic privacy called probable cause privacy. We now explain these techniques.

Challenge 1: Performance. Existing encryption schemes that enable computing the desired functionality on encrypted data are prohibitively slow (as described in Sec. 10.2). A naïve approach is to choose an existing encryption scheme that seems most fit to our setting (described in Sec. 9.2.1) and most efficient among the schemes available; such strawman results in more than 10^6 times overhead on top of HTTPS, which is not acceptable.

Instead, we achieve better performance by providing a *novel encryption scheme DPIEnc* and a *novel detection protocol BlindBox Detect* that uses this encryption scheme.

DPIEnc is fast because:

- We designed it custom for this setting. It consists of four building blocks: the AES block cipher, Yao garbled circuits [41, 24], hashes, and digital signatures, which are tied together by a careful use of randomness.
- We ensured that the operations that need to run very frequently are fast: encryption is done with a combination of AES and hashing, and detection with equality checks. We achieve this by pushing the main cryptographic work in the less frequent operation of setting up a connection.

Since each of the building blocks has a simple black-box interface, our encryption scheme can be understood easily by a reader with no cryptographic background.

Our detection algorithm BlindBox Detect is fast because it uses DPIEnc in a clever way: it maintains certain state at the endpoints and middlebox such that it minimizes the number of equality checks performed during inspection of a packet, while maintaining security.

In comparison to the strawman above, we decreased the cryptographic overhead for encryption and detection by 10^6 times – thus making a major step towards practicality. In fact, fortunately, BlindBox's detection throughput is high, twice faster than Snort [2]. However, the time to setup a connection is not yet competitive: it takes 414 s for ABC Co.'s IDS. Nevertheless, this is $1.8 \cdot 10^3$ faster than the strawman's setup for the same security level. Hence, our current system is more fit for persistent connections or SPDY-type [31] connections which perform setup once or rarely. Nevertheless, we believe that we can reduce the setup time in the near future: this time depends mainly on the size and speed of the garbled circuits. Garbled circuits have been continually studied at security conferences and their performance has steadily increased over the years: for example, from 2011 [17] and 2012 [23] to 2013 [8], their performance improved by two orders of magnitude [8]. Overall, we take the first major step towards achieving the vision of supporting both encryption and middlebox in a practical way.

Challenge 2: Security. It turns out that the desired security exposes a contradictory situation. On the one hand, to detect matching of a rule against the encrypted traffic, the rule needs to be encrypted with the same key k as the key for the traffic. On the other hand, it seems that no party in our setup is fit to encrypt the rules under key k: the middlebox cannot give the rules to the endpoints for encryption since they are proprietary (as we explain in Sec. 2.2) and the endpoints cannot give the key k to the middlebox, because the middlebox can then decrypt the traffic.

Our encryption scheme DPIEnc resolves this tension through a new technique called *obfuscated rule encryption*: the endpoints essentially obfuscate the AES encryption algorithm together with the key k and give an obfuscated AES algorithm to the middlebox. The middlebox can run this obfuscated algorithm on the rules to obtain encrypted rules, without learning the key k. This obfuscation is achieved using Yao garbled circuits [41, 24] described in Sec. 3.2.

Challenge 3: Wide range of functionality. Different DPI applications require different functionality. We divide such functionality in three groups: basic string matching, multiple string matching with offset information, and arbitrary regexp and scripts. We provide three protocols, one for each category. Regarding arbitrary regexp on encrypted traffic, there is currently no encryption scheme or protocol that can support *general* operations over encrypted data in a practical way: functional encryption and FHE are at least 9 orders of magnitude slower than regular computation [14]. Hence, to enable regexp efficiently, our protocol for regexp provides a weaker security guarantee than the other two protocols.

The three protocols are not separate, but they provide a technical progression towards the third protocol. Table 1 summarizes these protocols and their relationship.

Protocol I: Basic string matching. This protocol supports watermarking and parantal filtering, and can be implemented with the tools we discussed already, DPIEnc and BlindBox Detect.

Protocol II: Limited IDS. Commonly-used IDS tools such as Snort [2, 37] have rules that match multiple strings and also specify offset ranges where these strings should be matched. We can support this protocol using DPIEnc and BlindBox Detect, along with a protocol for annotating encrypted packets with certain offset information.

Protocol	Applications supported	Techniques used	Security guarantee
I	data loss/exfiltration prevention through watermarking, parental filtering	DPIEnc, BlindBox Detect	MB does not see traffic; learns where *suspicious* strings match
II	limited IDS	as above + add offset information to packets	same as above
III	full IDS	as above + probable cause privacy protocol	probable cause privacy

Table 1: Overview of contributions: the protocols we provide, the functionality they cover, which of our techniques they use and the security guarantee they provide. MB denotes middlebox.

Protocol III: Full IDS. Some IDS such as Snort [2] have rules that contain arbitrary regular expressions. To support these efficiently, we introduce a new model of privacy of the traffic, called *probable cause privacy*, and a protocol to achieve it. This model is inspired from two ideas. First, most rules in Snort that contain regexp first attempt to find a suspicious string in the packet – this string is selective so only a small fraction of the traffic matches this string and gets passed through the regexp. Indeed, the Snort's users manual [37] urges the presence of such selective strings because otherwise, detection would be too slow. Second, we are inspired from the notion of probable cause from United States' criminal law: our idea is that one should give up privacy *only* if there is a reason for suspicion.

Hence, our model says that if a packet contains a *suspicious* keyword from an attack rule, the middlebox should be able to decrypt that stream of traffic; otherwise, the middlebox must not see the traffic. Our protocol ensures that, as soon as the middlebox possesses an attack rule that matches the packet, the middlebox gains the ability to decrypt the stream; then, the middlebox can run regexp and scripts on it. However, if no suspicious string matches the stream, the middlebox *cannot possibly decrypt the traffic*, and the same security guarantees as in the previous two protocols apply. Since the suspicious strings are selective, a small fraction of the traffic is decrypted.

Finally, we have implemented and evaluated our protocols on realistic rulesets such as data watermarking [35], Snort Community IDS, Snort Emerging Threats IDS, XYZ Co. (McAffee) IDS, ABC Co. IDS and Parental Filtering [5], and we already summarized BlindBox's performance above.

2 Overview

Fig. 1 presents the system architecture. There are 4 parties involved in the protocol: two endpoints – the sender S and the receiver R – the middlebox (denoted MB), and a rule generator (denoted RG). RG generates attack rules (also called signatures) to be used by MB in detecting attacks. For example, RG can be Emerging Threats, McAffee, or Symantec. S and R send traffic through MB. We explain the purpose of each module from Fig. 1 in Sec. 2.3.

2.1 Usage scenarios

Before formalizing our threat model, we illustrate our usage scenario with two examples. For each individual in these examples, we indicate the party in our model (R, S, MB, or RG) that they correspond to.

Example #1: University Network. Alice (R or S) is a student at the University of SIGCOMM and brings her own laptop to her dorm room. However, university policy requires that all student traffic be monitored for botnet signatures and illegal activity by a middlebox (MB) running an IDS. Alice is worried about her computer being infected with botnet software, so she wants this policy applied to her traffic. McAfee (RG) is the service that provides attack signatures to the middlebox and Alice trusts it. However, she is uncomfortable with the idea of someone she doesn't know (who has access to the middlebox) potentially being able to read her private Facebook messages and emails.

Alice has installed BlindBox HTTPS with McAfee's public key, allowing the IDS to scan her traffic for SecurityCorp's signatures, but not read her private messages.

Example #2: ISP Service. Bob has two young children (S/R) at home, and registers for parental filtering with his ISP so that all traffic is filtered for adult content. But, Bob has read stories in the news of ISPs selling user browsing data to marketers [38] and wants to prevent his ISP (MB) from using his data in this way. Bob trusts the Electronic Filtering Foundation (RG), a non-profit which generates rulesets for filtering and pledges not to sell user data. Bob installs BlindBox HTTPS on his home computer with the Electronic Filtering Foundation's public key, allowing his traffic to be scanned for EFF rules only, but no other data.

In these scenarios, Alice and Bob want to have a middlebox in their network check *for the signatures the corresponding trusted parties permit*, but this middlebox should not learn anything else about the content of the traffic.

2.2 Threat model

Like in any intrusion detection protocol, the rule generator is assumed to create the attack rules honestly. In some cases, an enterprise purchases a middlebox from the RG already containing the rules. Crucially, RG does not want to disclose these rules to the endpoints because RG's business is based on the rules' secrecy. Moreover, [28] argues that rules should

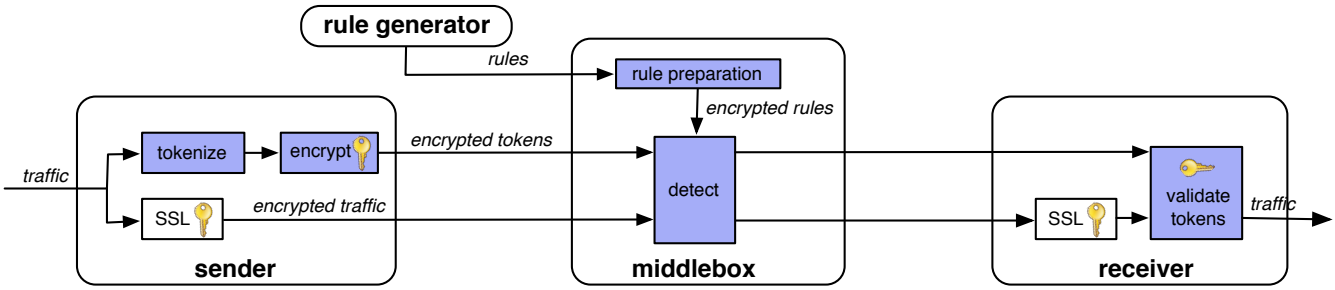


Figure 1: System architecture. Shaded boxes indicate algorithms added by BlindBox.

remain secret for security reasons too. Hence, the middlebox cannot give away the rules to the endpoints.

There are two types of attackers in our setup.

The original attacker considered by IDS. This is the same attacker that traditional (unencrypted) IDS consider and we do not change the threat model here. Our goal is to enable detecting such attacker over *encrypted* traffic. As in traditional IDS, one endpoint can behave maliciously, but at least one endpoint must be honest. This is a fundamental requirement of any IDS [28]; the reason is that the problem is impossible to solve otherwise: two malicious endpoints can always agree on a secret key through an out-of-band channel and encrypt their traffic under that key using a strong encryption scheme, making any prevention impossible by the security properties of the encryption scheme.

The attacker at the middlebox. This is the new attacker in our setting. This attacker tries to subvert our scheme by attempting to extract private data from the encrypted traffic in BlindBox. We assume that the middlebox MB performs the detection honestly, but that it tries to learn private data from the traffic and violate the privacy of the endpoints. In particular, we assume that an attacker at MB reads *all* the data accessible to the middlebox, including traffic logs and other state.

Given this threat model, BlindBox's goal is to hide the content of the traffic from MB, while allowing MB to do DPI. We do not seek to hide the attack rules from the MB itself; many times these rules are hardcoded in the MB.

2.3 System architecture

We now explain each module from our system architecture depicted in Fig. 1. The purpose of these modules is the same in all our three protocols.

In the initial system setup, MB obtained the detection rules from the rule generator. Also, S and R obtained the public key of the rule generator. The rule generator is never again involved in the protocol.

Let us now discuss what happens when a sender and receiver initiate and send traffic on a connection.

Connection setup. First, the sender and receiver run the regular SSL handshake which permits them to agree on a key k_0. The sender and receiver use k_0 to derive three keys (e.g., using a pseudorandom generator):

- k_{SSL}: the regular SSL key, used to encrypt the traffic as in the SSL protocol,
- k: used in our detection protocol, and
- k_{rand}: used as a seed for randomness. Since both endpoints have the same seed, they will generate the same randomness, which enables some later crosschecks.

Next, MB runs the rule preparation algorithm (helped by the sender and the receiver). In this step, MB obtains encryptions of the rules with key k – this will later enable MB to perform the detection.

Sending traffic. When the sender has traffic to send, two things happen. (1) The traffic gets encrypted with SSL as before. (2) The traffic gets tokenized and the resulting tokens get encrypted. The tokenize algorithm, discussed in Sec. 7, transforms the traffic into a set of tokens – these tokens are substrings of the traffic of various lengths taken from various offsets in the traffic. In regular IDS, rules contain strings that should be matched at any offset in the traffic stream. In our case, rules will match against tokens. The encryption module will encrypt each token with a key k.

Detection. The middlebox receives the SSL-encrypted traffic and the encrypted tokens. It ignores the SSL-encrypted traffic. The detect module will search for matchings between rules encrypted with a key k against the tokens encrypted with k using BlindBox Detect (Sec. 5). If there is a match, one can choose the same actions as in a regular (unencrypted IDS) such as drop the packet, stop the connection, or notify an administrator. In some cases, a system administrator may want to inspect the packets manually. However, the packets are now encrypted. If a system administrator needs the traffic to be decrypted if and only if it matched a suspicious content, and if the endpoints agree to run such a protocol, one can use our probable cause protocol (Sec. 6) even for systems that do not need to run regexp computation. After completing detection, MB forwards the SSL traffic and the encrypted tokens to the sender.

Receiving traffic. Two actions happen at the receiver. First, the receiver decrypts and authenticates the traffic using regular SSL. Second, the receiver checks that the encrypted to-

kens were encrypted properly by the sender. This happens in order to establish if the sender encrypted the tokens properly. Recall that, in our threat model, one endpoint may be malicious – this endpoint could try to cheat by not encrypting the tokens correctly or by encrypting only a subset of the tokens to eschew detection at the middlebox. Since we assume that at least one endpoint is honest, such verification will prevent this attack.

3 Protocol I: Basic detection

In this protocol, each rule consists of one string. MB must be able to detect if the string appears at any offset in the traffic.

This protocol suffices for applications such as document watermarking [35] and parental filtering [5]. The goal of watermarking is to prevent an employee from exfiltrating sensitive documents outside of a company. A data owner (e.g., an enterprise) encodes a watermark, which is typically a random serial number, into confidential documents. The middlebox knows these serial numbers and checks if outgoing traffic contains these watermarks. If it finds them, MB likely discovered an exfiltration attack and flags it to an administrator.

Even though this protocol provides the simplest functionality, it contains the meat of our techniques; the other protocols simply build on this protocol. The protocol consists of two parts: our encryption scheme DPIEnc and our detection protocol BlindBox Detect. We describe DPIEnc here and dedicate Sec. 5 to BlindBox Detect.

3.1 Intuition behind DPIEnc

We start by presenting the intuition behind DPIEnc. We remark that a non cryptography-trained reader can understand DPIEnc because it is composed of encryption schemes with a clean black box interface.

As already discussed in the introduction, in the encryption and detection phases, the protocol must perform very simple operations such as equality, hashes or block ciphers for each position in the stream to be tested. Any more complicated operations or cryptography will result in poor throughput.

A promising first idea is to encrypt each token with a deterministic encryption scheme, such as AES. This has the property that encrypting the same value twice results in the same encryption. To illustrate how MB can use this property for detection, consider that MB has a rule consisting of the string r = "attack" and its encryption $\mathsf{AES}_k(r)$ under key k, where k is some connection key. Now consider that the traffic includes the word r. Hence, there will be a token for r and the endpoint will create an encryption for it: $\mathsf{AES}_k(r)$. Now MB only needs to run a simple equality check to identify the presence of r in the stream because the encryption is deterministic. This scheme has promising performance because detection consists from a simple equality and encryption consists of AES. However, this approach has two security shortcomings.

The first issue is that deterministic encryption leaks more about the traffic than the middlebox needs to know to perform the detection efficiently. In particular, it leaks whenever a token repeats. For example, if the encrypted tokens are [AES_k("secret"), AES_k("attack"), AES_k("secret")], the attacker will know that the first and third tokens are equal – even if there is no rule for "secret". An attacker can construct frequency histograms and learn about the data. To address this problem, a natural idea is to randomize the encryption by adding a random salt to each word, for example by having [salt_1, AES(salt_1, "secret"), salt_2, AES_k(salt_2, "attack"), salt_3, AES_k(salt_3, "secret")]. The problem now is that MB can no longer run the detection with AES_k("attack") because it has no way to include the salt in the encryption (the salt cannot be fixed either because it must be different for every occurrence of the word "attack").

Instead, the solution is to note that one does not need to be able to decrypt an encrypted token – hence one can use a hash H on top of the encryption. For a cryptography-trained reader, H is modeled as a random oracle. The resulting traffic is [salt_1, H(salt_1, AES_k("secret")), salt_2, H(salt_2, AES_k("attack")), salt_3, H(salt_3, AES_k("secret")]. Now, MB can perform the detection by using the encrypted rule AES_k("secret") and combining it with salt_2 from the traffic stream to compute H(salt_2, AES_k("attack")).

The second issue is even more challenging. Since each pair of endpoints (or each connection) have a different key k, each rule r at MB must be encrypted into $\mathsf{AES}_k(r)$ using k to enable detection. But who can perform this encryption? One idea is to have the endpoints send the key k to MB so that MB can encrypt. But this means that MB can also decrypt the whole traffic, violating privacy. (Even though the hash H is not decryptable, MB can try a word w at a time, compute $\mathsf{AES}_k(w)$, apply H and see if it matches the encrypted token from the traffic). Hence, another idea is to have MB send the rules to the endpoints so they can encrypt using k. However, by our threat model in Sec. 2.2, the endpoints are not allowed to see the rules. Hence, it seems that there is no party fit to do the encryption.

Instead, our idea is to have the endpoints *obfuscate* the encryption function of AES together with the key k and send this obfuscation to the middlebox. The resulting function $\mathsf{ObfAES}_k()$ hides the key k. When establishing a new connection with key k, the endpoints compute and send ObfAES_k to MB. MB can run this function on each rule r and obtain $\mathsf{ObfAES}_k(r)$. While obfuscation per-se is impossible or prohibitively slow [7, 12], we can achieve the properties we need in this setting using *Yao garbled circuits* [41, 24]. An endpoint can *garble* the AES function to obtain ObfAES_k which hides k. With garbled circuits, MB cannot directly plug in r to $\mathsf{ObfAES}_k()$, but it must obtain an encoding of r from the endpoints that works with ObfAES_k. This encoding is obtained using a protocol called oblivious transfer [26, 6] which provides the guarantee that the endpoints will not learn r. We call this technique *obfuscated rule encryption*.

However, an active attacker at MB might try to run ObfAES_k on every possible word so as to decrypt the traffic

via a dictionary attack. To prevent this, we augment each rule r from the rule generator RG with a signature by RG. Now, we can ensure that MB will obtain an encoding from the endpoints of *only* those rules r for which it has a signature from RG. This step can be done by incorporating the signature verification into the garbled circuit (although we have a more efficient way of achieving the same result).

3.2 Building blocks

Before we present the protocol, let us briefly present the functionality and API of our crypto building blocks.

Yao garbling scheme [41, 24]. A garbled circuit scheme, first introduced by Yao, consists of three algorithms Garble and Eval. Garble takes as input a function F with n bits of input and outputs a garbled function ObfF and n pairs of labels $(L_1^0, L_1^1), \ldots, (L_n^0, L_n^1)$, one pair for every input bit of F.

Let us explain the functionality of ObfF. Consider any input x of n bits with x_i being its i-th bit. ObfF has the property that $\mathsf{ObfF}(L_1^{x_1}, \ldots, L_n^{x_n}) = F(x)$. Basically, ObfF produces the same output as F if given the labels corresponding to each bit of x.

Regarding security, ObfF and $L_1^{x_1}, \ldots, L_n^{x_n}$ do not leak anything about F and x beyond $F(x)$. However, the ObfF circuit can be used *only once*. That is, an adversary who has ObfF and $L_1^{x_1}, \ldots, L_n^{x_n}$ is not allowed to receive labels for a different input than x, because he can learn information otherwise.

1-out-of-2 oblivious transfer (OT) [26, 6]. Consider that a party A has two values, L^0 and L^1, and party B has a bit b. Consider that B wants to obtain the b-th label from A, L^b, but B does not want to tell b to A. Also, A does not want B to learn the other label L^{1-b}. Hence, B cannot send b to A and A cannot send both labels to B. Oblivious transfer (OT) enables exactly this: B can obtain L^b without learning L^{1-b} and A does not learn b.

One can see how OT helps with garbled circuits: it enables one party B holding an input x to obtain the labels for x from party A without telling A what x is.

Others. Let H be a hash function (e.g., SHA-1), which is modeled as a random oracle, AES our block cipher, and sig is a digital signature scheme.

3.3 Protocol

We are now ready to present the protocol. Recall that S and R are the sender and receiver, MB the middlebox and RG the rule generator.

Rule preparation. To prepare the rules, MB uses garbled circuits for AES to encrypt the rules as in Fig. 2. One endpoint could be malicious and attempt to perform garbling incorrectly to eschew detection. To prevent such an attack, both endpoints have to prepare the garbled circuit and send it to MB to check that they produced the same result. If the garbled circuits and labels match, MB is assured that they

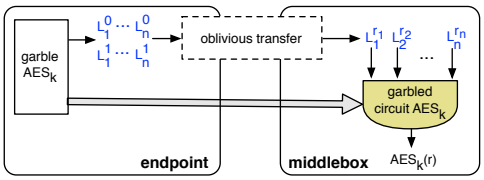


Figure 2: Rule preparation.

are correct because at least one endpoint is honest (as discussed in Sec. 2.2). The only issue is that the garbling process requires randomness. If the endpoints use different randomness, their garbled circuits will not match even if they are equal. Hence, the two endpoints generate the same randomness using a pseudorandom generator seeded with k_{rand} (which is discussed in Sec.2.3).

DPIEnc Rule preparation:

1: MB tells S and R the number of rules N it has.

2: For each rule $1, \ldots, N$, do:

2.1: S and R: Garble the following function F.

F on input $[x, \mathsf{sig}(x)]$ checks if $\mathsf{sig}(x)$ is a valid signature on x using RG's public key and if so, it encrypts x with AES_k and outputs $\mathsf{AES}_k(x)$.

In the garbling process, use randomness based on k_{rand}. Send the resulting garbled circuit and labels to MB.

2.2: MB: Verify that the garbled circuits from S and R are the same, and let ObfAES_k be this garbled circuit. Let r be the current rule. Run oblivious transfer with each of S and R to obtain the labels for r. Verify that the labels from S and R are the same, and denote them $L_1^{r_1}, \ldots, L_n^{r_n}$.

2.3: MB: Evaluate ObfAES_k on the labels $L_1^{r_1}, \ldots, L_n^{r_n}$ to obtain $\mathsf{AES}_k(r)$.

Importantly, Steps 2 and 2c for each rule above are run in parallel.

We use an optimization that, instead of garbling sig verification, it garbles a hash computation while achieving the same security level. Due to space constraints, we do not describe it here.

Tokenize. For the purpose of this section alone, consider a naïve tokenization, in which a token is created for every offset in the packet stream for every string length from 1 bytes to 100 bytes. For example, if the packet stream is "alice apple", if the minimum length is 4 and the maximum is 5, the tokens are "alic", "alice", "lice", "lice ", "ice ", "ice a", etc.

Since this yields a large number of tokens, in Sec. 7, we describe a more clever tokenization algorithm that creates

fewer tokens.

Encrypt. To encrypt the tokens, we use AES and H as discussed above. Additionally, we reduce the size of the encrypted token by taking the last 5 bytes, so that we can have a smaller packet size. This makes the result un-decryptable, but this is not a problem because we do not need to decrypt the tokens. A token of size 5 bytes ensures that collisions between a suspicious string and a different string happen very infrequently. Let $\mathsf{RS} = 2^{40}$.

DPIEnc Encrypt:
The inputs are a set of tokens $t_1, \ldots, t_m$ corresponding to a packet.

1: For each token t_i, MB chooses a random value salt_i, and computes $H\left(\mathsf{salt}, \mathsf{AES}_k(t_i)\right) \bmod \mathsf{RS}$.
2: The output is the list of encrypted tokens:

$$[\ \mathsf{salt}_1, H\left(\mathsf{salt}_1, \mathsf{AES}_k(t_1)\right) \bmod \mathsf{RS}, \ldots, \mathsf{salt}_n, H\left(\mathsf{salt}_n, \mathsf{AES}_k(t_n)\right) \bmod \mathsf{RS}].$$

Detect. Checking for a match is easy with DPIEnc:

DPIEnc Match:
The input is a rule encryption $\mathsf{AES}_k(r)$ and the ciphertext $[\mathsf{salt}, H(\mathsf{salt}, \mathsf{AES}_k(t)) \bmod \mathsf{RS}]$ of a token t.

1: Compute $H(\mathsf{salt}, \mathsf{AES}_k(r)) \bmod \mathsf{RS}$ and check it is equal to the input $H(\mathsf{salt}, \mathsf{AES}_k(t)) \bmod \mathsf{RS}$.

The detection protocol at MB works with many rules and many encrypted tokens. It is important that the detection is as fast as possible because it is one of the main factors in the throughput at the middlebox. Hence, MB must invoke DPIEnc Match algorithm as few times as possible, which turns out to be challenging due to a tension with security. To resolve this tension, we provide a fast detection protocol in Sec. 5, called BlindBox Detect.

Validate tokens. This procedure takes the decrypted traffic from SSL and applies Tokenize and Encrypt as above. It checks that the result is the same as the encrypted tokens from MB. If not, there is a chance that the other endpoint is malicious and flags the misbehavior.

Decrypt. This procedure is the same as in SSL.

3.4 Security guarantee

Our protocols are provably secure; we present security proofs in the Appendix to this document.

The security of Yao garbled circuits and of OT ensures that MB learns $\mathsf{AES}_k(r)$ only for those rules r that come from RG; moreover, MB does not learn anything else about the key k in the process.

BlindBox does not hide the number of tokens in a packet. Also, for every malicious content that matches the traffic stream, MB learns the offset where it matches the stream.

DPIEnc hides data content from MB and it provides a strong security guarantee. We formalize this guarantee using an indistinguishability-type security definition: given tokens t_0 and t_1 for which MB does not have an encrypted rule, and given a ciphertext $c = [\mathsf{salt}, H(\mathsf{salt}, \mathsf{AES}_k(t_b)) \bmod \mathsf{RS}]$ for some bit b, MB cannot guess what b is with chance better than half. In other words, MB cannot tell if t_0 or t_1 is encrypted in c. We can see why this property holds: if MB does not have $\mathsf{AES}_k(t_b)$, this value is indistinguishable from a random value by the pseudorandom permutation property of AES.

Moreover, since our encryption scheme is randomized, MB cannot tell if two tokens are equal to each other (as long as they do not match a malicious string). We can see why this is true: given $[\mathsf{salt}_1, H(\mathsf{salt}_1, \mathsf{AES}_k(t)) \bmod \mathsf{RS}]$ and $[\mathsf{salt}_2, H(\mathsf{salt}_2, \mathsf{AES}_k(t)) \bmod \mathsf{RS}]$, MB cannot tell these are encryptions of the same token t because it cannot invert the hash H.

Note that BlindBox maintains the authenticity property of SSL, but it necessarily breaks the end-to-end security of SSL so as to allow detection.

4 Protocol II: Limited IDS

This protocol supports a limited form of an IDS. Namely, it allows a rule to contain

- multiple keyword matches, and
- absolute and relative offset information within the packet.

This protocol supports most of the keywords in the rule language of Snort [37]. A few keywords are not supported, the most notable being *pcre*, which allows arbitrary regular expressions to be run over the payload.

For example, consider rule number 2003296 from the Snort Emerging Threats ruleset:

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

This rule gets triggered if the flow is from the server, it contains the string "Server|3a| nginx/0." at an offset of at least 17 but no more than 19, and it also contains the strings "Content-Type|3a| text/html" and "|3a|80|3b|255.255.255.255". The symbol "|" denotes binary data.

Protocol II builds on Protocol I and supports this rule as follows. Each content gets encrypted separately with AES as in Protocol I. Each token is now accompanied by its offset in the stream to allow for offset and depth keywords. For example, assume that the stream consists of the simple string "zero-day attack", and assume that the only tokens formed are: "zero", "zero-", "zero-day", "zero-day attack", "-day",

"day", "day attack", "attack". (The tokenization algorithm is explained in Sec. 7.) The packet becomes:

[0, 4, Enc("zero-"), Enc("zero"),
Enc("zero-day attack"), Enc("zero-day");
4, 1, Enc("-day");
5, 2, Enc("day attack"), Enc("day");
9, 1, Enc("attack")],

where the numbers represent offset and number of tokens for that offset respectively, and Enc(t) = [salt, $H(\mathsf{salt}, \mathsf{AES}_k(t))$ mod RS] as in DPIEnc Encrypt. The order of the tokens for each offset is random – upon a match of a token to a content in a rule, this prevents the middlebox from learning the index of the token among the tokens at that offset.

4.1 Security guarantee

The security guarantee is the same as in Protocol I: for each suspicious content of each rule, the middlebox learns if the content appears in the traffic and at what offset. The only difference is that the middlebox now also sees the number of tokens per offset. Nevertheless, the middlebox does not learn the content of the encrypted tokens that do not match suspicious strings, and it also does not learn how long each token is.

5 BlindBox Detect

Before presenting protocol III, we explain how to perform the detection efficiently.

Performing the match of the encrypted rules against the encrypted traffic in a fast way is crucial for the throughput at the middlebox. To come up with a fast detection algorithm, it turns out we need to resolve a tension between performance and security.

Security versus performance tension. Recall that an encrypted token for token t is of the form [salt, c= $H(\mathsf{salt}, \mathsf{AES}_k(t))$ mod RS], the encrypted rule for a rule r is $\mathsf{AES}_k(r)$, and checking for a match involves computing $H(\mathsf{salt}, \mathsf{AES}_k(r))$ and checking if it equals to c. To obtain high performance at the middlebox, MB should perform as few hashes as possible.

Let R be the number of contents in all rules and T the number of tokens in a packet. For example, in ABC Co. there are 3000 rules, $R \approx 9000$ contents, and $T \approx 214$ tokens per packet. A naïve algorithm invokes DPIEnc Match $R \times T$, once for each pair of rule content and token, which is very slow.

An ideal number of matches would be $O(T \log R)$, thus removing the large multiplicative factor of R. In fact, this cost would be easy to achieve if we used one *fixed salt for all the tokens* in a stream, as follows. MB could precompute $H(\mathsf{salt}, \mathsf{AES}_k(r))$ mod RS for every r and arrange these values in a search tree. For each encrypted token c, one checks in $O(\log R)$ time if it exists in the tree.

However, having one fixed salt for all tokens is not secure: the encryption becomes deterministic which means that an attacker at MB can see which token in the traffic equals which other token in the traffic. The role of the salt was to randomize the encryption and thus hide what tokens are equal to each other. Even though the content of the tokens remains hidden, one can build a frequency histogram based on the equality patterns. Since tokens can be both full words and subwords of various lengths, such frequency analysis can leak a lot about the data. Hence, this exposes a tension between security and performance.

Solution. To resolve this tension, the idea is to use a small number of salts while preventing such frequency analysis. First, note that if $t_1 \neq t_2$, one can use the same salt in the encryption of t_1 and t_2 (for a cryptographer reader, the reason is that H is modeled as a random oracle). The problem only appears when two tokens, both equal to t, are encrypted with the same salt.

To keep the number of different salts small, the sender will use the same salt for all distinct tokens, and increment the salt for every token repetition. Concretely, the sender keeps a table mapping the tokens encrypted so far to how many times each one of them appeared in the stream – this table is restarted every P packets so that it remains modest in size. In our experiments, $P = 10$MB gave a modest table size while maintaining good MB detection performance. The sender sends one freshly-chosen random salt and MB records it. Then, for all the following P packets, the sender does not send any other salt. When encrypting a token t, the sender checks the number of times it was encrypted so far, say ct, which could be zero. It then encrypts this token with the salt (salt + ct) by computing $H(\mathsf{salt} + \mathsf{ct}, \mathsf{AES}_k(t))$. Note that this satisfies security because no two equal tokens will have the same salt.

This strategy enables the middlebox to construct a fast search tree over the rules. For each rule, MB hashes it along with S salts, salt, salt + 1, …, salt + S − 1, and adds it in a fast search tree. S is chosen to be an upper bound on the number of times a content from a rule matches a token in the P packets. Note that each content match indicates potentially malicious/suspicious behavior so S is not large. Based on our experiments, $S = 10$ suffices.

Nevertheless, even if some rule has a number of matches more than S, the correctness of detection is not affected: is that MB can tell when the match was to an encrypted rule whose salt was salt + S − 1. In this infrequent case, MB can compute another S hashes for that rule for salt + S, …, salt + 2S − 1 and insert them in the tree, *etc.*.

Note that another advantage of this scheme is that there is only one salt sent for the whole stream as opposed to a salt per token, which decreases bandwidth consumption.

Now, for each packet consisting of T tokens, the cost of the matching is $T \cdot \log(S \cdot R)$ which is much smaller than $T \cdot R$ by three orders of magnitude on the ABC Co. ruleset. Every P packets, the middlebox has to rehash the rules and recompute the tree which yields an additional amortized cost per packet of $O(RS/P \cdot \log RS)$.

6 Protocol III: Full IDS with probable cause privacy

This section enables full IDS functionality, including regexp and scripts, based on our probable cause privacy model. If a content of a rule (a suspicious string) matches a stream of traffic, MB should be able to decrypt the traffic. This enables the middlebox to then run regexp (e.g., the "pcre" field in Snort) or scripts from Bro on the decrypted data. However, if such a suspicious content does not match the packet stream, the middlebox cannot possibly decrypt the traffic, and the security guarantee is the same as in Protocol II.

Protocol insight. The approach is to somehow embed the SSL key k_{SSL} into the encrypted tokens, such that, if the server has a token for t, it can obtain k_{SSL}. Concretely, if the server has $\mathsf{AES}_k(t)$ and the token t shows up in the encrypted traffic, the middlebox should be able to obtain k_{SSL}. To achieve this goal, we replace the encrypted token $\mathsf{salt}, H(\mathsf{salt}, \mathsf{AES}_k(t)) \bmod \mathsf{RS}$ with itself XOR-ed with k_{SSL}: $\mathsf{salt}, H(\mathsf{salt}, \mathsf{AES}_k(t)) \bmod \mathsf{RS} \oplus k_{\mathsf{SSL}}$. If the server has $\mathsf{AES}_k(t)$, the server can construct the encrypted token using the corresponding salt, and hence XOR out k_{SSL}. The problem is that this would slow down detection because it would not allow a simple lookup of an encrypted token into the rule tree described in Sec. 5 – the reason is that the encrypted token is now combined with the SSL key.

Protocol. To maintain the efficiency of the detection, we retain the same encrypted token as in DPIEnc and use it for detection, but additionally create an encrypted token that has the key plugged in. Now, the encryption of a token t becomes: $[\mathsf{salt}, c_1 = H(\mathsf{salt}, \mathsf{AES}_k(t)) \bmod \mathsf{RS}, c_2 = H_2(\mathsf{salt}, \mathsf{AES}_k(t)) \oplus k_{\mathsf{SSL}}]$, where H_2 is a different and independent hash function from H (also modeled as a random oracle). Note that it is crucial that H_2 be different from H because otherwise an attacker can compute $c_1 \oplus c_2$ and obtain k_{SSL}.

MB uses c_1 to perform the detection as before. If MB detects a match using BlindBox Detect, MB computes $H_2(\mathsf{salt}, \mathsf{AES}_k(t))$ using $\mathsf{AES}_k(t)$ (which it has since it found a match), and then $H_2(\mathsf{salt}, \mathsf{AES}_k(t)) \oplus c_2$ which yields k_{SSL}.

7 Optimizations

A more efficient tokenization. The number of tokens affects the bandwidth overhead. In Sec. 3, we proposed a naïve tokenization protocol forming tokens at every offset in a packet for every possible length. This results in a large number of tokens. Fortunately, we can reduce the number of tokens significantly with the following observations.

Observation 1. Images and videos should not be tokenized because intrusion rules such as Snort Community, Snort Emerging Threats and ABC Co. do not look for attacks over this content. Nevertheless, the metadata of this content (e.g., any titles or text associated) is still tokenized.

Observation 2. Rules that are very long can always be detected by concatenating a sequence of shorter tokens. Therefore, at a given offset, we need not generate multiple tokens of varying lengths: we generate one token of a fixed length. For example, to detect the word "constantinople", one can instead detect the overlapping tokens "constant" and "ntinople." We would also transmit "onstanti", "nstantin", "stantino", etc. We refer to this tokenization method as "window-based" tokenization, as we generate tokens using a sliding window of fixed length; we run using a fixed length of 8.

Observation 3. The strings matched in rules start and end before or after a delimiter. Delimiters are punctuation, spacing, and special symbols. For example, if we have the payload "login.php?user=alice", possible content strings are typically "login", "login.php", "?user=", "user=alice" – but not substrings like "logi" and other combinations. Hence, we only need to generate tokens such that we can detect content strings that start and end on delimiter-based offsets; this allows us to ignore redundant tokens in the window. For example, returning to our "constantinople" example, we might transmit " constantinople: ". Here, we would simply not transmite "onstanti", and "nstantin", because they are redundant to a middlebox looking for words that start and end only with spaces or other non-alphanumeric characters. We refer to this tokenization as "delimiter-based" tokenization.

8 System Implementation

We prototyped BlindBox's two components: a client/server library for transmission, and a Click-based [22] middlebox to perform detection.

BlindBox library. The BlindBox HTTPS protocol is implemented in a simple C library that allows applications to connect, transmit, and receive from other BlindBox protocol users. When a client opens a connection, our protocol actually opens three separate sockets: two to the server, and one listener in case a middlebox is on path and needs to request garbled circuits. The two sockets to the server consist of a normal SSL channel (on top of a modified GnuTLS [4] library which allows us to extract the session key under Protocol III), and a secondary channel for 'searchable' encrypted content strings. On send, a user tokenizes the data for transmission, and *first* sends the encrypted content strings, and then sends the traffic over normal HTTPS. At the receiver side, the client receives both the encrypted content strings and the normal HTTPS data. The receiver verifies that tokenization was performed properly, and, if so, returns the decrypted data from the HTTPS connection. If the HTTPS session receives data for which there are no corresponding tokens, it can signal to the sender that there was packet loss, and the tokens should be re-transmitted; until the receiver is sure that all tokens have been transmitted (and observed by the middlebox), it will not pass the decrypted HTTPS data up to the application layer.

Should there be a middlebox on path, the client receives a SYN request from the middlebox on the port + 1 that the BlindBox connection was opened on. The client then immediately begins generating garbled circuits representing an

AES implementation with the client's key hard-coded; to generate the circuits we use JustGarble [8] in combination with the OT Extension library [1]. As we show in the following section, this circuit exchange process is by far the largest cost of using the BlindBox library.

The middlebox. We implemented detection in a Click-based [22] middlebox with two elements: a 'GarbleSpeaker' and a 'BlindBoxFilter'. When a new SYN packet arrives at the BlindBoxFilter element, the connection signature is placed in a queue and then forwarded along; the Filter then marks the connection signature as 'Pending' in a shared SignatureTable structure. The GarbleSpeaker reads in new connections from the queue from the BlindBox Filter and then initiates the handshake with both clients. Once the GarbleSpeaker has evaluated all circuits received from the clients, it constructs the search tree and leaves the encrypted tokens in the shared SignatureTable structure. Finally, it updates the connection from "pending" to "allowed".

While a connection is marked as pending, the BlindBoxFilter allows the first few bytes – the TCP and SSL handshakes – to proceed as normal, but does *not* permit any further data packets through until the connection is marked as Allowed. During this time the client waits to transmit, so this check in practice is never enforced. Once the connection is marked allowed, the BlindBoxFilter allows data packets in the SSL channel to proceed, so long as no attack has been deetcted. Searchable packets are compared against a search tree containing the tokens generated from the garbled circuits. On receipt of a salt update packet from the client, the BlindBoxFilter inserts new tokens in to the tree at the signature; this stalls processing for a few milliseconds and hence it is important that salt updates be rare. If a client sends too many salt updates this is interpreted as a DoS attempt and the connection is stopped.

On signature detection, under Protocol I the connection status is marked as 'DENIED' and all future packets are dropped. Under Protocol II, a signature vector (containing potentially multiple tokens) is updated to mark that an additional token has been detected. If all tokens in the signature are now marked, the connection status is updated to DENIED. Finally, under Protocol III, the connection is passed to a decryption element (which wraps the open source ssldump [3] tool). Like SSL-Termination devices [9] (which today man-in-the-middle all traffic) the traffic can then be passed on to one or several DPI and monitoring appliances, all of which can now operate on plaintext data.

9 Evaluation

To evaluate BlindBox, we answer two questions: First, how completely can BlindBox support our target applications – exfiltration, parental filtering, and HTTP intrusion detection? Second, what are the performance overheads of BlindBox at both the client and middlebox?

9.1 Functionality evaluation

We targeted several use-cases in our design an implementation of BlindBox: Parental Filtering, Document Watermarking, and HTTP Intrusion Detection. We now evaluate how comprehensively BlindBox can support each application.

Can BlindBox implement the functionality required for each target system? Table 2 shows what fraction of 'rules' BlindBox can implement using either protocol I, II, or III for each of several systems. We report on several public datasets, as well as two industrial [2] datasets to which we had (partial) access for the purposes of this evaluation.

Document Watermarking and Parental Filtering can be completely supported using protocol I, as each system relies only on detection of a single substring to trigger an alarm or filtering. However, protocol I can only support between 1.6-5% of the policies required by the more general HTTP IDS applications (the two public Snort datasets, as well as the datasets from XYZ Co. and ABC Co.). This limitation is due to the fact that most IDS policies require exact match detection of multiple substrings, or regular expressions and scripting.

Protocol II, by supporting multiple exact match substrings, extends support to 29-67% of policies for the HTTP IDS applications. Protocol III supports all applications, including regular expressions and scripting, by enabling decryption, only when there is probable cause to do so.

Does BlindBox fail to detect any attacks/policy violations that these standard implementations would detect? In §7, we described two tokenization techniques: Window-based and Delimiter-based tokenization. Window-based tokenization can detect every match in any ruleset (excluding regular expressions). Delimiter-based tokenization relies on the assumption that, in IDSes, most rules occur on the boundary of non-alphanumeric characters, and thus does not transmit all possible tokens – only those required to detect rules which occur between such 'delimiters.' To test this hypothesis, we ran BlindBox over the ICTF2010 [39] trace using the Snort Emerging Threats ruleset excluding all rules with regular expressions. The ICTF trace is a network trace during a college 'capture the flag' contest during which students attempt to hack different servers to win the competition. With BlindBox and delimiter based-detection, we detected 97.1% of all content strings that would have been detected using Snort, and 99% of all attacks that Snort detected.

9.2 Performance & cryptographic overheads

We implemented both the client software required to perform DPIEnc, and a middlebox to perform BlindBox detect, as described in §8. We now use these systems to investigate BlindBox's performance overheads at both the client and the network.

For all experiments, the client software uses Protocol II, which has higher overhead than Protocol I. The primary over-

[2] We thank ABC Co. and XYZ Co. for the data required for this analysis.

Dataset	I.	II.	III.
Document Watermarking [35]	100%	100%	100%
Parental Filtering [5]	100%	100%	100%
Snort Community (HTTP)	3%	67%	100%
Snort Emerging Threats (HTTP)	1.6%	42%	100%
XYZ Co. IDS	5%	40%	100%
ABC Co.	0	29.1%	100%

Table 2: Fraction of attack signatures in public and industrial signature sets addressable with protocols I, II, and III.

head of Protocol III comes from the secondary middlebox to perform regular expression processing. Our prototype of the client software runs on two servers with 2.60 GHz processors connected by a 10GbE link. The machines are multicore, but we used only one thread per client. The CPU supports AES-NI instructions and thus the encryption times for both SSL and BlindBox reflect this hardware support. Since typical clients are not running in the same rack over a 10GbE links, in some experiments we reduced throughput to 20Mbps (typical of a broadband home link) and increased latency to 10ms RTT. Our prototype middlebox runs with four 2.6GHz Xeon E5-2650 cores and 128 GB RAM; the network hardware is a single 10GbE Intel 82599 compatible network card.

Because BlindBox is the only system we know of to enable DPI over encrypted data, our performance baselines are not apples-to-apples comparisons. We measure BlindBox's overhead relative to two protocols: standard SSL, which does not enable DPI as BlindBox does, and a standard functional encryption scheme which provides the correct functionality but which is not tailored to packet processing. The comparison to this strawman demonstrates how far our new techniques in BlindBox towards practicality.

We now describe the functional encryption algorithm we compare against, before discussing bandwidth and computational overheads at both the clients and middlebox.

9.2.1 *Strawman*

If one wants to use existing cryptography, functional encryption is the most applicable encryption scheme to our setting as follows. Since the middlebox (MB) needs to compute on the encrypted traffic, the candidate encryption schemes are fully homomorphic encryption (FHE) and functional encryption. FHE does not permit MB to learn the result of the detection, namely if there is an attack or not, so MB cannot take actions upon an attack, such as drop a packet or alert an administrator. Functional encryption does allow MB to learn whether there is an attack in the packet and at the same time, it prevents MB from learning the content of the traffic. Fortunately, we have constructions of functional encryption which support any function [12, 15] and, hence, they support DPI. However, these are prohibitively impractical: they are slower than FHE which is currently at least 9 orders of magnitude slower than regular computation. For example, [15] nests FHE in itself, resulting in an overhead of at least 18 orders of magnitude.

The only functional encryption schemes that are feasible are specialized functional encryption schemes. We implemented one of the simplest and fastest such scheme [20], which enables only inner product computation on encrypted data (which can also be used for equality checks).

We note that even if functional encryption were efficient, it still does not meet all the security requirements of our setting, although it comes closer than other schemes. The main drawback is that tokens over all streams (for every pair of endpoints) are encrypted with the same key, unlike SSL and BlindBox which has a per-key connection. This means that, if someone gets hold of this key, they can decrypt the traffic between every single pair of endpoints. Also, if someone compromises the rule generator, they can decrypt all traffic sent in the past or future; whereas, in BlindBox, past traffic is not affected and there is a limited effect on future traffic. It is not clear how can one enable a per-connection key as in SSL and BlindBox. One would most likely need to use BlindBox's technique of converting the rule using garbled circuits. However, the rule encryption algorithm is more complicated than BlindBox's: it is based on modular exponentiations which result in very large garbled circuits. Since these are very complicated circuits, we did not implement this rule conversation. Nevertheless, we can estimate a generous lower bound on this phase based on the sizes of modular exponentiation circuits reported in [8]: the garbled circuit will be at least $1.8 \cdot 10^3$ times larger and hence the setup phase will be at least $1.8 \cdot 10^3$ times slower than BlindBox's.

9.2.2 *Client performance*

BlindBox introduces noticeable overheads in client performance stemming from three root causes: the bandwidth overhead due to transmitting encrypted tokens, the computational overhead of generating and encrypting these tokens, and the setup time required to do the handshake with the middlebox and exchange garbled circuits. By far, the largest overhead comes from the initial handshake, which takes on the order of minutes to complete – typically about 414s for a 3000 signature ruleset in our experiments. Post-handshake, however, the overheads are much more modest. A page download during a persistent connection in our experiments increased by between 13% and 300% depending on the page size and contents, resulting in a slower but practical and usable download time.

How long are page downloads with BlindBox, excluding the handshake setup cost? Figure 3 shows page download times using our "typical end user" testbed with 20Mbps links; we show in this figure five popular websites: YouTube, AirBnB, CNN, The New York Times, and Project Gutenberg. The data shown represents the post-handshake (persistent connection) page download time. YouTube and AirBnB load video, and hence have a large amount of binary data which

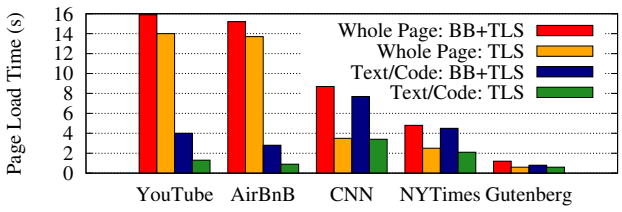


Figure 3: Download time for TLS and TLS+BlindBox at 20Mbps×10ms.

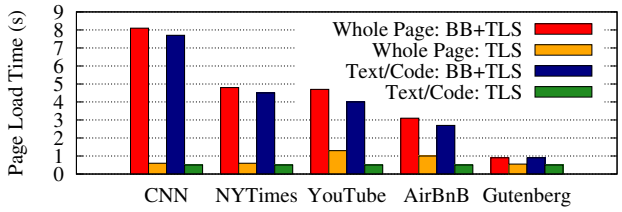


Figure 4: Download time time for TLS and TLS+BlindBox at 1Gbps×10ms.

is not tokenizer. CNN and The New York Times have a mixture of data, and Project Gutenberg is almost entirely text. We show results for both the amount of time to download the page including all video and image content, as well as the amount of time to load only the Text/Code of the page. The overheads when downloading the whole page are at most 2×; for pages with large amount of binary data like YouTube and AirBnB the overhead was only 10-13%. Load times for Text/Code only – which are required to actually begin rendering the page for the user – are impacted more strongly, with penalties as high as 3× and a worst case of about 2×.

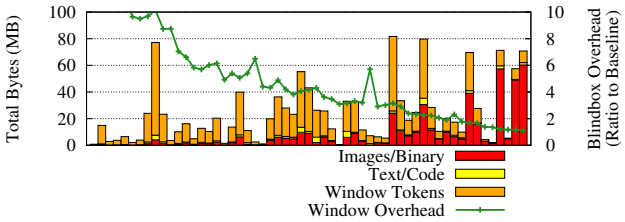
What is the computational overhead of BlindBox encryption, and how does this overhead impact page load times? While the encryption costs are not noticeable in the page download times observed over the 'typical client' network configuration, we immediately see the cost of encryption overhead when the available link capacity increases 1Gbps in Figure 4 – at this point, we see a performance overhead of as much as 16× relative to the baseline SSL download time. For both runs (Figs. 3 and 4), we observed that the CPU was almost continuously fully utilized to transfer data during data transmission. At 20Mbps, the encryption cost is not noticeable as the CPU can continue producing data at around the link rate; at 1Gbps transmission with BlindBox stalls relative to SSL, as the BlindBox sender cannot encrypt fast enough to keep up with the line rate. This overhead can be mitigated with extra cores; while we ran with only one core per connection, tokenization can easily be parallelized.

We provide microbenchmarks of the encryption cost in Table 3, with comparisons to both the Functional Strawman and to SSL. We see that, as observed with CPU utilization, encryption of a 1500 byte packet with BlindBox takes 30× longer than with SSL, but is still 5 orders of magnitude faster than the strawman scheme.

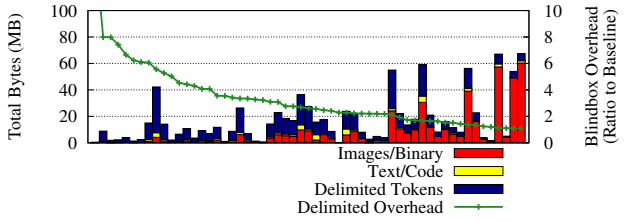
What is the bandwidth overhead of transmitting encrypted tokens for a typical web page? Minimizing bandwidth overhead is key to client performance: less data transmitted means

		Vanilla HTTPS	Strawman HTTPS	BlindBox HTTPS
Client	*Encrypt* (128 bits)	13ns	$70 \cdot 10^6$ ns	69ns
	Encrypt (1500 bytes)	3μs	$15 \cdot 10^6 \mu$s	90μs
	Setup (1 Content)	73ms	$8.3 \cdot 10^4$ms	119 ms
	Setup (3K Rules)	73ms	$7.5 \cdot 10^5$ms	414 s
MB	*Detection:*			
	1 Rule, 1 Token	NP	$1.7 \cdot 10^5 \mu$s	20ns
	1 Rule, 1 Packet	NP	$3.6 \cdot 10^7 \mu$s	5μs
	3K Rules, 1 Token	NP	$5.0 \cdot 10^8 \mu$s	137ns
	3K Rules, 1 Packet	NP	$1.1 \cdot 10^{11} \mu$s	33μs

Table 3: Connection and detection microbenchmarks in comparison between Vanilla HTTPS, our functional strawman, and BlindBox HTTPS. NP Stands for not possible.



(a) Window-Based Tokenization



(b) Delimiter-Based Tokenization

Figure 5: Bandwidth overhead over top-50 web dataset.

less cost, faster transfer times, and lower encryption overhead. The number of encrypted tokens varies widely depending on three parameters of the page being loaded: what fraction of bytes are text/code which must be tokenized, how "dense" the text/code in number of delimiters, and whether or not the web server and client support compression.

Figures 5 (a) and (b) break down transmitted data in to the number of text-bytes, binary-bytes, and tokenize-bytes using the Window-based and Delimiter-based tokenization algorithms (as discussed in §**??**); the y2 axis shows the overhead of adding tokens over transmitting just the original page data. The median page with delimited tokens sees a 2.5× increase in the number of bytes transmitted. In the best case, some pages see only a 1.1× increase, and the worst page sees a 14× overhead. The median page with window tokens sees a 4× increase in the number of bytes transmitted; the worst page sees a 24× overhead. The first observable factor in what impacts this overhead – seen in these figures – is simply what fraction of bytes in the original page load required tokenization – pages which were mostly video suffered lower penalties than pages with large amounts of text, HTML, and javascript as we do not tokenize video that our DPI services have no aim to scan.

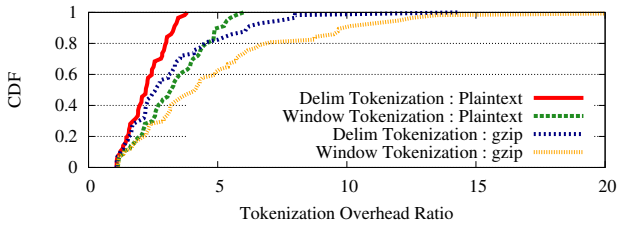
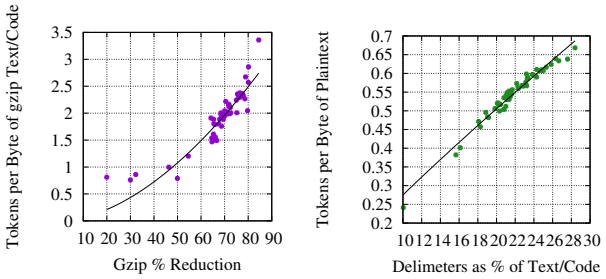


Figure 6: Ratio: transmitted bytes with BlindBox to transmitted bytes with SSL.



(a) Gzip (b) Delimiter Density

Figure 7: Impact of compression and delimiter density on tokenization overhead for delimiter-based tokenization only.

A second factor, better observed in Figures 6 and 7(a) is whether or not the web server hosting the page supports gzip compression. Many web servers will compress content before sending it to clients, which then unzip the data before passing to rendering in the browser. Where window based tokenization imposes a penalty of one token (five bytes) per plaintext byte (and delimiter-based tokenization imposes less than half of a token – 2.2 bytes – by eliminating tokens which are redundante to the DPI engine), compressing the plaintext makes the perceived penalty higher: the baseline data can be compressed, but encrypted tokens cannot. In Figure 6 we show a CDF of the ratio of BlindBox bytes to SSL bytes when gzip is not enabled, and when gzip is enabled exactly as in the original trace (*i.e.* we compare against the bytes gzipped when we downloaded the dataset from the webservers; if any data was not compressed we left it as-is and did not try to compress it further). When compared against plaintext, both window and delimiter based tokenization have 'tight' tails – the worst page with window based tokenization has slightly more than 5× overhead, and the worst page with delimeter tokenization has around 4× overhead. But, for pages which benefit strongly from compression, the penalty can begin to look dramatic at the tail, going as high as 24× for one page (Craigslist.com, which is mostly text/code and benefits strongly from compression). Figure 7(a) shows for each page the number of tokens produced on average per byte, plotted against the page reduction achieved by the web server by using gzip.

The final factor is simply the number of delimiters seen in a page – text-only pages like Project Gutenberg do well in this metric, since there are few code-like characters in the text. The worst performers in this area are pages which make large use of compressed javascript code, where a large fraction of characters result in tokenization. Figure 7(b) illustrates this effect for the same dataset as previously.

9.2.3 Middlebox throughput and performance

We compared the throughput of detection at the middlebox in BlindBox to the throughput in Snort [2]. The table below summarizes our results. We can see that, from this standpoint, BlindBox is twice faster than Snort, a deployed and common IDS system today.

	Snort	BlindBox
Line Rate for Established Connections	85Mbps	166Mbps

This is a result of the fact that BlindBox reduces all detection to exact matching, pushing all regular expression parsing to a secondary middlebox, invoked rarely. Hence, it is unsurprising that BlindBox performs detection more quickly.

While we did not implement a version of BlindBox which relied on our strawman, we can compare against it using a smaller benchmark, which illustrates that the strawman would be prohibitively impractical to use at the middlebox: detection over a single packet against a 3000 rule signature set would take more than a day.

10 Related work

Related work falls into three categories: insecure proposals some being deployed today, relevant work on computing on encrypted data, and other related work.

10.1 Insecure proposals

Given that deployed systems could not support both encryption and deep packet inspection (DPI), they chose the insecure path. Existing systems mount a man-in-the-middle attack on SSL [18, 16] by installing fake certificates at the middlebox [21, 33]. This enables the middlebox to break the security of SSL and decrypt the traffic. Once the traffic is decrypted, middleboxes can run DPI protocols. However, this breaks the end-to-end security of SSL, and as a consequence, many issues follow as surveyed in Jarmoc [18].

Some proposals allow users to tunnel their traffic to a third party middlebox provider, *e.g.* Meddle [32], Beyond the Radio [38], and APLOMB [34]. These approaches allow the middlebox owner to inspect/read all traffic; although the situation is preferable (from the client's perspective) in that the inspector is one with whom the client has a formal/contractual relationship. However, this approach is *not* preferable the network service providers, who may wish to *enforce* policy on users in the network, using a local middlebox to ensure, *e.g.*, that no hosts within the network are infected with botnet malware.

As compared to these works, BlindBox maintains the benefits of encryption.

10.2 Computing on encrypted data

Fully homomorphic encryption (FHE) [13] and general functional encryption [12, 15] are encryption schemes that can compute any function over encrypted data and thus promise to support the complexity of deep packet inspection tasks. They do not address all the desired security properties in our threat model, and more importantly, they are prohibitively slow, currently at least 9 orders of magnitude slower than unencrypted computation [14].

Even a specialized functional encryption scheme [20] as our strawman described in Sec. 9.2.1 is six orders of magnitude slower than BlindBox. Also, some recent systems [29, 30] showed that some specialized computation can be performed efficiently on encrypted data. However, these systems perform SQL operations or certain types of search on the encrypted data, and do not enable the computations we are interested in for deep packet inspection.

There is a large body of work on searchable encryption, which enables finding keywords on encrypted text. A part of our protocol has the flavor of searchable encryption because it seeks to match the content of a rule on encrypted traffic. However, existing searchable encryption schemes do not provide the desired security or functionality in BlindBox. Symmetric-key searchable encryption schemes [36, 19] require the endpoint to encrypt the rule with the symmetric key which means that the endpoint sees the rule, thus violating our threat model. Public-key searchable encryption schemes [10] do not have this drawback, but they are still less secure than BlindBox and remain too slow for this setting. First, the security they provide is weaker because all tokens for all connections (over all pairs of endpoints) are encrypted with the same key unlike a different key per flow as in SSL – when this key gets compromised, everyone's traffic can be decrypted. Second, their performance is unacceptable because they perform a slow operation, called a cryptographic pairing, for every pair of token to rule content – in our experimental setup, for the ABC Co. ruleset and using the pbc library [25] for bilinear maps, inspecting one packet against the ruleset at the middlebox takes $\geq 9 \cdot 10^8 \mu$s, which is orders of magnitude more than BlindBox and regular SSL. Moreover, none of these schemes provide a story for supporting arbitrary regexp and scripts on the traffic, unlike our probable cause privacy model and protocol.

10.3 Other related work

Yamada et al. [40] show how one can detect some limited intrusion attacks by using only the data size and timing of SSL-encrypted packets. Their approach cannot do detection based on the content of a packet and thus cannot support IDSes, exfiltration detection or parental filtering.

11 Conclusion and future directions

In this paper, we argue that the vision of computing on encrypted packets promises to solve the tension between security and functionality at middleboxes.

To the best of our knowledge, BlindBox is the first system that makes inspecting encrypted packets usable on human timescales. It improves performance by six orders of magnitude as compared to what existing techniques could do today in this setting, and, for frequent operations such as detection, it is faster than Snort, a common IDS deployed today. As discussed, current research holds the potential to bring BlindBox's overhead further down and this is the subject of our future work.

References

[1] OT Extension library. https://github.com/encryptogroup/OTExtension.

[2] Snort. https://www.snort.org/.

[3] ssldump. http://www.rtfm.com/ssldump/.

[4] The GnuTLS Transport Layer Security Library. http://www.gnutls.org/.

[5] University of Toulouse Internet Blacklists. http://dsi.ut-capitole.fr/blacklists/.

[6] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS*, 2013.

[7] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs.

[8] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy*, 2013.

[9] BlueCoat. SSL Encrypted Traffic Visibility and Management. https://www.bluecoat.com/products/ssl-encrypted-traffic-visibility\-and-management.

[10] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of the 23rd Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, Interlaken, Switzerland, May 2004.

[11] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges.

[12] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Proceedings of the 54th Annual Symposium on Foundations of Computer Science (FOCS)*, 2013.

[13] C. Gentry. Fully homomorphic encryption using ideal lattices. In *41st*, pages 169–178, Bethesda, MD, May–June 2009.

[14] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. Cryptology ePrint Archive, Report 2012/099, June 2012.

[15] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable

garbled circuits and succinct functional encryption. In *ACM Symposium on Theory of Computing (STOC)*, 2013.

[16] L.-S. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing forged ssl certificates in the wild. In *IEEE Symposium on Security and Privacy*, 2014.

[17] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.

[18] J. Jarmoc. Ssl/tls interception proxies and transitive trust, 2012.

[19] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[20] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products.

[21] A. Kingsley-Hughes. Gogo in-flight Wi-Fi serving spoofed SSL certificates. *ZDNet*, 2015.

[22] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router.

[23] B. Kreuter, A. Shelat, and C. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium*, 2012.

[24] Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptol.*, 22:161–188, April 2009.

[25] B. Lynn. PBC library: The pairing-based cryptography library. http://crypto.stanford.edu/pbc/.

[26] M. Naor and B. Pinkas. Oblivious transfer with adaptive queries. In *CRYPTO*, 1999.

[27] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafo, K. Papagiannaki, and P. Steenkiste. The cost of the "s" in https. 2014.

[28] V. Paxson. Bro: A system for detecting network intruders in real-time. https://www.bro.org/, 1999.

[29] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, Cascais, Portugal, Oct. 2011.

[30] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using Mylar. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, Seattle, WA, Apr. 2014.

[31] T. C. Projects. Spdy: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.

[32] A. Rao, J. Sherry, A. Legout, W. Dabbout, A. Krishnamurthy, and D. Choffnes. Meddle: Middleboxes for increased transparency and control of mobile traffic. In *CoNEXT Student Workshop*, 2012.

[33] Runa. Security vulnerability found in cyberoam dpi devices (cve-2012-3372). *Tor project's blog*.

[34] 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. ACM SIGCOMM*, 2012.

[35] G. J. Silowash, T. Lewellen, J. W. Burns, and D. L. Costa. Detecting and preventing data exfiltration through encrypted web sessions via traffic inspection. Technical Report CMU/SEI-2013-TN-012.

[36] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 44–55, Oakland, CA, May 2000.

[37] The Snort Project. Snort users manual, 2014. Version 2.9.7.

[38] N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, N. Weaver, and V. Paxson. Beyond the radio: Illuminating the higher layers of mobile networks. Technical Report TR-14-003, ICSI, 2014.

[39] G. Vigna. ICTF Data. https://ictf.cs.ucsb.edu/#/.

[40] A. Yamada, Y. Saitama Miyake, K. Takemori, A. Studer, and A. Perrig. Intrusion detection for encrypted web accesses. In *21st International Conference on Advanced Information Networking and Applications Workshops*, 2007.

[41] A. C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (FOCS)*, 1986.

[42] K. Zetter. The feds cut a deal with in-flight Wi-Fi providers, and privacy groups are worried. *Wired*, 2014.

APPENDIX

In this appendix, we treat cryptographically the algorithms in BlindBox: the DPIEnc encryption scheme, BlindBox Detect, and the probable cause privacy protocol.

We first introduce notation and provide the syntax for our scheme, then provide a definition of what it means to be secure, and then prove that DPIEnc achieves this definition under standard cryptographic assumptions. Next, we define security formally for probable cause privacy and prove that our construction achieves it.

A Notation

Let κ denote the security parameter throughout this paper. For a distribution $\mathcal{D}$, we say $x \leftarrow \mathcal{D}$ when x is sampled from the distribution $\mathcal{D}$. If S is a finite set, by $x \leftarrow S$ we mean x is sampled from the uniform distribution over the set S. We use $p(\cdot)$ to denote that p is a function that takes one input. Similarly, $p(\cdot,\cdot)$ denotes a function p that takes two inputs.

We say that a function f is negligible in an input parameter κ, if for all $d > 0$, there exists K such that for all $\kappa > K$, $f(\kappa) < \kappa^{-d}$. For brevity, we write: for all sufficiently large κ, $f(\kappa) = \mathrm{negl}(\kappa)$. We say that a function f is polynomial in an input parameter κ, if there exists a polynomial p such that for all κ, $f(\kappa) \leq p(\kappa)$. We write $f(\kappa) = \mathrm{poly}(\kappa)$.

Let $[n]$ denote the set $\{1, \ldots, n\}$ for $n \in \mathbb{N}^*$. When saying that a Turing machine A is p.p.t. we mean that A is a non-uniform probabilistic polynomial-time machine.

Two ensembles, $X = \{X_\kappa\}_{\kappa\in\mathbb{N}}$ and $Y = \{Y_\kappa\}_{\kappa\in\mathbb{N}}$, are said to be *computationally indistinguishable* (and denoted $\{X_\kappa\}_{\kappa\in\mathbb{N}} \overset{c}{\approx} \{Y_\kappa\}_{\kappa\in\mathbb{N}}$) if for every probabilistic polynomial-time algorithm D,

$$|\Pr[D(X_\kappa, 1^\kappa) = 1] - \Pr[D(Y_\kappa, 1^\kappa) = 1]| = \mathrm{negl}(\kappa).$$

In our security definitions, we will define probabilistic experiments and denote by random variables their outputs. For example, $\mathsf{Exp}_{\mathsf{Adv}}(1^\kappa)$ denotes the random variable representing the output of the experiment with adversary Adv on security parameter κ. Moreover, $\{\mathsf{Exp}_{\mathsf{Adv}}(1^\kappa)\}_{\kappa\in\mathbb{N}}$ denotes the ensemble of such random variables indexed by $\kappa \in \mathbb{N}$.

B Syntax

We now define the syntax for the class of encryption schemes we call middlebox searchable encryption scheme, or shortly MBSE. DPIEnc is such an encryption scheme.

Definition 1 (Syntax)**.** *An MBSE scheme associated with message space $\mathcal{M}$ is is a tuple of p.p.t. algorithms* (Setup, Enc, $\mathsf{RuleEnc}$, Match) *as follows:*

- $\mathsf{Setup}(1^\kappa)$: *Takes as input a security parameter 1^κ and outputs a key k.*
- $\mathsf{Enc}(k, t_1, \ldots, t_n)$: *Takes as input the key k and a set of n tokens each in $\mathcal{M}$ where $n = \mathrm{poly}(\kappa)$, and outputs a salt* salt*, and a set of ciphertexts $[c_1, \ldots, c_n]$.*
- $\mathsf{RuleEnc}(k, r)$: *Takes as input a key k and a rule string $r \in \mathcal{M}$, and outputs an encrypted rule* encr*.*
- $\mathsf{Match}(\mathsf{encr}, \mathsf{salt}, c_1, \ldots, c_n)$*: Takes as input an encrypted rule* encr *corresponding to a rule r, a salt* salt *and a set of ciphertexts and outputs the set of indexes $\{\mathsf{ind}_1, \ldots, \mathsf{ind}_\ell\}$, where each index is in $[n]$.*

Correctness. *For any polynomial $n(\cdot)$, for every sufficiently large security parameter κ, if $n = n(\kappa)$, for all $[t_1, \ldots, t_n] \in \mathcal{M}^n$, for every rule $r \in \mathcal{M}$, for every index i such that $t_i = r$ and for every index j such that $t_j \neq r$, we have:*

$$\Pr\left[\begin{array}{l} k \leftarrow \mathsf{Setup}(1^\kappa); \\ \mathsf{salt}, c_1, \ldots, c_n \leftarrow \mathsf{Enc}(k, t_1, \ldots, t_n); \\ \mathsf{encr} \leftarrow \mathsf{RuleEnc}(k, r); \\ S \leftarrow \mathsf{Match}(\mathsf{encr}, \mathsf{salt}, c_1, \ldots, c_n) : \\ i \in S \end{array}\right] = 1.$$

and

$$\Pr\left[\begin{array}{l} k \leftarrow \mathsf{Setup}(1^\kappa); \\ \mathsf{salt}, c_1, \ldots, c_n \leftarrow \mathsf{Enc}(k, t_1, \ldots, t_n); \\ \mathsf{encr} \leftarrow \mathsf{RuleEnc}(k, r); \\ S \leftarrow \mathsf{Match}(\mathsf{encr}, \mathsf{salt}, c_1, \ldots, c_n) : \\ j \in S \end{array}\right] = \mathrm{negl}(\kappa).$$

The correctness property above specifies that a match is detected with probability 1 for every token that matches the rule, but for a token that does not match the rule, the probability of a match is negligibly small.

C Security definition

Our security definition is standard for searchable encryption schemes. For example, it is similar to the security definition of Song et al. [36]. This is helpful because it means that the security guarantee we provide is well-studied as opposed to new and not well understood.

This security definition is indistinguishability-based: at a high level, given two sets of tokens, and an encryption of one of these two sets of tokens, no polynomial-time adversary can tell with chance significantly better than half, which of the two sets of tokens were encrypted. In other words, the adversary gains no side information from the encryption scheme.

However, when given an encrypted search word (called rule string in our case), the attacker can tell precisely which encrypted token this rule matches, while not learn anything else about the data. This statement is typically formalized by allowing the attacker to choose any two sets of tokens of the same length, any number of rules, and as long as those sets of tokens match the set of rules at the same tokens, no attacker can distinguish between encryptions of the two sets of tokens. Note that the property in the paragraph above follows from this property (in the case that an adversary chooses an empty set of rules).

Our security definition does not specify that rule strings are also hidden, which means that the attacker is allowed to learn the rules. (As a side note, the encryption scheme does hide the rules, providing a deterministic encryption guarantee, but in our systems setup, the middlebox will get to know the rules from the rule generator anyways, so formalizing this property is not useful).

Definition 2 (MBSE security)**.** *Consider an MBSE scheme with algorithms* (Setup, Enc, RuleEnc, Match) *and associated message space $\mathcal{M}$. Let* Adv *be a p.p.t. stateful adversary with oracle access to H. Consider the following experiment.*

$\underline{\mathsf{Exp}_{\mathsf{Adv}}(1^\kappa)}$**:**

1: $k \leftarrow \mathsf{Setup}(1^\kappa)$
2: $T^0 = (t_1^0, \ldots, t_n^0), T^1 = (t_1^1, \ldots t_n^1) \leftarrow \mathsf{Adv}(1^\kappa)$
3: $b \leftarrow \{0,1\}$*, a random bit.*
4: $\mathsf{salt}, c_1, \ldots, c_n \leftarrow \mathsf{Enc}(k, t_1^b, \ldots, t_n^b)$
5: $r_1, \ldots, r_\ell \leftarrow \mathsf{Adv}(\mathsf{salt}, c_1, \ldots, c_n)$
6: $\mathsf{encr}_1, \ldots, \mathsf{encr}_\ell \leftarrow \mathsf{RuleEnc}(k, r_1), \ldots \mathsf{RuleEnc}(k, r_\ell)$
7: $b' \leftarrow \mathsf{Adv}(\mathsf{encr}_1, \ldots, \mathsf{encr}_\ell)$
8: *Let I_i^0 be the set of indexes that match r_i in T^0 and I_i^1 be the set of indexes that match r_i in T^1. If $b' = b$ and $I_i^0 = I_i^1$ for all i, output "Success" else output "Fail".*

We say that the scheme is secure if for all p.p.t. stateful adversaries Adv*, and for all sufficiently large κ:*

$$\Pr[\mathsf{Exp}_{\mathsf{Adv}}(1^\kappa) = \textit{"Success"}] \leq 1/2 + \mathrm{negl}(\kappa).$$

In this security definition, the adversary Adv chooses two sets of tokens T^0 and T^1, receives an encryption of one of these at random (the bit b controls which set of tokens will be encrypted) and then tries to guess b by outputting b'. Adv is also allowed to choose the rules. As with typical searchable encryption security definitions, the adversary succeeds if his guess b' equals b and only if he chose rules that do not automatically distinguish T^0 and T^1: these rules must match T^0 and T^1 at the same indexes; otherwise, the functionality we desire from the scheme will enable anyone to distinguish these two sets of tokens trivially. We want to ensure that the attacker dos not learn anything from the scheme other than the pattern of matching, hence, he only succeeds if he chooses sets of tokens with the same pattern of matchings with the rules.

D Construction

For preciseness, we provide the construction of the scheme here too. This construction consists of DPIEnc enhanced with the encryption from BlindBox Detect. Here, we are concerned only with security. Hence, we do not include in the construction the data structure BlindBox Detect builds at the middlebox which is only meant for performance and has no implication on security. However, we do include the enhanced encryption due to BlindBox Detect which chooses salts in specific ways, because we do need to prove that this mechanism does not weaken security.

Also, in this treatment, we do not include the rule encryption step using Yao garbled circuits: the security of the overall scheme with the Yao garbled circuits follows trivially from composing the security of this scheme with the security guarantees of Yao garbled circuits.

Let H be a hash function modeled as a random oracle.

The setup algorithm $\mathsf{Setup}(1^\kappa)$**:** Generate AES key k as in AES.

The encryption algorithm $\mathsf{Enc}(k, t_1, \ldots, t_n)$**:**

1: Let salt be a random salt as in AES.
2: For each $i \in [n]$, do:

2.1: let ct be the number of times that t_i repeats in the sequence $t_1, \ldots, t_{i-1}$. ct could be 0.

2.2: Compute $c_i = H(\mathsf{salt} + \mathsf{ct}, \mathsf{AES}_k(t_i)) \bmod RS$.

3: Output $\mathsf{salt}, c_1, \ldots, c_n$.

Note that the strategy above for generating salts is from BlindBox Detect.

The rule encryption algorithm $\mathsf{RuleEnc}(k, r)$**:** Output $\mathsf{encr} = \mathsf{AES}_k(r)$.

We do not include a second description of the matching algorithm here. The reason is that it has no bearing on security: it does not show up in the security definition (Def. 2). This makes sense because matching is performed on data that is *already* available to the attacker from other algorithms such as Enc and RuleEnc. It provides no new information to the attacker (and in fact it is ran by the attacker). The matching algorithm is involved only in the correctness of functionality of our scheme, which we already explained in the body of the paper.

E Security proof

The security of our scheme relies on the standard cryptographic assumption that AES is pseudorandom permutation [**?**], and on H being a random oracle.

Theorem 1. *Assuming that* AES *is a pseudorandom permutation and H is a random oracle, our construction in Sec. D is a secure MBSE scheme.*

Proof. It is easy to check why this construction satisfies the syntax of MBSE as described in Sec. B.

We now prove security. We prove security through a sequence of two hybrids. The first hybrid replaces the AES encryption of tokens with deterministic random values, based on the pseudorandom security property of AES. The second hybrid then replaces the random oracle with deterministic random values based on the property of the random oracle. This results in an experiment in which the distribution of encryptions of T^0 and T^1 are statistically equal and thus indistinguishable, proving our theorem.

Hybrid 1. The Enc algorithm is changed to replace $\mathsf{AES}_k(\cdot)$ with random values. Concretely:
$\mathsf{Hybrid1.Enc}(k, t_1, \ldots, t_n)$**:**

1: Let salt be a random salt as in AES.

2: **For each** $i \in [n]$**, generate a random value** R_i **in the ciphertext space of** AES_k**, with the only restriction that it preserves equality. Namely, iff** $t_i = t_j$**,** $R_i = R_j$**.**

2.1: let ct be the number of times that t_i repeats in the sequence $t_1, \ldots, t_{i-1}$. ct could be 0.

2.2: **Compute** $c_i = H(\mathsf{salt} + \mathsf{ct}, R_i) \bmod RS$**.**

3: Output $\mathsf{salt}, c_1, \ldots, c_n$.

The rows in bold indicate differences from the regular encryption.

We also define $\mathsf{Hybrid1.RuleEnc}(k, r)$ to output R_i if $r = t_i$ for some t_i, otherwise to output a fresh random value R. A future rule $r' = r$ should also be assigned the same random value as r.

One can define $\mathsf{Exp}_{\mathsf{Adv},\mathsf{Hybrid\ 1}}(1^\kappa)$ in the same way as $\mathsf{Exp}_{\mathsf{Adv}}(1^\kappa)$ by replacing Enc with Hybrid1.Enc and RuleEnc with Hybrid1.RuleEnc.

Lemma 2. *Assuming AES is a pseudorandom permutation, for all p.p.t. stateful adversaries* Adv*, for all sufficiently large κ:*

$$\Pr[\mathsf{Exp}_{\mathsf{Adv}}(1^\kappa) = \textit{“Success”}] \leq \Pr[\mathsf{Exp}_{\mathsf{Adv},\textit{Hybrid 1}}(1^\kappa) = \textit{“Success”}] + \mathrm{negl}(\kappa).$$

Proof. The proof follows directly from the pseudorandom property of AES, which means that AES_k is computationally indistinguishable from a random oracle. □

Hybrid 2. The Enc algorithm is changed to replace H with random values.
$\mathsf{Hybrid2.Enc}(k, t_1, \ldots, t_n)$**:**

1: Let salt be a random salt as in AES.

2: **For each** $i \in [n]$**, generate a random value** R_i **in the ciphertext space of** $[0, RS - 1]$ **bits and let** $c_i = R_i$**.**

3: Output $\mathsf{salt}, c_1, \ldots, c_n$.

The rows in bold indicate differences from Hybrid 1. Unlike Hybrid 1, there is no restriction on the random values R_i any more.

We also define $\mathsf{Hybrid2.RuleEnc}(k, r)$ to output a random value R for each rule r to be encrypted, with the only restriction that any rule string $r' = r$ is also assigned to R.

In the case of Hybrid 2, we also need to program the random oracle H. When Adv gives as input to H salt^*, $\mathsf{Hybrid2.RuleEnc}(k, r)$, for r such that $r = t_i$ for some i and $\mathsf{salt}^* = \mathsf{salt} + \mathsf{ct}_i$, H returns R_i.

One can define $\mathsf{Exp}_{\mathsf{Adv},\text{Hybrid 2}}(1^\kappa)$ in the same way as $\mathsf{Exp}_{\mathsf{Adv}}(1^\kappa)$ by replacing Enc with Hybrid2.Enc and RuleEnc with Hybrid2.RuleEnc.

Lemma 3. *Assuming H is a programmable random oracle, for all p.p.t. stateful adversaries* Adv*, for all sufficiently large κ:*

$$\Pr[\mathsf{Exp}_{\mathsf{Adv},\textit{Hybrid 1}}(1^\kappa) = \textit{"Success"}] \leq \Pr[\mathsf{Exp}_{\mathsf{Adv},\textit{Hybrid 2}}(1^\kappa) = \textit{"Success"}] + \mathrm{negl}(\kappa).$$

Proof. The proof follows directly from the properties of the random oracle. □

Lemma 4. *For all p.p.t. stateful adversaries* Adv*,*

$$\Pr[\mathsf{Exp}_{\mathsf{Adv},\textit{Hybrid 2}}(1^\kappa) = \textit{"Success"}] = 1/2.$$

Proof. We can see that Hybrid 2 loses all the information about $t_1, \ldots, t_n$ and the rules r except for the pattern of matching between the rules and the tokens $t_1, \ldots, t_n$. In other words, all encryptions are random values preserving the pattern of matching between rules and tokens. This pattern is the same for T^0 and T^1. Hence, the two distributions for T^0 and T^1 are statistically the same, which means that any adversary Adv has a chance of distinguishing them of exactly half. □

By Lemmas 2, 3, and 4, we obtain:

$$\Pr[\mathsf{Exp}_{\mathsf{Adv}}(1^\kappa) = \text{“Success”}] \leq \Pr[\mathsf{Exp}_{\mathsf{Adv},\text{Hybrid 2}}(1^\kappa) = \text{“Success”}] + \mathrm{negl}(\kappa) = 1/2 + \mathrm{negl}(\kappa),$$

which concludes our proof.

□

F Probable cause security

The security of the probable cause algorithm specifies that the middlebox should obtain k_{SSL} if and only if there is a match of a rule string against a token. This security guarantee is similar to the one of attributed-based encryption schemes [11] (shortly denoted ABE). Hence, we formalize the security of probable cause in a similar way to ABE security.

In this security definition, Adv is allowed to choose two SSL keys k^0_{SSL} and k^1_{SSL}, one of these gets selected at random and then encrypted along with the tokens. The challenge to the adversary Adv is to guess which one of these keys were encrypted, when no rule string matches any of the contents.

We call such an encryption scheme, probable cause MBSE. The syntax of a probable cause MBSE scheme is the same as the syntax of MBSE (Sec. B) except for the encryption and match algorithms. The encryption algorithm takes an additional element k_{SSL}: $\mathsf{Enc}(k, t_1, \ldots, t_n, k_{\mathsf{SSL}})$. The matching algorithm additionally produces k_{SSL} upon a match.

Definition 3 (Probable cause security)**.** *Consider a probable cause MBSE scheme with algorithms* (Setup, Enc, RuleEnc, Match) *and associated message space $\mathcal{M}$. Let* Adv *be a p.p.t. stateful adversary with oracle access to H. Consider the following experiment.*

$\underline{\mathsf{Exp}_{\mathsf{Adv}}(1^\kappa)}$**:**

1: $k \leftarrow \mathsf{Setup}(1^\kappa)$
2: $k^0_{\mathsf{SSL}}, k^1_{\mathsf{SSL}} \leftarrow \mathsf{Adv}(1^\kappa)$
3: $T = t_1 \ldots t_n \leftarrow \mathsf{Adv}(1^\kappa)$
4: $b \leftarrow \{0, 1\}$*, a random bit*
5: $\mathsf{salt}, c_1, \ldots, c_n \leftarrow \mathsf{Enc}(k, t_1, \ldots, t_n, k^b_{\mathsf{SSL}})$
6: $r_1 \ldots r_\ell \leftarrow \mathsf{Adv}(\mathsf{salt}, c_1, \ldots, c_n)$
7: $\mathsf{encr}_1, \ldots, \mathsf{encr}_\ell \leftarrow \mathsf{RuleEnc}(k, r_1), \ldots \mathsf{RuleEnc}(k, r_\ell)$
8: $b' \leftarrow \mathsf{Adv}(\mathsf{encr}_1, \ldots, \mathsf{encr}_\ell)$
9: *If $b' = b$ and none of the rules r_j match any token in T, output "Success" else output "Fail".*

We say that the scheme has probable cause security if for all p.p.t. stateful adversaries Adv*, and for all sufficiently large κ:*

$$\Pr[\mathsf{Exp}_{\mathsf{Adv}}(1^\kappa) = \textit{"Success"}] \leq 1/2 + \mathrm{negl}(\kappa).$$

F.1 Probable cause construction

Let us recall the probable cause construction. As before, the matching algorithm has no bearing to security so we do not review it here. The setup and rule encryption algorithms are the same as in the regular construction.

Let H and H_2 be hash functions modeled as random oracles.

The setup algorithm $\mathsf{Setup}(1^\kappa)$**:** Generate AES key k as in AES.

The encryption algorithm $\mathsf{Enc}(k, t_1, \ldots, t_n, k_{\mathsf{SSL}})$**:**

1: Let salt be a random salt as in AES.
2: For each $i \in [n]$, do:
  2.1: let ct be the number of times that t_i repeats in the sequence $t_1, \ldots, t_{i-1}$. ct could be 0.
  2.2: Compute $c_i = (H(\mathsf{salt}, \mathsf{AES}_k(t_i)) \bmod \mathsf{RS}, H_2(\mathsf{salt}, \mathsf{AES}_k(t_i)) \oplus k_{\mathsf{SSL}})$.
3: Output $\mathsf{salt}, c_1, \ldots, c_n$.

The rule encryption algorithm $\mathsf{RuleEnc}(k, r)$**:** Output $\mathsf{encr} = \mathsf{AES}_k(r)$.

F.2 Security proof

Theorem 5. *Assuming that AES is a pseudorandom permutation, and H and H_2 are random oracles, the construction above has probable cause security.*

Proof. As before, we prove security through a sequence of hybrids.

Hybrid 1. The Enc algorithm is changed to replace $\mathsf{AES}_k(\cdot)$ with random values. Concretely:

$\mathsf{Hybrid1.Enc}(k, t_1, \ldots, t_n, k_{\mathsf{SSL}})$**:**

1: Let salt be a random salt as in AES.
2: **For each $i \in [n]$, generate a random value R_i in the ciphertext space of AES_k, with the only restriction that it preserves equality. Namely, iff $t_i = t_j$, $R_i = R_j$.**
  2.1: let ct be the number of times that t_i repeats in the sequence $t_1, \ldots, t_{i-1}$. ct could be 0.
  2.2: **Compute $c_i = (H(\mathsf{salt}, R_i) \bmod \mathsf{RS}, H_2(\mathsf{salt}, R_i) \oplus k_{\mathsf{SSL}})$.**
3: Output $\mathsf{salt}, c_1, \ldots, c_n$.

The rows in bold indicate differences from the regular encryption.

We also define $\mathsf{Hybrid1.RuleEnc}(k, r)$ to output R_i if $r = t_i$ for some t_i, otherwise to output a fresh random value R. A future rule $r' = r$ should be assigned the same random value.

Hybrid 2. The Enc algorithm is changed to replace H and H_2 with random values.

$\mathsf{Hybrid2.Enc}(k, t_1, \ldots, t_n, k_{\mathsf{SSL}})$**:**

1: Let salt be a random salt as in AES.
2: **For each $i \in [n]$, generate a random value R_i, R'_i in the ciphertext space of $[0, RS-1]$ bits and let $c_i = (R_i, R'_i \oplus k_{\mathsf{SSL}})$.**
3: Output $\mathsf{salt}, c_1, \ldots, c_n$.

The rows in bold indicate differences from Hybrid 1.

We also define $\mathsf{Hybrid2.RuleEnc}(k, r)$ to output a random value R for each rule r to be encrypted, with the only restriction that any rule string $r' = r$ is also assigned to R.

Hybrid 3. The Enc algorithm is changed to completely lose k_{SSL}. Concretely $R'_i \oplus k_{\mathsf{SSL}}$ becomes R'_i.

$\mathsf{Hybrid2.Enc}(k, t_1, \ldots, t_n, k_{\mathsf{SSL}})$**:**

1: Let salt be a random salt as in AES.
2: **For each $i \in [n]$, generate a random value R_i, R'_i in the ciphertext space of $[0, RS-1]$ bits and let $c_i = (R_i, R'_i)$.**
3: Output $\mathsf{salt}, c_1, \ldots, c_n$.

The rows in bold indicate differences from Hybrid 2.

The rule encryption algorithm does not chance in this hybrid: $\mathsf{Hybrid3.RuleEnc}(k, r) = \mathsf{Hybrid2.RuleEnc}(k, r)$.

By the pseudorandom properties of AES, Hybrid 1 is computationally indistinguishable from the original security game. By the random oracle properties of H and H_2, Hybrid 1 and 2 are computationally indistinguishable. Hybrid 2 and 3 are statistically indistinguishable because $R'_1 \ldots R'_n$ are independent and random values and they preserve the same distribution when xor-ed with k_{SSL}. In Hybrid 3, no adversary Adv can distinguish between k^0_{SSL} and k^1_{SSL} because these keys are not used in the encryption, thus concluding our proof.

□