# MBArk: Securely outsourcing middleboxes to the cloud

Paper **#59**

14 pages

## Abstract

Network middleboxes such as firewalls, NAT, proxies, and intrusion prevention systems are crucial components of modern networks. Recently, more and more organizations are outsourcing their middleboxes to the cloud. However, this poses a problem for the confidentiality of the traffic because now the cloud provider has access to the organization's traffic. We design and build MBArk (read "embark"), the first system that enables running a wide range of middleboxes at a cloud provider while maintaining the confidentiality of traffic. MBArk encrypts the traffic that reaches the cloud and enables the cloud to process the *encrypted* traffic without decrypting it. MBArk supports a wide-range of middleboxes such as firewall, NAT, web proxy, load balancing, IP forwarding, intrusion prevention systems, data exfiltration systems, and VPN. Our evaluation shows that MBArk supports these applications with competitive performance: because MBArk makes no modifications to the dataplane for most middleboxes, throughput at most middleboxes is unaffected by our changes.

## 1. Introduction

Network processing appliances ("middleboxes") such as firewalls, NATs, proxies, and intrusion detection systems are crucial components of modern networks [27]. In recent trends, more and more organizations are *outsourcing* their network processing, either to cloud providers [2, 27, 36] or to Internet service providers who refer to this trend as Network Functions Virtualization (NFV) [11]. This strategy promises to reduce costs, decrease the burden of managing and configuring these devices, and provide redundant resources for elasticity and fault tolerance [27]. Already now, the NFV working group [10] has over 250 members ranging from large telecoms to hardware manufacturers, all of whom are investing in new technologies to enable outsourced traffic processing.

Nevertheless, outsourcing middleboxes to a third party service provider brings a new and important challenge: the confidentiality of the traffic. In order to be able to process and examine the traffic of an organization, the cloud receives the traffic *unencrypted*. This means that the cloud now has access to potentially sensitive packet payloads, IP addresses, and ports revealing confidential information about the organization. This situation is worrisome considering the number of documented data breaches by cloud employees or hackers gaining access to clouds [7]. Hence, an important question is: can we enable a third party to perform traffic processing for an enterprise, *without seeing the enterprise's traffic*?

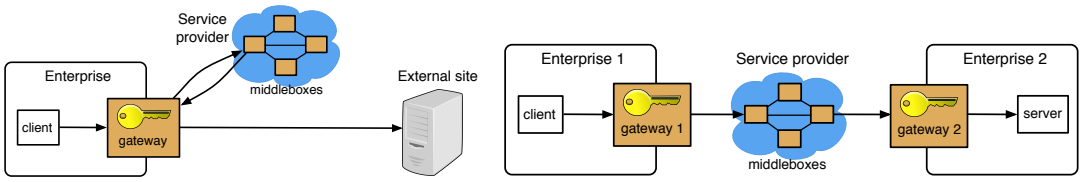
| | Middlebox | Type | Example |
|---|---|---|---|
| 1 | IP Firewall §4.1 | HO | IP address |
| 2 | Application Firewall §4.1 | BA | IP address |
| 3 | NAT §4.2 | HO | IP address |
| 4 | IP Forwarding §6 | HO | IP address |
| 5 | Load Balancer L4 §6 | HO | IP address |
| 6 | Load Balancer L7 §6 | BA | URL in payload |
| 7 | Web Proxy/Cache §4.3 | HO | URL in payload |
| 8 | Intrusion Detect (IDS) §5 | BA | payload |
| 9 | Data Exfiltration §5 | BA | payload |
| 10 | VPN Gateway §6 | HO | IP address |

**Table 1.** Middleboxes supported by MBArk, shown with the section that discusses them, their type, HO (header only) or BA (bytestream aware), and one example of an encrypted field each operates on.

We design and build MBArk, the first system that enables running a wide range of middleboxes at a cloud while maintaining the confidentiality of the traffic. MBArk provides these guarantees even against attackers who gained access to *all* the data at the cloud. MBArk's name concatenates the words MB (middlebox) and Ark (protection). MBArk supports a wide range of middleboxes, in fact, all middleboxes that [27] documents to be fit for cloud outsourcing. We list these middleboxes in Table 1. Moreover, MBArk supports these middleboxes with almost no penalty in typical middlebox throughput despite our changes.

The approach in MBArk is to encrypt the traffic that goes to the middleboxes in the cloud and enable the cloud to process encrypted traffic without ever decrypting it. MBArk encrypts the packet payload as well as important information in the header (such as IP addresses and ports). Since the service provider receives only encrypted traffic and no decryption key, it cannot see the confidential data. However, designing a practical system that supports a wide-range of middlebox applications over encrypted traffic is challenging and required a set of new cryptographic and systems techniques.

The first challenge is that performing generic computation on encrypted data is prohibitively impractical. The applications considered provide complex functionalities. For example, firewall and NAT examine packet header information such as IP addresses and ports; for IDS and data exfiltration detection, the middlebox (MB) examines the packet payload and tries to match complex rules (e.g., keywords, regular expressions). For each packet, a middlebox must pro-




(a) Enterprise to external site communication

(b) Enterprise to enterprise communication

**Figure 1.** System architecture. Aplomb and NFV system setup with MBArk encryption at the gateway. The arrows indicate traffic from the client to the server; the response traffic follows the reverse direction.

cess a packet on the timescale of microseconds. Existing generic homomorphic encryption schemes are many orders of magnitude impractical [13], not just for middleboxes but for most systems (even those with more relaxed performance requirements). CryptDB [25] proposed a practical way of computing on encrypted data, and we follow the same train of thought: instead of using a generic encryption scheme, CryptDB proposes to identify core operations of the system and to support each with a specialized and fast encryption scheme. Unfortunately, neither the encryption schemes nor the systems techniques in the database setting of CryptDB meet the tight performance and the security requirements of our network setting, so we must start from scratch.

We identify two core operations that underlie middlebox processing: *keyword match* and *range match*. Keyword match refers to identifying if a keyword appears in a byte stream. For example, keyword match is useful for a web proxy: the service provider can identify if a filename cached at the proxy appears in a HTTP GET request in the packet flow. Range match enables determining if a value $x$ is in an interval $[x_1, x_2]$, for example, if an IP address is in the range [18.0.0.0, 18.255.255.255] (*i.e.* the prefix 18.0.0.0/8). Note that range match supports a basic version of keyword match, namely complete equality check, when the range consists of only one value. Table 1 summarizes the middleboxes we support and the operations they rely on.

A second challenge is that, although there exist algorithms for keyword match, there is no suitable encryption scheme for the range match scheme: the only practical schemes applicable here perform order-preserving encryption (OPE) [5, 24], and are both too insecure and too slow for our setting. For example, OPE schemes leak the order of the values encrypted. We designed a new encryption scheme, called *RangeMatch*, which is targeted at and takes advantage of the network setting. RangeMatch is fast (performing encryptions in under 3$\mu$s, 3 orders of magnitude faster than the OPE schemes) and it is more secure because it does not reveal the order of the values encrypted.

A last challenge is to design and build a system that supports the variety of applications mentioned, is practical and easy to deploy. To address this challenge, we make the following contributions:

(a) We design a protocol for each type of middlebox in Table 1 by employing our two encrypted match primitives.

(b) We integrate these protocols in a way that *does not change the existing packet structure*. We achieve this by ensuring that our encrypted values fit into the space of the unencrypted values, leverage the IPv6 options header, and sometimes add a second network flow.

(c) For most middleboxes, we keep *unchanged the algorithms employed on the fast path* data plane which processes packets. In particular, firewalls implemented in hardware can use the existing hardware unchanged. This enables middlebox throughput with MBArk to almost match throughput without MBArk, despite our changes.

We implemented and evaluated MBArk on EC2. As mentioned, we support all applications that fit in the cloud outsourcing model, as surveyed in [27] – any appliance which is outsourced today can also be outsourced using MBArk. Further, MBArk imposes negligible throughput overheads at the service provider: for example, a firewall operating over encrypted data achieves 9.8Gbps, equal to the same firewall over unencrypted data. Our gateway can forward at 1.5 Gbps on a single core; our 8 core server can transmit MBArk encrypted data at up to 8Gbps.

## 2. Overview

In this section, we present MBArk's architecture, the threat model and applications supported.

### 2.1 System architecture

MBArk uses the same architecture as APLOMB [27], a system which redirects an enterprise's traffic to the cloud for middlebox processing. MBArk augments this architecture with confidentiality protection. We do not delve into the details, motivation, and gains of APLOMB's setup, and refer the reader to [27] for details. In this setup, there are three parties: enterprise(s), the service provider (SP), and an external site providing some service. The enterprise runs a gateway (GW) which sends traffic to a middlebox (MB) running in the cloud. The service provider runs a set of middleboxes.

We illustrate two redirection setups in Fig. 1. The first setup, in Fig. 1(a), occurs when the enterprise communicates with an external site: traffic goes to the cloud and back before it is sent out to the Internet. It is worth mentioning that APLOMB allows an optimization that saves on bandwidth and latency relative to Fig. 1(a): the traffic from SP can go directly to the external site and does not have to go

back through the gateway. MBArk does not allow this optimization fundamentally: the traffic from SP is encrypted and cannot be understood by an external site. We evaluate in §8 what MBArk loses by not allowing this optimization. Nevertheless, for traffic within the same enterprise, where the key is known by two gateways owned by the same company, we can support this optimization as shown in Fig. 1(b).

### 2.2 Threat model

The goal of MBArk is to protect the privacy of the traffic against an attacker at the service provider (cloud employee, or hacker gaining access to cloud machines). We consider a strong attacker, one that has gained access to *all the data at SP*. This includes any traffic and communication SP receives from the gateway, any logged information, cloud state, and so on. Nevertheless, we assume that SP provides good service and runs middlebox functionality correctly – but we do not want SP to see the traffic in the process of doing so.

We assume that the gateways are trusted; in particular, they do not leak information.

Some middlebox functionalities (such as intrusion or exfiltration detection) have a threat model of their own about the client and the server. For example, intrusion detection assumes that the client or the server could misbehave and mount an intrusion attack, but at most one of them misbehaves [23]. We preserve these threat models unchanged. These applications rely on the middlebox to detect attacks in these threat models. Since we assume the middlebox executes its functions correctly and MBArk preserves the functionality of these middleboxes, these threat models are irrelevant to the protocols in MBArk.

### 2.3 Goals for the gateway

The goals of NFV and APLOMB are to delegate the burden of managing and configuring middleboxes (e.g., upgrading, deciding which vendor to use, monitoring), reduce costs of hardware, and provide elasticity and fault tolerance; hence MBArk should maintain these goals. These translate into two requirements for the gateway, which should be:

**(1)** Implementation agnostic: for each *type* of middlebox, the gateway should implement a generic functionality, and it should not depend on the specific version and vendor of the software running at the middlebox. For example, the gateway should not need to change when a new version of the McAfee firewall is installed, when the admin changes the firewall from McAfee to Juniper, or when the NAT software gets upgraded.

**(2)** Should perform lightweight operations and be parallelizable. If our design were not to meet this goals, it would defeat the purpose of outsourcing in the first place: were the gateway just as complex and costly as the middleboxes, there would be no cost or manageability benefits.

MBArk achieves both these goals, as we explain next.

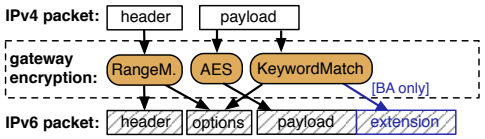



**Figure 2.** Packet encryption at the gateway. Patterned squares indicate encrypted data. Only for BA middleboxes, KeywordMatch produces additional encrypted data.

### 2.4 MBArk overview

MBArk encrypts all the traffic passing through the service provider (SP). MBArk encrypts IP addresses, ports, and the payload of the packet, thus protecting the privacy of all these parameters. As in Fig. 1, the gateway has a secret key $k$; in the setup with two gateways, they share the same secret key. The gateway encrypts all traffic going to the service provider using MBArk's encryption schemes. The middleboxes at SP process *encrypted traffic* using MBArk's protocols. After the processing, the middleboxes will produce encrypted traffic which SP sends back to the gateway. The gateway decrypts the traffic using the key $k$.

Throughout this process, middleboxes at SP handle only encrypted traffic and never get the decryption key. This ensures that an attacker that steals all the data from SP will see only encrypted traffic, which protects the privacy of the traffic.

**Two types of middleboxes.** MBArk enables encrypted operation for *all* typical middleboxes in outsourcing scenarios, despite substantial differences in how the middleboxes operate, what packet data they access, and whether they modify or merely observe packets. We classify middleboxes in two broad categories based on the type of encrypted fields they compute on.

*Header-Only Middleboxes:* operate over packet headers (*e.g.*, IP, TCP, or even HTTP headers) on a packet-by-packet basis. Examples include IP Firewalls and Network Address Translators (NATs).

*Bytestream-Aware Middleboxes:* these middleboxes operate over a TCP bytestream – the concatenation of all *payloads* – and hence must keep copies of the data transmitted for each connection. For example, many Intrusion Detection Systems (IDS) are bytestream-aware; in order to detect attack signatures that span across multiple packets they keep a buffer with copies of each payload it has forwarded.

Web proxies are a special case: even though this middlebox reconstructs the bytestream, the only computation on encrypted data is on the HTTP header, so we classify them as a HO middlebox.

In Table 1, we list the type of each middlebox in MBArk.

**Packet encryption.** The gateway always generates IPv6 packets as output to the cloud, regardless of whether the input packets from the enterprise are IPv4 or IPv6. The tunnel connecting the gateway to the cloud may be either v4 or v6. As we will see in the following section, our RangeMatch scheme requires 128 bits to encode encrypted IP addresses.

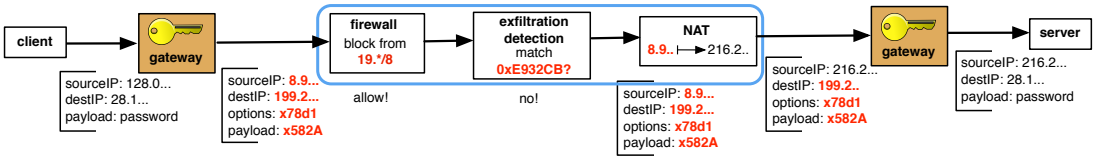



**Figure 3.** Example of packet flow through a few middleboxes. Red in bold indicates encrypted data.

Fig. 2 shows how the gateway encrypts each packet. The gateway encrypts all IP addresses and ports in the header with RangeMatch. Some encrypted values fit in the header in the place of the unencrypted values, and some other values are placed in the IPv6 options header. As we will see, all header-based middleboxes can operate directly on RangeMatch encrypted values as if they were normal IP addresses and port values. Some network appliances are commonly configured to drop packets with IP options; these middleboxes must be configured under MBArk to permit packets with MBArk options to pass through. Next, the gateway encrypts the payload with a standard cipher such as AES. The last step is to encrypt the payload with KeywordMatch, and here we have two cases. The first case is when all the middleboxes at SP are header-only. In this case, the gateway produces one encrypted value based on the payload and places it in the options header. The second case is when there is at least one bytestream-aware middlebox at SP, in which case the gateway produces a set of encrypted values using KeywordMatch. For the purpose of the design, these can be viewed as an extension to the encrypted payload, although in our implementation (§7), they are sent in a second stream.

**Packet flow example.** To understand MBArk, Fig. 3 shows the end-to-end flow of a packet through three example middleboxes in the cloud. First, the gateway encrypts the packet as explained above. The packet passes through the firewall which tries to match the encrypted information from the header against its encrypted rule, and decides to allow the packet. Next, the exfiltration device checks for any suspicious (encrypted) strings on the extension of the packet's payload, and not finding any, it allows the packet to continue to the NAT. The NAT maps the source IP address to a different IP address. Back at the gateway, the gateway decrypts the packet.

On top of MBArk's encryption, the gateway can use SSL to secure the communication to SP. It can use SSL also on the links client-G and G-server.

## 3. Building blocks

In this section we present the building blocks MBArk relies on: keyword match and range match. The first scheme already existed and the second is a new encryption scheme we provide. When describing these schemes, for concreteness, we refer to the encryptor as the gateway whose secret key is $k$ and to the entity computing on the encrypted data as the service provider (SP) because this is how these schemes will be used by MBArk.

### 3.1 Keyword match

KeywordMatch allows detecting if an encrypted keyword matches an encrypted data item by equality. For example, given an encryption of the keyword "malicious", and a list of encrypted strings [Enc("alice"), Enc("malicious"), Enc("alice")], SP can detect that the keyword matches the second string. For this, we use a searchable encryption scheme [28, 29]. Using this scheme, the gateway can encrypt a value $v$ into $\mathsf{Enc}(v)$ and a rule $r$ into enc_r and SP can detect if there is a match between $v$ an $r$. The security of searchable encryption is well studied [28, 29]: at a high level, given a list of encrypted strings, and an encrypted keyword, SP does not learn anything about the encrypted strings, other than which strings match the keyword. The encryption of the strings is *randomized*, so it does not leak whether two encrypted strings are equal to each other, unless, of course, they both match the encrypted keyword. As an optimization, in some applications, we can use a deterministic encryption scheme instead of the scheme above *without sacrificing security*: this is the case when we are guaranteed that the strings encrypted are distinct.

### 3.2 Range match

Many middleboxes perform detection over a *range* of port numbers or IP addresses. The functionality of the RangeMatch scheme is to encrypt a set of ranges $[s_1, e_1]$, ..., $[s_n, e_n]$ into $[\mathsf{Enc}(s_1), \mathsf{Enc}(e_1)]$, ..., $[\mathsf{Enc}(s_n), \mathsf{Enc}(e_n)]$, and a value $v$ into $\mathsf{Enc}(v)$, such that anyone with access to these encryptions can determine in which range $v$ lies, while not learning the value of $v$. MBArk does not aim to hide the values of values of $s_1, e_1, \ldots, s_n, e_n$ because many times these come from SP. For concreteness, we explain our scheme by considering $v$, $e_i$ and $s_i$ as IP addresses (although it can be used for encrypting ports too).

**Requirements.** In order for this encryption scheme to fit MBArk efficiently and securely, it must: (1) *Be fast* and ideally, the scheme should preserve the ability to use *existing fast packet matching algorithms*, such as various kinds of tries, area-based quadtrees, FIS-trees, or hardware-based algorithms [17]. All of these rely on the ability of SP to compute ">" between $v$ and the endpoints of an interval; (2) *Provide strong security*: SP does not learn anything about $v$ other than what interval it matches to. Even if $v_1$ and $v_2$ match the same range, SP should not learn their order. SP may learn the order relation of the intervals (in fact, in many setups, SP provides the intervals); (3) *Be deterministic* to integrate with the NAT; (4) *Be format-preserving*:

For example, an encrypted IP address should look like an IP address. This property is important to avoid changing the packet header structure.

Unfortunately, there is no encryption scheme that supports all these requirements. The closest schemes to these requirements is *order-preserving encryption*, BCLO [5] and mOPE [24]. These schemes are both less secure and less efficient than our scheme. In terms of security, they leak the order between any two IP addresses encrypted, and not just whether they match the same interval or not. Moreover, BCLO [5] leaks the top half of the bits too. In terms of performance, they cannot keep up with packet-processing demands, as they take milliseconds per encryption (as demonstrated in §8), which would only allow the gateway to forward a few hundred packets per second.

Instead, we designed a new encryption scheme that satisfies all the requirements above called *range match* scheme.

**Intuition.** To encrypt the endpoints of the intervals, we sort them, and choose as their encryptions values equally distributed in the domain space in a way that preserves the order of the endpoints. For example, the encryption of the intervals 127.0.0.0/8 and 172.16.0.0/16, is [51.0.0.0, 102.0.0.0] and [153.0.0.0, 204.0.0.0]. This preserves the order of the intervals but does not leak anything else.

For now, consider that the gateway maintains a mapping of each interval endpoint to its encryption, called the *interval map*. The interval map also contains the points $-\infty$ and $+\infty$, encrypted with 0.0.0.0 and 255.255.255.255.

When the gateway needs to encrypt an IP address $v$, the gateway first determines what is the interval $v$ falls in. Then, it chooses a random value in this interval. For example, if $v$ is 127.0.0.1, a possible encryption is 48.124.24.85. This is great for security because the encryption does not retain any information about $v$ other than the range it is in. In particular, for two values $v_1$ and $v_2$ that match the same interval, SP does not learn their order. Thus, this satisfies the security requirement above.

We need to specify how to encrypt $v$ when $v$ fits in multiple intervals or in no interval at all. Consider an example in which $v$ fits in no interval at all. Let $v$ be 127.0.0.0 and the intervals be 18.0.0.0/8 and 172.16.0.0/16 with encryptions [51.0.0.0, 102.0.0.0] and [153.0.0.0, 204.0.0.0]. The encryption of $v$ should not be chosen at random between 102.0.0.0 and 153.0.0.0 because SP learns that $v$ is between the two intervals. Recall that we want SP to learn only whether $v$ matches an interval or not, but nothing else. Hence, $v$ should be mapped to a random value anywhere in the intervals [0, 51.0.0.0), (102.0.0.0, 153.0.0.0) and (204.0.0.0, 255.255.255.255].

To achieve the desired security, the idea is to find the interval $I$ inside which $v$ should be mapped at random. The equation for $I$ is as follows. Consider the intervals *inside* which $v$ falls, and let $I_0, I_1, \ldots, I_{n_I}$ be their encryptions. We always include the total interval $I_0 = [0, 0, 0, 0, 255.255.255.255]$.

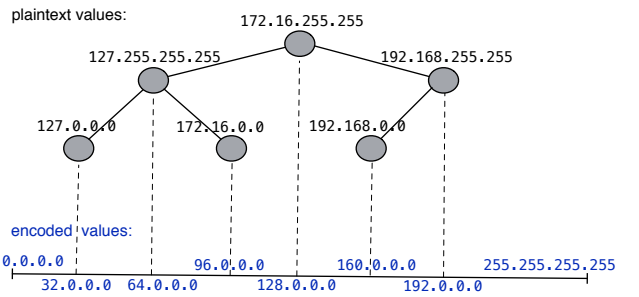



**Figure 4.** Range match tree. The values of nodes in the tree are the unencrypted IP addresses, and the blue values on the horizontal axis are their encryptions.

Now consider the intervals *outside* of which $v$ falls and let $O_1, \ldots, O_{n_O}$ be their encryptions. Then, to provide our desired security goal, $v$ should be chosen at random from the interval $I = I_0 \cap I_1 \cap \ldots \cap I_{n_I} - (O_1 \cup \cdots \cup O_2)$.

To ensure the encryption is deterministic, we use a pseudo-random function [16], $\mathsf{prf}$, seeded in $k$. Let $|I|$ be the size of the interval $I$. Then, the encryption of $v$ is the index-th element in $I$, where index is $\mathsf{index}(v) = \mathsf{prf}_k(v) \bmod |I|$. Note that, in the system's setup with two gateways, the gateways generate the same encryption because they share $k$.

When encrypting IP addresses, we do not want two different IP addresses to map to the same encryption (which would break the NAT). Fortunately, a simple calculation shows that the probability that two IP addresses get assigned to the same encryption is negligibly low for IPv6 (but not quite for IPv4).

The last issue to address is addition and removal of intervals. For example, this can happen when a new rule is added to a firewall. Since we tried to spread out the encryptions of the intervals evenly in the IP space, there is no room for the new interval. We can readjust all the encryptions of these intervals to make space for the new interval. However, this would require the firewall hardware to reconfigure fully which is slow. Ideally, we would only reconfigure the firewall hardware incrementally, for the new interval. For this, we build on the idea from mOPE [24] and store the intervals at the gateway in a balanced scapegoat tree as in Fig. 4. This tree is a search tree that has the property that when inserting or deleting a node, the number of other nodes that change encryption is small, namely, $O(\log n)$ amortized worst case where $n$ is the number of nodes. Each node in the tree is now encrypted similarly to before: the root gets the middle of the IP range, the left node gets the middle of the IP space to the left of the middle, and so on, as in Fig. 4. Note that, since the tree is balanced, it maintains our desire of having the encryptions of endpoints roughly uniformly distributed.

### 3.2.1 Algorithms used by the gateway

**Gateway state.** The gateway keeps a small amount of state (in our implementation, about 200 bytes/range) encryption the intervals, but maintains no state per IP address encrypted or per connection. The gateway stores the tree in Fig. 4: for each node, it stores the unencrypted endpoint, whether it is

the left or right margin of an interval, and the other endpoint of the interval it is part of. It does not need to store the encryption of the endpoint because this is easy to derive from the position in the tree.

The gateway can use the following functions. EncryptRanges encrypts the initial ranges. Note that some ranges could consist of one point only, namely $s = e$.

1: **procedure** ENCRYPTRANGES($[s_1, e_1], \ldots, [s_n, e_n]$)
2: Build scapegoat tree on the values $\{s_1, \ldots, s_n\} \cup \{e_1, \ldots, e_n\} \cup \{-\infty, \infty\}$.
3: Assign an encryption $\mathsf{Enc}(x)$ to each node $x$ in the tree:
- the root gets the middle of the IP range, $e$,
- the node to the left of the root gets the middle of the interval to the left of $e$: $(e/2)$,
- the right node gets the middle of the range to the right of $e$: $(3e/2)$, and so on.

4: **return** $[\mathsf{Enc}(s_1), \mathsf{Enc}(e_1)], \ldots, [\mathsf{Enc}(s_n), \mathsf{Enc}(e_n)]$

EncryptValue encrypts the values to be matched in ranges.

1: **procedure** ENCRYPTVALUE($v$)
2: Search the tree for $v$ to compute efficiently I.
3: Compute $\mathsf{index}(v) = \mathsf{prf}_k(v) \bmod |I|$.
4: Let $\mathsf{Enc}(v)$ to be the index-th element of $I$.
5: **return** $(\mathsf{Enc}(v), \mathsf{IV}, \mathsf{AES}_k(\mathsf{IV}, v))$ for random IV.

Here is how to compute $I$ efficiently. When searching for $v$ in the tree, the gateway can identify the tightest enclosing interval $[p_1, p_2]$ in logarithmic time. If $[p_1, p_2]$ are endpoints of the same interval, then I = $[p_1, p_2]$. Otherwise, move towards the left in the tree until you identify the first endpoint $\ell_1$ that belongs to an interval $[\ell_1, \ell_2]$ enclosing $v$. Then, using the tree, scan $[\ell_1, \ell_2]$ and eliminate any intervals not containing $v$. The gateway can precompute and store this interval for every two consecutive nodes in the tree.

EncryptValue returns an AES encryption of $v$ too, because $\mathsf{Enc}(v)$ is not decryptable.

We now describe the procedure for AddRange, which adds an interval. Deleting an interval is similar and we do not present it. These will modify the state at the gateway. Besides the interval added or deleted, a small number of other intervals may be moved. For these, the algorithm returns the old and new encryption of the interval.

1: **procedure** ADDRANGE($[s, e]$)
2: Insert $s$ and $e$ into the scapegoat tree. If $s = e$, insert the value only once.
3: Initialize $L$ to be the empty list.
4: **if** tree needs to be rebalanced **then**
5: Record which nodes change position in the tree during rebalancing, together with their old and new encryptions. Namely, record

$$L = \{\mathsf{enc}_1 \leftarrow \mathsf{enc}_1^*, \ldots, \mathsf{enc}_m \rightarrow \mathsf{enc}_m^*\},$$

where $m$ is the number of nodes who changed position in the tree, and $\mathsf{enc}_i$ and $\mathsf{enc}_i^*$ are the old and new encryption of the $i$-th node that changed position.
6: Compute $\mathsf{Enc}(s)$ and $\mathsf{Enc}(e)$, the encryptions of $s$ and $e$, as in EncryptRanges.
7: **return** $[\mathsf{Enc}(s), \mathsf{Enc}(e)], L$

Since we are using a scapegoat tree, the number of nodes that change position during rebalancing is amortized worst case $O(\log n)$ where $n$ is the total number of nodes in the scapegoat tree.

A natural question is whether there exists a scheme that does not need to adjust already encrypted intervals. For a related setting, Popa et al. [24] proved that whenever one supports "$>$" over encrypted data, there must be some adjustment similar to this; the proof holds for our setting too.

#### 3.2.2 Algorithms used by the cloud

The cloud can run "$\geq$" and "$\leq$" between any encrypted value $\mathsf{Enc}(v)$ and an encrypted endpoint $\mathsf{Enc}(s)$ and $\mathsf{Enc}(e)$, and will obtain a correct answer. Computing $\mathsf{Enc}(v_1) < \mathsf{Enc}(v_2)$ between two encrypted values in the same range is meaningless, and returns a random value.

#### 3.2.3 Security guarantees

The scheme achieves our desired security goal: the only information leaked about a value $v$ encrypted is which ranges it matches. In particular, the scheme is not order preserving because it does not leak the order of two encrypted values that match the same range. It is easy to check that the scheme is secure: since the encryption of $v$ is random in $I$, the scheme only leaks the fact that $v$ is in $I$. $I$ is chosen in such a way that the only information about $v$ it encodes is which intervals $v$ matches and which it does not match.

## 4. Header-only middleboxes

Most classes and functions within a middlebox codebase can be labeled as one of these three categories: the dataplane (which performs packet processing), the middlebox state (which stores and updates rules and policies, as well as per-flow state, and the control logic (which converts user configurations to rules and policies, and updates/refreshes state). Importantly, packet processing only occurs in dataplane functions, which are usually tightly engineered to process each packet in microseconds or faster.

As we will show, adapting middleboxes to operate over encrypted data requires (a) encrypting the rules given to the middleboxes, and (b) changes to the control logic and some of the processing state. However, the key algorithms that process packets on the dataplane remain unchanged for header-only middleboxes, with the exception of proxies.

This allows the packet-processing throughput to remain fast; as we see in our evaluation §8 header-only middleboxes under MBArk achieve near identical throughput to the same middleboxes over unencrypted data.

### 4.1 Middlebox: Firewall

We use the term "firewall" for stateful and stateless packet filters that filter the traffic based on network layers and transport layers. Our RangeMatch scheme supports both types of firewalls. We now explain the design of the firewall based on this scheme.

**Setup.** Initially, the gateway (G) encrypts the rules to be used by the firewall, by encrypting all IP addresses and ports in the rules, as follows.

First, it prepares the IP and port intervals. Recall that RangeMatch requires the IP addresses to be IPv6 to guarantee the injectivity of encryption (see §3.2); hence, we extend an IPv4 prefix such as 157.161.48.0/24 to an IPv6 one ::ffff:157.161.48.0/120. This problem does not show up for ports because they only need to be distinct within the same IP address. The gateway then expands every prefix into an interval, and every exact match $x$ into $[x, x]$.

Next, the gateway encrypts these intervals using EncryptRanges (§3.2) by running one instance of RangeMatch for IP addresses and one instance for ports. It then converts each encrypted IP range into a set of prefixes and duplicates the rule for each prefix in this set.

Consider an example rule from `pf`, the default firewall under BSD: `block out log quick on $ext_if from 157.161.48.0/24 to any.`

Let the encryption of 157.161.48.0/24 be the interval [80.0.0.0, 160.0.0.0] (for brevity, we use IPv4). This interval is equivalent to the prefixes: 80.0.0.0/4, 96.0.0.0/3, 144.0.0.0/4, and 160.0.0.0/32. Hence, the gateway replaces the original rule with four rules, one for each encrypted prefix. The worst-case number of prefixes for IPv6 is $O(\min(\log \text{number of rules}, 128)) = O(\log \text{number of rules})$, which is small.

The gateway sends the new rules to the service provider (SP) which installs them into the firewall *the same way it would install them if they were not encrypted*.

**Processing traffic.** The gateway sends the packet to the firewall using the format described in §2. The firewall can execute on the encrypted header *the same way as on the unencrypted header* because RangeMatch maintains the order relation between values in rules and in packet headers. In particular, it can use any of the existing fast matching algorithms unchanged. Further, many high-speed firewalls have a hardware datapath and these can be used without change.

**Updating rules.** When an administrator adds a new rule, the gateway computes the new values $[\mathsf{Enc}(s), \mathsf{Enc}(e)]$ as well as a list of prefix changes based on $L$: this list maps old prefix to new prefix so that the firewall knows how to update its rules. The gateway sends the new rules and prefix changes to the firewall, and the firewall reconfigures itself. Due to the guarantees of our RangeMatch protocol, this list contains a small number (logarithmic) of intervals that changed. However, some firewalls keep per-flow state; if a packet's encryption changes due to a new rule, this state will no longer be usable. We discuss how updates impact stateful middleboxes in §7. Such adjustment of encryptions is inconvenient but it is necessary for an encryption scheme like RangeMatch; without it, an impossibility result applies, as discussed in §3.2. Nevertheless, this operation happens rarely and is not expensive: it happens only when firewall rules change and the number of ranges changed is amortized logarithmic in the total number of ranges.

### 4.2 Middlebox: NAT

Instead of mapping unencrypted addresses/ports to public IP addresses/ports, the NAT maps IP addresses/ports that are *encrypted* to public IP addresses/ports. Consider an example: an encrypted private IP address ab12:342:: and encrypted private port 231 are mapped to a public IP address 160.1.1.1 with port 445.

The fast path of the NAT in MBArk is unchanged from the regular NAT. When a packet arrives at the NAT with an encrypted IP address of ab12:342:: and an encrypted port of 231, the NAT maps these to 160.1.1.1 and 445, and does the reverse for reverse traffic. This process proceeds the same as on unencrypted data because encrypted values look the same as unencrypted values. We discuss in §7 what the NAT does when there are changes in the ranges stored at the firewall.

### 4.3 Middlebox: Proxy/cache

The proxy caches HTTP static content (e.g., images) in order to improve client-side performance. When a client opens a new HTTP connection, a typical proxy will capture the client's SYN packet and open a new connection to the client, as if the proxy were the web server. The proxy then opens a second connection in the background to the original web server, as if it were the client. When a client sends a request for new content, if the content is in the proxy's cache, the proxy will serve it from there. Otherwise, the proxy will forward this request to the web server and cache the new content.

**Encryption at the gateway.** The gateway parses the HTTP header in a packet's payload to identify the file path $F$ in a "GET" request. This is easy to do because the header appears at the start of the payload. For this task, the gateway does negligible work, resulting in less than 1% throughput loss over the RangeMatch encryption. Next, the gateway encrypts $F$ using the KeywordMatch scheme. In this case, it can use the deterministic version (discussed in §3.1) with no reduction in security because the proxy caches a set of *unique* file paths. Hence, using this scheme, the gateway encrypts $F$ into $\mathsf{AES}_k(F)$ and places it into a packet sent on the metadata channel.

**Processing at the proxy.** The proxy has a map of encrypted file path to encrypted file content. When the proxy receives a

packet, the proxy extracts $\mathsf{AES}_k(F)$ from the options header and looks it up in the cache *as if it were not encrypted*. The use of deterministic encryption enables the proxy to use a fast search data structure/index, such as a hash map, unchanged. We have two cases: there is a hit or a miss. For hit, the proxy assembles a packet header for reverse traffic and attaches to it the encrypted file content from the cache. Even without being able to encrypt IP addresses or ports, the proxy can create the header by reversing the encrypted information in the original header. For a miss, the proxy forwards the request to the web server. When receiving the response, the proxy caches it to serve future requests.

We can extend our proxy support to HTTP pipelining. In this case, an HTTP header may no longer appear at the beginning of a packet and it can appear in the middle of the packet or across packet boundaries. We do not give the details of our solution here, but at a high level, one can use the IDS approach in §5 to enable the web proxy to search for keywords on the payload (effectively parsing the HTTP header over encrypted data) and identify the encrypted file path. In this case though, MBArk can no longer support the web proxy in the header-only mode; it becomes a bytestream-aware application.

## 5. Middlebox: IDS and exfiltration detection

In intrusion detection [30] and data exfiltration, a middlebox (IDS) attempts to detect intrusion attacks by looking to match attack signatures against the traffic.

MBArk's IDS builds on top of BlindBox [28], an IDS which uses KeywordMatch to detect malicious signatures on encrypted traffic. Due to space limits, we cannot describe BlindBox, but refer the reader to [28] for details. MBArk improves both the performance and security of BlindBox as follows. BlindBox has a different threat model where the encryptor of the traffic is not trusted and hence it must run a sophisticated algorithm that is slow. MBArk's gateway is trusted and hence MBArk can avoid this slowdown. This improves the overall time to setup a connection by orders of magnitude, as discussed in §8. Each attack signature in BlindBox and Snort that has a regular expression $r$ must also have an exact match string $s$. Moreover, since there exists no encryption scheme that supports generic regular expressions efficiently over encrypted data, BlindBox decrypts any packet that matches $s$ so that it can run $r$ on it. Since the setup in MBArk is much cheaper, MBArk can expand the regular expressions into exact matches for a number of signatures. In this way, the number of rules that have regular expressions decreases up to half which means that less packets are getting decrypted.

## 6. Other middleboxes

MBArk supports other middleboxes as straightforward extensions of the ones we presented so far. MBArk supports the L4 balancer similarly to the NAT, the L7 balancer similarly to the web proxy, the IP forwarder similarly to a combination

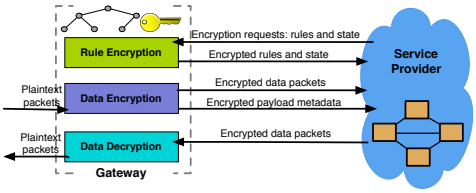



**Figure 5.** Communication between the cloud and gateway.

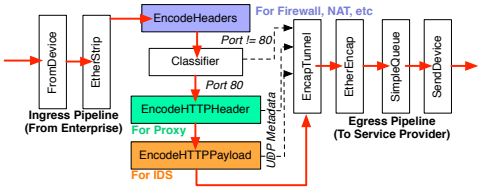



**Figure 6.** Data encryption (enterprise to cloud).

of NAT and firewall. In APLOMB [27], the VPN requires an APLOMB client as part of the VPN client; MBArk also adds a module which performs the encryption at the gateway. WAN optimizers run at the gateway in APLOMB so they can run on unencrypted traffic.

## 7. System implementation

We now describe the MBArk architecture and implementation. As discussed in §2, MBArk redirects traffic to the cloud and back for *middlebox processing* using a *redirection gateway*. We now present the implementation and architecture of the redirection gateway (§7.1), followed by the modifications we made to middleboxes to support MBArk (§7.2).

### 7.1 Redirection & gateway implementation

The gateway serves two purposes. First, it redirects traffic to/from the cloud for middlebox processing. Second, it provides the cloud with encryptions of IDS/FW rulesets and updates to the RangeMatch tree. A gateway (Figure 5) can be logically thought of as three services: the rule encryption service, the pipeline from the enterprise to the cloud (Data encryption), and the pipeline from the cloud to the enterprise (Data decryption). All three services share access to the RangeMatch tree and the private key $k$.

**Rule Encryption.** The rule encryption component provides the cloud provider with encrypted rules/policies to use at the middlebox. There are two possible ways rules can be generated. First, an enterprise may choose to generate their own rules, in which case, they send encrypted versions of the rules directly to the cloud. Rules can contain IP addresses, port numbers, and substrings of attack signatures; the first two must be encrypted with both keyword match and range match, the last needs only to be encrypted with keyword match. Alternately, the enterprise may opt in to a 'default' set of rules from the cloud provider, in which case the cloud provider sends the rules to the gateway which encrypts them and sends them back. The rule encryption component also sends rule updates. Whenever an adjustment is made to the RangeMatch table, it sends an update to the cloud

provider with the adjusted mappings/rules. If the gateway ever changes its key, the encryption component must signal to the cloud provider and re-encrypt all rules.

**Data Encryption.** In Figure 6, we show the DPDK-Click [19] packet processing pipeline that implements data encryption.Traffic that is returning from the Internet and traffic that is about to be sent to the Internet are both sent along this pipeline. This figure assumes the enterprise is already using IPv6, if it is not, the pipeline would also include a '4to6' element. In §2, we described how MBArk encodes packet content using the AES, Keyword Match, and Range Match encryption algorithms. The encrypted values are placed either directly back in to a data packet, or transmitted over a *metadata channel*. Below, we detail the three elements we implemented to encrypt the packet data:

*Encode Headers:* Encrypts the IP, TCP, and UDP headers, replacing all IP and Port numbers in the packet with values calculated using the *RangeMatch* encryption algorithm. Appends the AES-encrypted values to the IP options field of the data packet. Required to support all middleboxes.

*Encode HTTP Header.* Encrypts the HTTP header for *the first GET request only.* Does not modify the data packet itself, but instead places the keyword match encryptions of the values in a new packet sent over the metadata channel. The new packet marks (a) the encrypted 5-tuple flow ID for the packet, (b) that this is HTTP GET data for the proxy, and (c) the encrypted values. This element is only needed if there is an HTTP proxy enabled at the cloud, otherwise it can be disabled at the gateway.

*Encode HTTP Payload.* Encrypts the entire HTTP payload (as described in §5), placing the keyword match values in a new packet in the metadata channel. Unlike the other two elements, this element keeps per-flow state, reconstructing the TCP stream in order to generate keyword match tokens for keywords which are divided across two subsequent packets. The metadata channel packets contain (a) the encrypted 5-tuple for the packet, (b) that the packet contains keyword match data for the IDS, and (c) the encrypted values. This element is only needed if there is an IDS enabled at the cloud, otherwise it can be disabled at the gateway.

**Data Decryption.** Packets returning from the cloud follow a much simpler packet processing pipeline than those being encrypted. The packet payloads are decrypted using standard AES; the IP and port values from the options header are decrypted and then copied in to the packet header. The options header is removed. If the enterprise is running IPv4, the traffic is sent back through the 4to6 converter to convert the packet back to IPv4. The packet is then sent out to the Internet or the client at the enterprise.

### 7.2 Middlebox implementations

We implemented MBArk for all middleboxes listed in Table 1 using DPDK-Click [19], with varying levels of modification to a baseline (unencrypted) design required to make them compatible with MBArk. As we presented in §4, we adapted all header-only middleboxes *without modifying steady-state dataplane functions*, and hence preserving the performance properties of the existing middleboxes. We were able to do this because the encrypted values in the IPv6 header are stored in the normal IPv6 header formats, and because our encryption schemes preserve the exact match and range match semantics these devices use to operate correctly. All we need to do to be compatible with normal dataplane behavior is to encrypt all values in the rules and configuration files using the the range match encryption algorithm.

We do need to change the control logic and add one 'special case' check to the dataplane; this is to handle rule *updates*. The update code is triggered rarely – when the gateway changes its key $k$, or when there is an update to the Range Match tree. In this case the rules must be re-encrypted, and any per-session state (*e.g.* the flow table in the NAT) must be re-encrypted as well. When such an update is about to occur, the gateway first re-computes all rules *before* it begins to use the new encryption values on traffic. It signals to the cloud provider that it is about to perform a rule update. Then, all middleboxes transmit their rules / state tables to the gateway for re-encryption. Once all middleboxes have received their updated rules and state, the gateway 'switches' to the new ruleset, sending a signal packet in the dataplane. When a middlebox receives this signal packet (the only change we make to the dataplane is this) it switches out the ruleset to use the new values. [1]

Bytestream-aware middleboxes require changes in the dataplane as well as the control logic: MBArk makes reading the payload directly impossible, and hence these middleboxes must be modified to operate over the metadata channel instead. We re-implemented all such middleboxes from scratch. However, these changes need not be a performance penalty: as we show in §8, we wrote an HTTP proxy from scratch to cache HTTP content; this proxy achieves comparable performance to existing, unencrypted proxies despite our encryption. The primary difference between our proxy and a standard implementation is that the 'table' storing file names/URIs and the file data is now over encrypted values; the content served is also opaque, encrypted data. In addition, the file names/URIs are read from the metadata channel, rather than the primary connection with the data packets. Much of the complicated aspects of the codebase (*e.g.*, TCP session termination, storage and cache optimization, etc) can be implemented using out of the box libraries.

[1] There is a race condition for the flow state: consider a SYN packet which arrives just before the signal packet at a NAT. The NAT will encode the SYN packet in its state table using the old encryption scheme, and then immediately need to switch over to the new encryption scheme – before it has the new mapping for the new flow. Hence, a small number of packets may be misclassified during the transition.

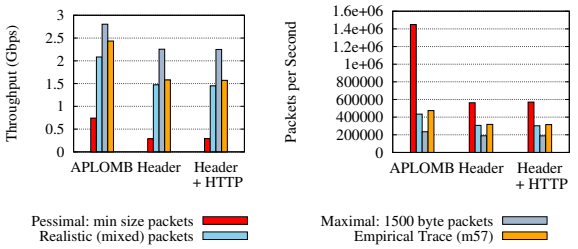



**Figure 7.** Throughput/Packets per Second on a single core at the stateless gateway.

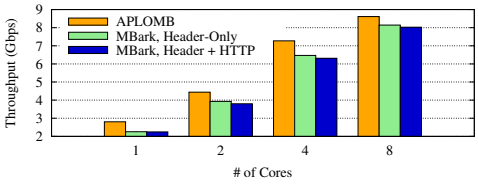



**Figure 8.** Gateway throughput with increasing parallelism.

## 8. Evaluation

As we showed in §7, MBArk supports all middlebox applications in typical outsourcing environments [11, 27] – including header-only middleboxes as well as bytestream-aware middleboxes.

We now investigate whether MBArk is practical from a performance perspective, looking at the overheads due to encryption (over the header or the payload) and redirection. MBArk has low hardware overheads at the enterprise: our eight-core server generates up to 8Gbps encrypted traffic to and from the cloud; we evaluate the gateway throughput in §7.1. Because most middleboxes are unchanged in the dataplane, throughput overheads at the cloud due to MBArk are negligible, typically under 1%, as we show in §4.

We ran our experiments using the implementation described in 7. Our prototype gateway 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. We deployed our prototype in our research lab and redirected traffic from a 3-server testbed through the gateway; these three client servers had the same hardware specifications as the server we used as our gateway. We deployed our middleboxes on Amazon EC2. For most experiments, we use a synthetic workload generated by the Pktgen-DPDK [33]; for experiments where an empirical trace is specified we use the m57 patents trace [9] and the ICTF 2010 trace [32].

### 8.1 Enterprise Performance

We first evaluate MBArk's overheads at the enterprise.

#### 8.1.1 Gateway

*How many servers does a typical enterprise require to outsource traffic to the cloud?* Figure 7 shows the gateway throughput when encrypting traffic to send to the cloud, first with normal redirection (as used in APLOMB [27]), then with MBArk's L3/L4-header encryption, and finally with

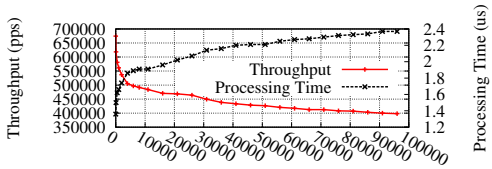



**Figure 9.** Throughput as number of rules for range encrypt increases.

L3/L4-header encryption as well as HTTP/proxy encryption. For empirical traffic traces with payload encryption (IDS) disabled, MBArk averages 1.5Gbps per core; for full-sized packets it achieves over 2Gbps. In scalability experiments (shown in Fig. 8), our 8-core server could forward at up to 8Gbps while encrypting headers and payloads with a combination of AES, KeywordMatch, and RangeMatch schemes.

With IDS enabled (not shown), throughput dropped to 240Mbps per core, suggesting that an enterprise would need a small cluster to reach higher bandwidth for these kinds of devices. While IDS introduces a reduction to about one-sixth relative to other encryption schemes, there is little difference between the HTTP overhead and the L3/L4 overhead, as the HTTP encryption only occurs on HTTP requests – a small fraction of packets.

*How do throughput and latency at the gateway scale with the number of rules for range encryption?* In §3.2, we discussed how our range encryption scheme stores encrypted values in a tree; every packet encryption requires a traversal of this tree. Hence, as the size of the tree grows, we can expect to require more time to process each packet and throughput to decrease. We measure this effect in Figure 9. On the $y_1$ axis, we show the aggregate per packet throughput at the gateway as the number of rules from 0 to 100k. The penalty here is logarithmic, which is the expected performance of the tree data structure. From 0-10k rules, throughput drops from 670Kpps to 480Kpps; after this point the performance penalty of additional rules tapers off. Adding an additional 90k rules drops throughput to 400Kpps. On the $y_2$ axis, we measure the processing time per packet, *i.e.*, the amount of time for the gateway to encrypt the packet; the processing time follows the same logarithmic trend.

*Is range encryption faster than existing order preserving schemes?* Our range encryption is the only encryption scheme that has low enough latency for packet processing while preserving the ordering information needed for firewall rules. We benchmarked BCLO and mOPE with 10K and 100K rules. With both 10K and 100K rules BCLO required 9333$\mu$s to encrypt a single value. For mOPE, it took 6640$\mu$s to encrypt with 10K rules and 8300$\mu$s with 1000K rules. RangeMatch, on the otherhand, takes only microseconds: 3$\mu$s with 100K rules, and 1.9$\mu$s with 10K rules. *What is the memory overhead of the stateful range map encryption scheme?* Storing 10k rules in memory requires 1.6MB, and storing 100k rules in memory requires 28.5MB – using

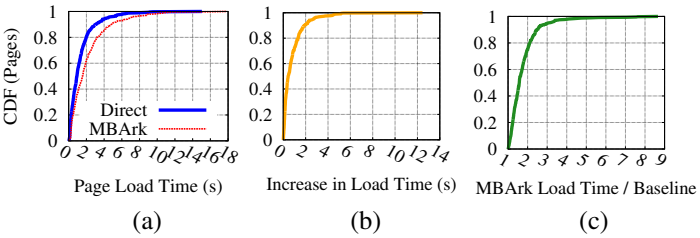



**Figure 10.** Page load times through MBArk compared to direct download.

unoptimized C++ objects. This state overhead is negligible on any modern server.

### 8.1.2 Client Performance

We use web performance to understand end-to-end client experience using MBArk. Figure 10 shows a CDF for the Alexa top-1000 sites loaded through our testbed, we plot (a) the page load times in seconds, (b) the absolute increase in page load times in seconds, and (c) the relative increase in page load time as a multiple of the baseline. Because of the 'bounce' redirection MBArk uses, all page load times increase; in the median case this increase is less than a second. At the 90th percentile, page loads increase by 2 seconds. Half of all pages see less than a 50% overhead in load times.

### 8.1.3 Bandwidth Overheads

We evaluate two costs: the increase in bandwidth due to our encryption and metadata, and the increase in bandwidth cost due to 'bounce' redirection.

*How much does MBArk encryption increase the amount of data sent to the cloud?* The gateway inflates the size of traffic due to three encryption costs: (1) If the enterprise uses IPv4, there is a 20-byte per-packet cost to convert from IPv4 to IPv6. If the enterprise uses IPv6 by default, there is no such cost. (2) If HTTP proxying is enabled, there are on average 132 bytes per request in additional encrypted data. (3) If HTTP IDS is enabled, there is at worst a 5× overhead on all HTTP payloads. We used the m57 trace to understand how these overheads would play out in aggregate for an enterprise. On the uplink, from the gateway to the middlebox service provider, traffic would increase by 2.5% due to encryption costs for a Header-Only Gateway. Traffic would increase by 4.3× on the uplink for a bytestream-aware gateway.

*How much does bandwidth increase between the gateway and the cloud from using MBArk? How much would this bandwidth increase an enterprises networking costs?* MBArk sends all network traffic to and from the middlebox service provider for processing, before sending that traffic out to the Internet at large. In NFV contexts, the clients' middlebox service provider and network connectivity provider are one and the same and one might expect costs for relaying the traffic to and from the middleboxes to be rolled in to one 'package.' However, in the APLOMB setting, the middlebox service provider is a cloud, meaning that the client must pay a third party ISP to transfer the data to and from the cloud, before paying that ISP a third time actually transfer the data over the network.

Using current bandwidth pricing [8, 20, 31], we can estimate how much outsourcing would increase overall bandwidth costs. Multi-site enterprises typically provision two kinds of networking costs: Internet access, and intra-domain connectivity. Internet access typically has high bandwidth but a lower SLA – 99 or 99.5% uptime; traffic may also be sent over shared Ethernet [8, 31]. Intra-domain connectivity usually has a private, virtual Ethernet link between sites of the company with a high SLA (over 99%) and lower bandwidth. This latter connectivity tends to cost about two orders of magnitude more than the former, because of the high SLA, extra configuration, and support required. Because bounce redirection is over the 'cheaper' link, the overall impact to bandwidth cost with header-only encryption given public sales numbers is between 15-50%; with bytestream aware encryption this cost increases to between 30-150%.

| Application | Baseline Throughput | MBArk Throughput |
|---|---|---|
| IP Firewall | 9.8Gbps | 9.8Gbps |
| NAT | 3.6Gbps | 3.5 Gbps |
| Load Balancer L4 | 9.8 Gbps | 9.8Gbps |
| Web Proxy | 1.1Gbps | 1.1Gbps |
| IDS | 85Mbps | 166Mbps [28] |

**Table 2.** Middlebox applications supported by Header-Only MBArk; throughput measured with an empirical traffic workload.

## 8.2 Middleboxes

We now evaluate the performance overhead at each middlebox. Overall, we find that throughput is negligibly impacted due to MBArk because MBArk keeps most dataplane operations unchanged.

*Is throughput reduced at the middleboxes due to MBArk?* Table 2 shows the throughput sustained for the apps we implemented. The IP Firewall, NAT, and Load Balancer are all 'header only' middleboxes; the results shown compare packet processing over the same dataplane, once with encrypted IPv6 data and once with unencrypted IPv4 data. The only middlebox for which any overhead is observable is the NAT – and this is a reduction of only 2.7%.

The IDS is twice faster than the baseline, which is Snort [30], which is slower than DPDK-Click pipeline our IDS is built on; nevertheless, this still shows we achieve comparable throughput.

We re-implemented the Web Proxy and IDS to enable the byte-stream aware operations they require over our encrypted data. The Web Proxy sustains the same throughput with and without encrypted data, but, as we will present later, does have a higher service time per cache hit. The IDS num-

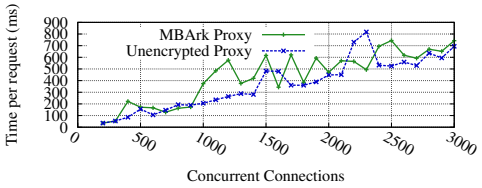



**Figure 11.** Access time per page against the number of concurrent connections at the proxy.

bers compare Snort (baseline) to the BlindBox implementation; this is not an apples-to-apples comparison as BlindBox performs mostly exact matches where Snort performs regular expressions.

In what follows, we provide some further middlebox-specific benchmarks for the firewall, proxy, and IDS.

**Firewalls:** *Does MBArk support all rules in a typical firewall configuration? How much does the ruleset "expand" due to encryption?* We tested our firewall with three actively-used rulesets provided to us by a network administrator at our institution; two rulesets were in a proprietary firewall format and one was a set of IPTables rules for a software Linux firewall. We were able to encode all rules using range and keyword match encryptions.

We also investigated how much the ruleset increased due to range encryption. Rules are typically encoded in the firewall as prefixes, hence a rule over the range [4.0.0.0, 4.255.255.255] is in practice implemented as a bitmask over the first 8 bits of every packet; checking to see if the packet matches the prefix 4.0.0.0/8. Using range encryption, the same prefix might be mapped to 9.128.0.0-10.128.0.0.0, resulting in two /9 prefix ranges in the final rule encoding: 9.128.0.0/9 or 10.128.0.0/9. As throughput for many middleboxes decreases with the number of rules, we were concerned that these mappings might degrade performance. However, in practice rules increased modestly, by between 5.8% and 10.2%.

**Proxy/Caching:** The throughput number shown in Table 2 is not the typical metric used to measure proxy performance. A better metric for proxies is how many connections the proxy can handle concurrently, and what time-to-service it offers each client. In Figure 11, we plot time-to-service against the number of concurrent connections, and see that it is on average higher for MBArk than the unencrypted proxy, by tens to hundreds of milliseconds per page. This is not due to computation costs, but instead, due to the fact that the encrypted HTTP header values are transmitted on a different channel than the primary data connection. The MBArk proxy needs to synchronize between these two flows, reading in HTTP header values from the metadata channel and then determining which connection they belong to; this synchronization cost is what increases the time to service. The synchronization cost would go down if all of the encrypted header values were stored in the HTTP packets themselves.

**Intrusion Detection:** Our IDS is based on BlindBox [28]. However, MBArk affords better performance and stronger privacy than BlindBox. BlindBox incurs a substantial 'setup cost' every time a client initiates a new connection. With MBArk, however, the gateway and the cloud maintain one, long-term persistent connection. Hence, this setup cost is paid once when the gateway is initially configured. This results in two benefits:

*(1) End-to-end performance improves.* Where BlindBox incurs an initial handshake of 414s [28] to open a new connection, clients under MBArk never pay this cost; instead they perform a normal TCP or SSL handshake of only 3-5 RTTs. In our testbed, this amounts to between 30 and 100 ms, depending on the site and protocol – an improvement of 4 orders of magnitude.

*(2) Security improves.* As we presented in §5, BlindBox operates at one of two security levels: a stronger security level for exact-match strings, and a weaker security level for regular expressions. However, MBArk can convert many regular expressions to exact matches without the performance overhead that BlindBox incurs. Consequently, MBArk can detect more then 80-88% of attacks using the higher security level, rather than 42-67% as with BlindBox:

## 9. Related work

Confidentiality of data in the cloud has been widely recognized as an important problem and researchers proposed solutions for generic applications [3], web applications [14, 26], filesystems [4, 15, 18], databases [25], and virtual machines [35]. However, there has been almost no work on data confidentiality for network processing in the cloud.

A few systems [21, 22] attempt to anonymize the packet stream in the hope of hiding the identity of the hosts. Unlike MBArk, this approach breaks certain middlebox functionality and does not hide the data content. Yamada et al. [34] show how one can perform some very limited processing on an SSL-encrypted packet. by using only the size of the data and the timing of packets.

As discussed, computing on encrypted data promises to provide both confidentiality and functionality. Nevertheless, generic cryptography schemes [6, 12] remain impractical by orders of magnitude [13]. CryptDB [25] introduced a new vision for building practical systems that compute on encrypted data, but none of its encryption schemes or systems techniques apply to our setting. The order-preserving encryption [5] used in CryptDB is much slower and less secure than RangeMatch. Finally, the database techniques in CryptDB do not apply to our networking setting.

The BlindBox [28] system enables running an IDS on encrypted traffic. Our IDS is built on top of BlindBox, but it is much more efficient and more secure than BlindBox as discussed in §5 and §8. Moreover, BlindBox focuses only on DPI and IDS and does not provide solutions for other middleboxes unlike MBArk.

## References

[1] *Proceedings of the 23rd ACM Symposium on Operating Sys-*

*tems Principles (SOSP)* (Cascais, Portugal, Oct. 2011).

[2] ARYAKA. WAN Optimization. http://www.aryaka.com/.

[3] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, CO, Oct. 2014).

[4] BLAZE, M. A Cryptographic File System for UNIX. In *1st ACM Conference on Communications and Computing Security* (Nov. 1993), pp. 9–16.

[5] BOLDYREVA, A., CHENETTE, N., LEE, Y., AND O'NEILL, A. Order-preserving symmetric encryption. In *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)* (Cologne, Germany, Apr. 2009), pp. 224–241.

[6] BONEH, D., SAHAI, A., AND WATERS, B. Functional encryption: Definitions and challenges. In *Proceedings of the 8th Theory of Cryptography Conference (TCC)* (Providence, Rhode Island, Mar. 2011).

[7] CLEARINGHOUSE, P. R. Chronology of data breaches. http://www.privacyrights.org/data-breach.

[8] COMCAST. Small Business Internet. http://business.comcast.com/internet/business-internet/plans-pricing.

[9] DIGITAL CORPORA. m57-Patents Scenario. http://digitalcorpora.org/corpora/scenarios/m57-patents-scenario.

[10] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. NFV Membership. https://portal.etsi.org/TBSiteMap/NFV/NFVMembership.aspx.

[11] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. NFV Whitepaper. https://portal.etsi.org/nfv/nfv_white_paper.pdf.

[12] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *Principles of the 41st Symposium on the Theory of Computation (STOC)* (Bethesda, MD, May-Jun 2009), pp. 169–178.

[13] GENTRY, C., HALEVI, S., AND SMART, N. P. Homomorphic evaluation of the AES circuit. Cryptology ePrint Archive, Report 2012/099, June 2012.

[14] GIFFIN, D. B., LEVY, A., STEFAN, D., TEREI, D., MAZIÈRES, D., MITCHELL, J. C., AND RUSSO, A. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012).

[15] GOH, E.-J., SHACHAM, H., MODADUGU, N., AND BONEH, D. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium* (Feb. 2003), Internet Society (ISOC), pp. 131–145.

[16] GOLDREICH, O. *Foundations of cryptography: Basic tools*. Cambridge University Press, 2003.

[17] GUPTA, P., AND MCKEOWN, N. Algorithms for packet classification. *IEEE Network 15*, 2 (Mar. 2001), 24–32.

[18] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable secure file sharing on untrusted storage. In *2nd USENIX conference on File and Storage Technologies (FAST '03)* (San Francisco, CA, Apr. 2003).

[19] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Trans. Comput. Syst. 18*, 3 (Aug. 2000), 263–297.

[20] MEGAPATH. Ethernet Data Plus. http://www.megapath.com/promos/ethernet-dataplus/.

[21] PANG, R., ALLMAN, M., PAXSON, V., AND LEE, J. The devil and packet trace anonymization. In *Computer Communication Review* (2006).

[22] PANG, R., AND PAXSON, V. A High-level Programming Environment for Packet Trace Anonymization and Transformation. In *Proc. ACM SIGCOMM* (Karlsruhe, Germany, 2003).

[23] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-time. *Computer Networks, 31*, 23-24 (Dec. 1999), 2435–2463.

[24] POPA, R. A., LI, F. H., AND ZELDOVICH, N. An ideal-security protocol for order-preserving encoding. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)* (San Francisco, CA, May 2013), pp. 463–477.

[25] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* [1], pp. 85–100.

[26] POPA, R. A., STARK, E., VALDEZ, S., HELFER, J., ZELDOVICH, N., KAASHOEK, M. F., AND BALAKRISHNAN, H. 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).

[27] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making Middleboxes Someone Else's Problem: Network Processing As a Cloud Service. In *Proc. ACM SIGCOMM* (Helsinki, Finland, 2012).

[28] SHERRY, J., LAN, C., POPA, R. A., AND RATNASAMY, S. Blindbox: Deep packet inspection over encrypted traffic. In *Cryptology ePrint Archive* (2015), no. 2015/264.

[29] SONG, D. X., WAGNER, D., AND PERRIG, A. Practical techniques for searches on encrypted data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy (IEEE S&P)* (Oakland, CA, May 2000), pp. 44–55.

[30] THE SNORT PROJECT. Snort users manual, 2014. Version 2.9.7.

[31] VERIZON. High Speed Internet Packages. http://www.verizon.com/smallbusiness/products/business-internet/broadband-packages/.

[32] VIGNA, G. ICTF Data. https://ictf.cs.ucsb.edu/#/.

[33] WILES, K. Pktgen version 2.7.7 using DPDK-1.7.1. https://github.com/Pktgen/Pktgen-DPDK/.

[34] YAMADA, A., SAITAMA MIYAKE, Y., TAKEMORI, K., STUDER, A., AND PERRIG, A. Intrusion detection for encrypted web accesses. In *21st International Conference on Advanced Information Networking and Applications Workshops* (2007).

[35] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* [1].

[36] ZSCALAR. Internet Security. http://www.zscaler.com/.