Fast Privacy-Preserving Network Function Outsourcing[☆]

Hassan Jameel Asghar[a,b,*], Emiliano De Cristofaro[c], Guillaume Jourjon[b], Mohammed Ali Kaafar[a,b], Laurent Mathy[d], Luca Melis[c], Craig Russell[b], Mang Yu[b]

[a] *Macquarie University, Sydney, Australia*
[b] *Data61, CSIRO, Sydney, Australia*
[c] *University College London, London, UK*
[d] *University of Liège, Liège, Belgium*

Abstract

In this paper, we present the design and implementation of SplitBox, a system for privacy-preserving processing of network functions outsourced to cloud middleboxes—i.e., without revealing the policies governing these functions. SplitBox is built to provide privacy for a generic network function that abstracts the functionality of a variety of network functions and associated policies, including firewalls, virtual LANs, network address translators (NATs), deep packet inspection, and load balancers. We present a scalable design aiming to provide high throughput and low latency, by distributing functionalities to a few virtual machines (VMs), while providing provably secure guarantees. We implement SplitBox inside FastClick, an extension of the Click modular router, using Intel's DPDK to handle packet I/O. We evaluate our prototype experimentally to find its bottlenecks and stress-test its different components, vis-à-vis two widely used network functions, i.e., firewall and VLAN tagging. Our evaluation shows that, on commodity hardware, SplitBox can process packets close to line rate (i.e., 8.9Gbps) with up to 50 traversed policies.

Keywords: NFV, Privacy, Middlebox

[☆]This article is based, in part, on [1] published at the 3rd ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddleBox 2016).
[*]Corresponding author

Email addresses: `hassan.asghar@mq.edu.au` (Hassan Jameel Asghar), `e.decristofaro@ucl.ac.uk` (Emiliano De Cristofaro), `guillaume.jourjon@data61.csiro.au` (Guillaume Jourjon), `dali.kaafar@mq.edu.au` (Mohammed Ali Kaafar), `Laurent.Mathy@ulg.ac.be` (Laurent Mathy), `uca.melis.14@ucl.ac.uk` (Luca Melis), `craig.russell@data61.csiro.au` (Craig Russell), `g100335@gmail.com` (Mang Yu)

1. Introduction

Network functions support a variety of functionalities – typically configured via a set of *policies* – including network address translation (NAT), deep packet inspection, and firewalling. Traditionally, network functions have been implemented on hardware middleboxes deployed at the edge of an organization's network, however, these appliances often yield high infrastructure and management costs [2]. As a result, more and more organizations have taken advantage of advances in cloud computing and virtualization technologies to adopt Network Function Virtualization (NFV) [3]. With NFV, functions are implemented as software processes, outsourced to virtual machines running on commodity servers, leading to a significant reduction in associated costs and complexity, as well as flexibility in re-purposing generic hardware for a multitude of functions [3, 4, 5, 6, 7, 8, 9, 10].

At the same time, however, outsourcing network functions to third parties may introduce serious security and privacy threats. Consider, for instance, the case of a firewall: in the traditional setting, firewall rules (or policies) are hidden from potentially prying eyes, except for what can be inferred from observing incoming/outgoing traffic. Whereas, with NFV, policies could be accessed by third parties running processes on the same cloud infrastructure, and are inherently available to a cloud provider that may be compromised or not fully trusted [11]. This raises worrying concerns as the disclosure of the policies may reveal sensitive details about an organization's network, such as the IP addresses of hosts or the topology of the private network [12, 13].

These concerns motivate the need to enable network function outsourcing while protecting the confidentiality of the policies. There are at least two approaches to do so: one is to rely on virtual machine isolation [14], aiming to guarantee that a (client's) virtual machine is isolated from the rest of the processes/virtual machines running on a given server. Alas, this is far from trivial and infeasible with commodity virtual machine hypervisors. Another approach is to build on cryptographic primitives that provably minimize the amount of information disclosed to the cloud provider [15]. In theory, one could rely on tools like garbled circuits and fully homomorphic encryption to perform computation or evaluate functions, privately, without disclosing sensitive information. However, these tools are too expensive for most NFV settings [15], where low latency and minimization of computational overhead are crucial requirements.

Arguably, optimal solutions to the problem should not only guarantee confidentiality of the network policies—and with minimal trust assumptions—but also basic performance requirements such as low latency and high throughput. Moreover, clients (whose network functions are being outsourced) should be as thin as possible, otherwise outsourcing would only be nominal. Additionally, one should also provide compatibility with existing infrastructure (i.e., third parties do not need to implement new protocols) and support a wide range of functionalities. Previous work, thoroughly reviewed in Section 7, has made several attempts in this direction [12, 13, 15, 16, 17], but has fallen short of simultaneously achieving all these requirements. For instance, some of them are limited to simple firewalls [12, 13], whereas, we support more general network functions in which packets can be (privately) modified in different ways (e.g., changing destination IP). Other works such as Embark [16] provide a more general applicability to network functions, but at the cost of relying on the client itself to perform a significant portion of the functionality.

In this paper, we present the design, analysis and implementation of SplitBox, a system that supports fast and privacy-preserving network function outsourcing with high throughput and low latency. The privacy goal is to hide the policies governing the network functions from the VMs implementing the functions, thus not allowing the cloud (and third parties) learn network function policies of the client. Our intuition is to leverage the distributed nature of cloud VMs: rather than relying on a single one, we distribute functionality to a few VMs residing on multiple clouds or multiple compute nodes in the same cloud, and provide a scalable and provably secure solution. SplitBox supports the privacy-preserving modification of a packet's contents through the notion of secret sharing [18], ensuring that a collusion of any number of compute nodes less than the total does not reveal the modified content (including whether the packet has been tagged for a drop for the specific case of firewalls). This enables us to add more complex modifications, thus addressing a challenging open problem. In summary, we make the following contributions:

1. We introduce an abstract mathematical definition of a network function which is rich enough to capture many network functions, including, but not limited to, firewalls, virtual LAN (VLAN) tagging, and NAT. This helps us build solutions that guarantee privacy for a generic network function, and can be applied to many network functions used in practice.
2. We design and implement SplitBox, a system geared for privately and efficiently computing the aforementioned abstract network function, so that the "honest-but-curious" cloud, modeled as several VM middleboxes, cannot learn network policies. SplitBox relies on the distributed nature of cloud VMs, assuming that an adversary does not corrupt all middleboxes simultaneously, and only uses relatively inexpensive cryptographic primitives, while supporting a wide range of network policies (specifically, those that can be modeled as substring matching and replacement of packet contents). We provide security proofs of our construction using the real/ideal simulation paradigm [19].
3. We present an extensive evaluation of SplitBox, considering two network functions as examples, to thoroughly assess its feasibility in the wild. We consider a firewall application since it is widely used, including in related work [12, 13, 16], as well as VLAN tagging. The latter is more complex in terms of functionality, since it modifies packet header rather than simply allowing/dropping a packet. We show that on a firewall test case, SplitBox achieves the same throughput of the non-private version with 9.4Gbps and 1.5kB-sized packets when up to 10 rules are fired, and a decrease in performance limited to 5% (i.e., 8.9Gbps) with 50 rules.

Paper Organization. The rest of the paper is organized as follows. Next sec-

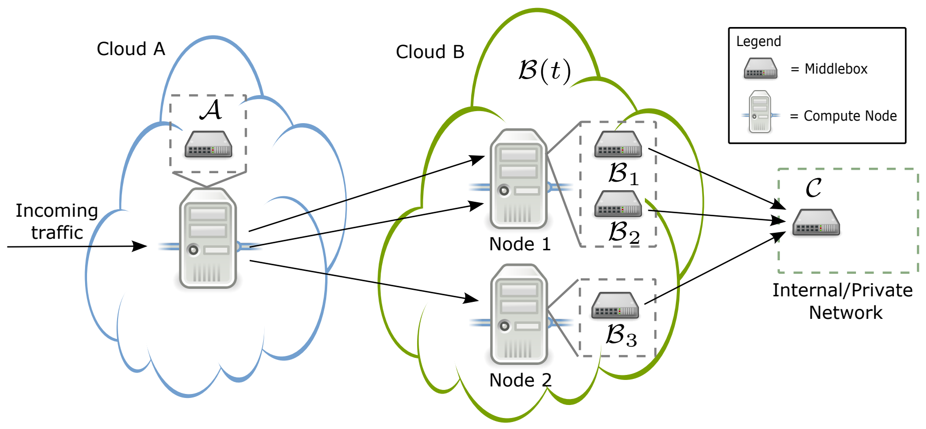


Figure 1: Our system model with Cloud A hosting MB $\mathcal{A}$ as a VM in one of its compute nodes. Cloud B hosts the MBs $\mathcal{B}(t)$ with $t = 3$ as VMs (not all t reside on the same compute node). The client MB $\mathcal{C}$ resides at the edge of the client's internal network. $\mathcal{A}$ and $\mathcal{B}(t)$ collaboratively compute network functions for the client.

tion presents the threat model as well as a generic model of network functions which abstracts the functionality of network functions. Then, Section 3 introduces the design of the SplitBox system and its security analysis, while Section 4 discusses our prototype implementation. After presenting an experimental evaluation of SplitBox in Section 5, Section 7 reviews related work. Finally, the paper concludes in Section 8.

2. System Model

2.1. Problem Statement

The problem addressed in this paper is where a client outsources one or more of its network functions (e.g., firewall) to the cloud. The cloud processes these network functions on behalf of the client, i.e., executes policies on incoming network traffic destined to the client, and sends the processed packets to the client. In the traditional setting, these network functions would be implemented at the edge of the client network. Hence any third-party can only see inbound/outbound traffic to/from the client network. In particular, third parties do not learn the policies implemented by the network functions (beyond what is deducible by observing the traffic). The client wishes that the same privacy be applicable to its network functions when they are being processed in the NFV setting, where the cloud is executes these network functions on behalf of the client. In particular, any third party, including the cloud, should not learn the policies implemented by these network functions. This essentially means that the cloud should process these policies *correctly* while remaining *oblivious* to them. In the following, we define the setting and these requirements more precisely, including trust assumptions, describe our proposed solution and prove its security and correctness.

2.2. Entities

As illustrated in Figure 1, SplitBox involves two types of cloud middleboxes (MBs): an *entry* MB $\mathcal{A}$, hosted on a cloud provider (Cloud A) and a set of $t \geq 2$ cloud MBs, $\mathcal{B}(t)$, which collaboratively compute a network function on behalf of a client, e.g., hosted on Cloud B. The client has its own client MB, denoted as $\mathcal{C}$, at the edge of its internal network. At a high level, $\mathcal{A}$ receives a packet, performs some computation on it, "splits" the result into t parts, and forwards part j to $\mathcal{B}_j \in \mathcal{B}(t)$. $\mathcal{B}_j$ performs local computations and forwards its part to $\mathcal{C}$, which reconstructs the network function's final result. (Note that there is also a direct link between $\mathcal{A}$ and $\mathcal{C}$ which is not illustrated in Figure 1 to ease presentation.)

2.3. Threat Model

Our main privacy goal is to limit information leakage of client's network policies. In a way, we set to emulate the "traditional" setting where the network function, in its entirety, is implemented client-side, i.e., at $\mathcal{C}$, thus revealing no information to the cloud. We assume an honest-but-curious adversary [20] with access to one or more compute nodes in the cloud, which can only learn whether or not the current packet "matches" some *unknown* policy of the network function. More specifically, we assume that the adversary does not control (have access to) both $\mathcal{A}$ and a MB in $\mathcal{B}(t)$ at the same time, which reflects our vision in which $\mathcal{A}$ runs on a different cloud provider (cloud A) than $\mathcal{B}(t)$ (cloud B). The adversary can control up to $t-1$ MBs from $\mathcal{B}(t)$, i.e., it does not compromise all the MBs at the same time, as not all MBs in $\mathcal{B}(t)$ reside on the same compute node. In other words, our threat model reduces to assuming that the adversary does not have access, simultaneously, to more than one cloud provider, and to all compute nodes in cloud B, in line with recent proposals around the idea of a *super cloud* [21].[1] Finally, note that the system model in Figure 1 implicitly targets inbound traffic. Outbound traffic is in general trusted, hence, we only focus on inbound traffic.

Out-of-scope Threats. In our setting, entry MB $\mathcal{A}$ or a third-party adversary could observe traffic inbound to $\mathcal{A}$ and traffic outbound from $\mathcal{C}$ to infer policies. But notice that this is also possible in the traditional setting if traffic is not encrypted. Even if traffic is encrypted, it still does not provide protection against an adversary generating traffic destined to the client. Thus, we can assume that the initial packet (at least its header) can remain in the clear. If privacy of packet contents (not just policies) is required, encryption can be used to

[1]Note that we are making a distinction between a compute node (physical server) and a virtual machine (MB). Our assumption is that only some of the physical servers in cloud B are accessible to the adversary. Hence we require the MBs from $\mathcal{B}(t)$, i.e., virtual machines, to not be running on the same physical server. This can be implemented in policy, requiring each of the MBs from $\mathcal{B}(t)$ to run on a different compute node. With this policy, our assumption translates to the adversary having access to at most $t-1$ compute nodes in cloud B (and hence at most $t-1$ MBs from $\mathcal{B}(t)$).

provide privacy against third-party adversaries, and/or traffic can be sent to $\mathcal{C}$ first before sending to $\mathcal{A}$. However, this is beyond the scope of this work.

Difference with Embark [16]. In our solution, inbound traffic goes directly to the cloud MB, as opposed to other approaches, such as Embark [16], where inbound traffic first goes to the client MB, which pre-processes it (e.g., encrypts it) and then routes it to the cloud MB, which in turn processes and sends the traffic back to the client MB. This is an inherently easier problem, as the client MB knows *exactly* which policies a given packet is expected to match, and can therefore "embed" the answers within the packet, as is done in Embark. In contrast, in our setting, the traffic goes straight to the cloud, which yields a harder problem, as these policies need to be hidden from the cloud. Note that there does not seem to be a straightforward way to modify Embark to provide privacy in our setting without compromising efficiency.

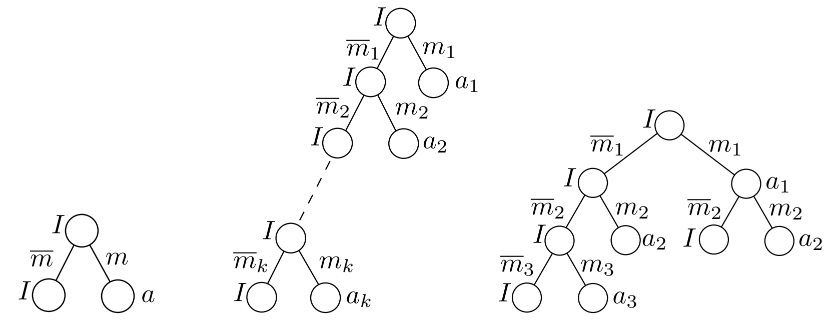
2.4. Network Function Model

We define a packet x as a binary string of arbitrary length, However, network functions will be applicable to the first n bits of x only. We define a *matching* function as $m : \{0,1\}^n \to \{0,1\}$, and its complement (i.e., $1-m$) as $\overline{m}$. Note that both are boolean functions. We also define *action* functions as transformations $a : \{0,1\}^n \to \{0,1\}^n$. Functions are evaluated on the first n bits of x, so, if $|x| > n$, a keeps x unaltered after the n-th bit. We also define the identity action function $I(x) = x$.

Let M and A, respectively, denote finite sets of matching and action functions. We define a *network function*, $\psi = (M, A)$, as a binary tree with edge set M and node set A such that each node is an action function $a \in A$ and each edge is either a matching function $m \in M$ or a complement $\overline{m}$ of a matching function $m \in M$. A node is either a leaf or a parent node, the latter having two children. The left child is I, the edge to the right child is a $m \in M$, the one connecting the left child is its complement $\overline{m}$. The root node is I. Examples of network functions are given in Figure 2.

Policies. Let $\psi = (M, A)$ be a network function. Then, there is a binary relation from M to A, such that for each (m, a) from this relation, there exists a parent node in ψ whose left child is connected via $\overline{m}$ and the right child via m, and the right child is a. We call each pair (m, a) in ψ a *policy.* A policy can also be represented as a subtree of ψ as shown in Figure 2(a). Policies serve as building blocks of a network function. The set of policies of ψ is the set of *distinct* policies (m, a) in ψ.

Network functions. A network function $\psi(x)$ is evaluated on a packet x using Algorithm 1. We create a separate writeable copy $x_{\mathtt{w}}$ to ensure the matching functions are applied on the "unmodified" x, i.e., $x_{\mathtt{r}}$, and not on $x_{\mathtt{w}}$, which is modified by the action functions. When a leaf node is entered, the network function terminates. Figure 2(b) shows a network function with k distinct policies: whenever a match is found, the corresponding action is performed and the function terminates. The function in Figure 2(c) has 3 distinct policies,



(a) Policy (m, a) as a tree.

(b) Network function with k distinct policies.

(c) Network function with 3 distinct policies.

Figure 2: Examples of network functions as binary trees.

Algorithm 1: `Traversal`

```
   Input: Packet $x$, network function $\psi$.
1  Make a read-only copy $x_{\mathtt{r}}$ and a writable copy $x_{\mathtt{w}}$ of $x$.
2  Start from the root node.
3  Compute $x_{\mathtt{w}} \leftarrow a(x_{\mathtt{w}})$, where $a$ is the current node.
4  if the current node is a leaf node then
5      output $x_{\mathtt{w}}$ and stop.
6  else
7      Compute $m(x_{\mathtt{r}})$, where $m$ is the right hand side edge.
8      if $m(x_{\mathtt{r}}) = 1$ then
9          Move to the right child node.
10     else
11         Move to the left child node.
12 Go to step 3.
```

$(m_1, a_1), (m_2, a_2)$ and (m_3, a_3), and (m_2, a_2) is repeated twice. This function does not terminate immediately after a match has been found (e.g., path $m_1 m_2$). Since $a \circ I = I \circ a = a$, we can easily "plug" individual policy trees to construct more complex network functions.

Remark. Our (abstract) definition of network functions yields several advantages compared to a standard representation using vectors of elements corresponding to different fields of a packet, and defining network functions as function composition [15]. First, defining packets as strings removes the need for padding. Second, we can support *branching*, i.e., network functions that do not necessarily apply all policies on a packet, by including multiple exit points (leaf nodes in our tree model).

2.5. Restriction of Policies

We restrict m and a, respectively, to substring matching and substitution. We introduce the *"don't care bit"*, denoted by $*$ in our alphabet. Given strings $x \in \{0,1\}^n$ and $y \in \{0,1,*\}^n$, $x = y$ if $x(i) = y(i)$ for all $i \in [n]$ s.t. $y(i) \neq *$. A matching function m is defined as $m(x) = 1$ if $x(1,n) = \mu$ and 0 otherwise, where $\mu \in \{0,1,*\}^n$. We call μ the *match* of m.

Example 1. Let $n = 4$. And let the match μ of a matching function m be $\mu = 10*0$. Let $x_1 = 1010, x_2 = 1000$ and $x_3 = 1001$ be three different packets. Then $m(x_1) = m(x_2) = 1$, whereas $m(x_3) = 0$. $\square$

To define the action function, we introduce substring replacement: given $x \in \{0,1\}^n$ and $z \in \{0,1,*\}^n$, $x \leftarrow z$ represents replacing each $x(i)$ with $z(i)$ if $z(i) \neq *$, and leaving $x(i)$ as is if $z(i) = *$, for all $i \in [n]$. Then, a is defined as $a(x) = x(1,n) \leftarrow \alpha$, where $\alpha \in \{0,1,*\}^n$. We call α the *action* of a. For the identity action function I, $\alpha = *^n$. Given $z \in \{0,1,*\}^n$, the *projection* of z, denoted π_z, is a string $\in \{0,1\}^n$, s.t. $\pi_z(i) = 1$ if $z(i) \in \{0,1\}$ and $\pi_z(i) = 0$ if $z(i) = *$. *Masking* a packet x, using $\pi_z \in \{0,1\}^n$, is denoted as $\omega(\pi_z, x)$, returning x' s.t. $x'(i) = x(i)$ if $\pi_z(i) = 1$ and $x'(i) = 0$ otherwise. Although $\omega(\pi_z, x)$ is defined for $x \in \{0,1,*\}^n$, we will use it exclusively for an $x \in \{0,1\}^n$. Figure 3 illustrates the projection and masking functions.

Lastly, we use $\mathbb{H} : \{0,1\}^n \to \{0,1\}^q$ for a cryptographic hash function; $\oplus$ for bitwise XOR, $\mathrm{wt}(x)$ for the Hamming weight of x, while $x \leftarrow_{\$} \{0,1\}^n$ denotes sampling a binary string of length n uniformly at random.

Example 2. Consider a simple firewall with the policy that only packets with destination ports 80 or 22 are allowed. This can be represented by the tree shown in Figure 4. Here m_1 is described by the match μ_1 whose bits corresponding to the destination port are set to the binary representation of 80 and the rest by the don't care bit $*$. Likewise m_2 corresponds to destination port 22. The match μ_3 of matching function m_3 is set to $*^n$, and therefore accepts every packet. The action function a is $x(1,n) \leftarrow 0^n$, which we model as dropping a packet.[2] $\square$

2.6. Generic Network Functions

Coverage. Our abstract definition of network functions captures many network functions used in practice. These include firewalls, access control lists (ACLs), NATs, virtual LANs (VLANs), and load balancers. These usually go through a matching step to inspect some "parts" of a packet, and, if a match is found, modify contents. With firewalls, this might also entail dropping a packet. In our model, packets are not dropped at $\mathcal{B}(t)$, but are (privately) tagged to be dropped. When $\mathcal{C}$ receives the packet parts from $\mathcal{B}(t)$, the reconstructed packet

[2]This is not the only way to model dropping. In our implementation, we introduce another bit which acts as a flag to indicate dropping/blocking a packet.

z	0	$*$	1	$*$	$*$	0
π_z	1	0	1	0	0	1
$\omega(\pi_z, z)$	0	0	1	0	0	0
$\omega(\pi_z, x)$	1	0	0	0	0	$*$
x	1	$*$	0	1	1	$*$

Figure 3: The projection and masking functions.

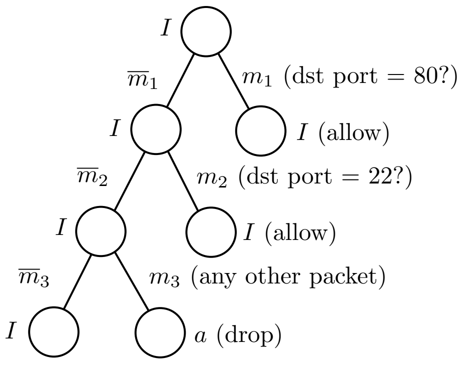


Figure 4: Network function for a simple firewall.

can have a bit within $x(1, n)$ set to 0 in case it is supposed to be dropped. Likewise, some QoS functions (e.g., rate limiters) can also be implemented by inserting a drop bit within the first n bits of a packet x.

Stateful network functions. Some network functions are stateful, i.e., they maintain a table of states, somewhat resembling dynamically generated policies. Upon arrival of a packet, the state table is consulted first, if no match is found, it is further processed as per the regular (static) policies. For instance, a stateful firewall may keep a list of currently open TCP/IP connections. Since a state can be modeled as a dynamically generated policy, our model of network functions can easily handle this by appending dynamic policies on top of the policy tree. However, in SplitBox, these dynamic policies can only be added by $\mathcal{C}$. We discuss this in more detail in Section 6.

Chaining. Our definitions also support function *chaining*, e.g., ψ_1's output is ψ_2's input. However, in our "default" SplitBox solution, chaining is not possible as outputs of the MBs in $\mathcal{B}(t)$ need to be combined in order to reconstruct a transformed packet. For chaining to work, ψ_2 needs to know the output of network function ψ_1, but, if ψ_2 only needs the original input x, instead of the overwritten copy $x_{\mathtt{w}}$, chaining can work by giving ψ_2 an auxiliary input – i.e., the share resulting from network function ψ_1, on which it can apply its own actions. We discuss this again in Section 6.

3. SplitBox

This section presents the design of the SplitBox system.

3.1. Requirements

SplitBox is designed by taking into account four main requirements: privacy, efficiency, thin client, and compatibility, as discussed next.

Privacy. Ideally, SplitBox should simulate a setting where $\mathcal{A}$ learns only the input packet, and $\mathcal{B}(t)$ learn neither the input nor the modified packet. We come close to achieving this as the MBs $\mathcal{B}(t)$ learn the projection π_μ and the output $m(x)$ for each $m \in M$, however, they do not learn the match μ for any $m \in M$ beyond what can be learned from π_μ. Although this could potentially reveal which field of the packet the current matching function corresponds to, it is not a major limitation as this information might anyway be obvious from the type of NFV considered. For instance, in case of a firewall, the IP fields will obviously be part of its policies.

Efficiency. SplitBox should be computationally fast, i.e., processing traffic at near-to-typical middlebox line rates, and limit MB-to-MB communication overhead. This makes it impossible to use (expensive) public-key operations as well as some simple solutions: for instance, one could let $\mathcal{A}$ compute all the matches from the set of matching functions M, then send the (encrypted) results of these matches to $\mathcal{B}(t)$, which could in turn execute actions based on the network tree. However, communication complexity would be proportional to $|M|$: depending on the number of policies, this may result in significantly larger packets sent to $\mathcal{B}(t)$ and severely affect throughput. Moreover, based on the network tree, not all matches need to be computed beforehand, and in this case this approach would be very inefficient. Note that we cannot reveal the result of the match to $\mathcal{A}$, as $\mathcal{A}$ already receives the packet x in the clear, and could deduce the match of the matching function. Likewise, any solution that requires back and forth communication between $\mathcal{A}$ and $\mathcal{B}(t)$ is not desirable as it would effect throughput.

Thin client. It is also crucial to impose minimal overhead on $\mathcal{C}$, otherwise we would nullify the benefits provided to the client by the outsourcing paradigm, i.e., reduced infrastructure costs, efficiency, and flexibility.

Compatibility. Third-parties should be oblivious to the virtual setting. This precludes implementing custom protocols for third-parties sending/receiving traffic from/to the cloud, which is done for instance by BlindBox [17].

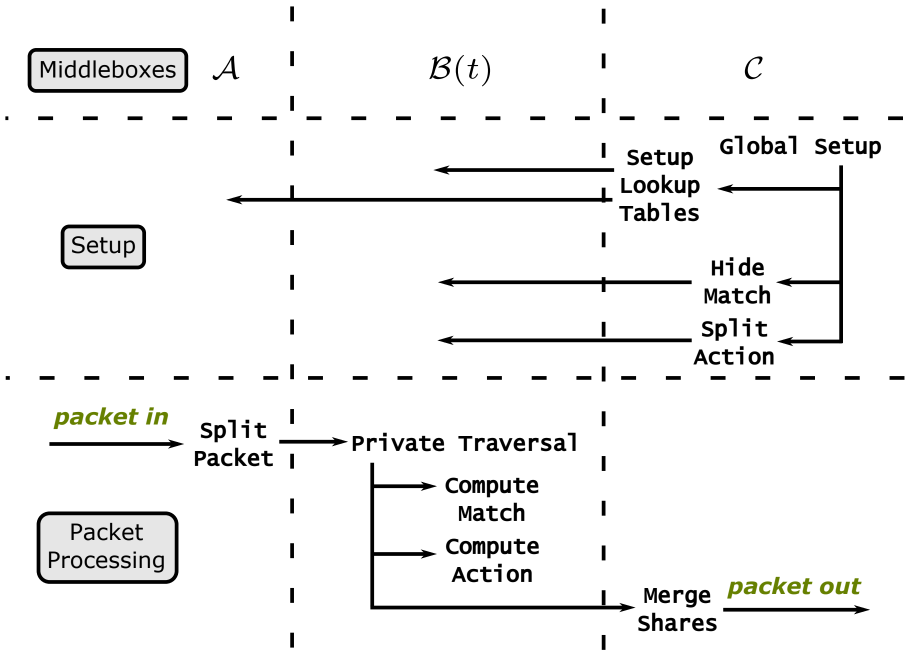


Figure 5: Breakdown of algorithms executed by each MB in SplitBox.

3.2. The SplitBox System

To illustrate the idea behind SplitBox, let us assume that the set of policies ψ only includes a single policy, (m, a). The strategy followed by SplitBox to hide m is to let $\mathcal{C}$ blind μ by XORing it with a random binary string s, and sending the hash of the result to each MB in $\mathcal{B}(t)$. Then, to hide a, $\mathcal{C}$ computes t *shares* of the action α using a t-out-of-t secret sharing scheme, and sends share j to $\mathcal{B}_j$. When a packet x arrives, $\mathcal{A}$ encrypts it by XORing it with the blind s, and sends the encrypted version to the MBs in $\mathcal{B}(t)$, which can then compute matches and actions on this encrypted packet.

In the rest of this section, we present SplitBox using a set of algorithms, grouped based on the MB executing them. Figure 5 shows a high-level overview of all the algorithms computed by each MB. We assume ψ_{priv} to be the private version of ψ, whose matching and action functions are replaced by unique identifiers.

Middlebox $\mathcal{C}$. The initial setup is performed by $\mathcal{C}$ via Algorithm 2. This includes creating lookup tables (Algorithm 3), hiding the matching functions (Algorithm 4), and splitting the action functions (Algorithm 5). There are two lookup tables in Algorithm 3: S for $\mathcal{A}$ and $\tilde{S}$ for $\mathcal{B}(t)$. Table S contains l "blinds" which are random binary strings used to encrypt a packet. For each $s \in S$ and $m \in M$, the portion of the blind corresponding to the projection of the match μ is extracted and then XORed with μ. Finally, this value is hashed using $\mathbb{H}$ and stored in the corresponding row of $\tilde{S}$. The `Hide Match` algorithm simply sends the projection π_μ of each match μ to $\mathcal{B}(t)$. This tells $\mathcal{B}(t)$ which

Algorithm 2: `Global Setup` ($\mathcal{C}$)

```
Input: Parameters $n$ and $l$, network function $\psi = (M, A)$.
1  for $j = 1$ to $t$ do
2      Send $\psi_{\text{priv}}$ to $\mathcal{B}_j$.
3  Run Setup Lookup Tables with parameter $l$, $M$.
4  for each $m \in M$ do
5      Run Hide Match algorithm.
6  for each $a \in A$ do
7      Run Split Action algorithm.
```

Algorithm 3: `Setup Lookup Tables` ($\mathcal{C}$)

```
Input: Parameter $l$, set $M$.
1  Initialize empty table $S$ with $l$ cells.
2  Initialize empty table $\tilde{S}$ with $l \times |M|$ cells.
3  for $i = 1$ to $l$ do
4      Sample $s_i \leftarrow_{\$} \{0,1\}^n$.
5      Insert $s_i$ in cell $i$ of $S$.
6      for $j = 1$ to $|M|$ do
7          Compute $\tilde{s}_{i,j} = \omega(\pi_{\mu_j}, s_i)$, where $\mu_j$ is the match of $m_j$.
8          Compute $\mathbb{H}(\mu_j \oplus \tilde{s}_{i,j})$.
9          Insert $\mathbb{H}(\mu_j \oplus \tilde{s}_{i,j})$ in cell $(i,j)$ of $\tilde{S}$.
10 Send $S$ to $\mathcal{A}$.
11 Send $\tilde{S}$ to $\mathcal{B}(t)$.
```

Algorithm 4: `Hide Match` ($\mathcal{C}$)

```
Input: Matching function $m \in M$ with match $\mu$.
1  Send $\pi_\mu$ to $\mathcal{B}(t)$.
```

locations of the incoming packet are relevant for the current match. The `Split Action` algorithm computes t shares of the action α and action projection π_α, for each $a \in A$, and sends them to $\mathcal{B}(t)$.

$\mathcal{C}$ uses one more algorithm, Algorithm 6, to reconstruct the transformed packet. This algorithm XORs the cumulative action shares α'_j and cumulative action projection shares β'_j from $\mathcal{B}_j$ to compute the final action α' and action projection β'. It also XORs the encrypted packet received from $\mathcal{A}$ with the current blind s in the lookup table S, in order to reconstruct the final packet. Note that dropping a packet is modeled as setting $x(1, n)$ to 0^n.

Middlebox $\mathcal{A}$. This MB only runs Algorithm 7, which keeps a counter initialized to 0 and incremented when a new packet x arrives. The value of the counter corresponds to a blind in the lookup table S, thus, its range is $[l]$ (barring the initial value of 0). The algorithm makes two copies of an incoming packet x, $x_{\mathtt{r}}$ (read-only copy) for matching to be sent to $\mathcal{B}(t)$, and $x_{\mathtt{w}}$ (writeable copy) for action functions to be sent to $\mathcal{C}$. Both $x_{\mathtt{r}}$ and $x_{\mathtt{w}}$ are XORed with the blind in S corresponding to the counter. The current counter value is also given to $\mathcal{B}(t)$ and $\mathcal{C}$.

Algorithm 5: `Split Action` ($\mathcal{C}$)

```
   Input: Action function $a \in A$ with action $\alpha$.
1  Sample $\alpha_1, \alpha_2, \ldots, \alpha_{t-1} \leftarrow_\$ \{0,1\}^n$.
2  Let $\tilde{\alpha} = \omega(\pi_\alpha, \alpha)$. Compute $\alpha_t = \tilde{\alpha} \oplus \alpha_1 \oplus \cdots \oplus \alpha_{t-1}$.
3  Sample $\beta_1, \beta_2, \ldots, \beta_{t-1} \leftarrow_\$ \{0,1\}^n$.
4  Compute $\beta_t = \pi_\alpha \oplus \beta_1 \oplus \cdots \oplus \beta_{t-1}$.
5  for $j = 1$ to $t$ do
6      Give $\alpha_j, \beta_j$ to $\mathcal{B}_j$.
```

Algorithm 6: `Merge Shares` ($\mathcal{C}$)

```
   Input: Index $i$, packet copy $x_{\mathtt{w}}$, $\alpha'_j$ and $\beta'_j$ from $\mathcal{B}_j$ for $j \in [t]$.
1  Compute $\alpha' \leftarrow \alpha'_1 \oplus \cdots \oplus \alpha'_t$.
2  Compute $\beta' \leftarrow \beta'_1 \oplus \cdots \oplus \beta'_t$.
3  Compute $x \leftarrow x_{\mathtt{w}} \oplus s_i$, where $s_i \in S$.
4  for $i = 1$ to $n$ do
5      if $\beta'(i) = 1$ then
6          $x(i) \leftarrow \alpha'(i)$
7  if $x(1, n) = 0^n$ then
8      Drop $x$.
9  else
10     Forward $x$.
```

Middleboxes $\mathcal{B}(t)$. Each MB $\mathcal{B}_j$ performs a private version of the `Traversal` algorithm as shown in Algorithm 8. $\mathcal{B}_j$ first initializes cumulative action strings

Algorithm 7: `Split Packet` ($\mathcal{A}$)

```
   Input: Packet $x$, lookup table $S$.
1  Get the index $i \in [l]$ corresponding to the current value of the counter.
2  Let $x_{\mathtt{w}} \leftarrow x \oplus s_i$ (writeable copy), where $s_i \in S$.
3  Compute $x_{\mathtt{r}} \leftarrow x(1, n) \oplus s_i$ (read-only copy), where $s_i \in S$.
4  for $j = 1$ to $t$ do
5      Send $x_{\mathtt{r}}$, $i$ to $\mathcal{B}_j$.
6  Send $x_{\mathtt{w}}$, $i$ to $\mathcal{C}$.
```

Algorithm 8: `Private Traversal` ($\mathcal{B}(t)$)

```
   Input: Index $i$, read-only copy $x_{\mathtt{r}}$, network function $\psi_{\text{priv}}$.
1  Initialize empty strings $\alpha'_j \leftarrow 0^n$ and $\beta'_j \leftarrow 0^n$.
2  Start from the root node.
3  Update $\alpha'_j$ and $\beta'_j$ by running the Compute Action algorithm on the current
     node $a$.
4  if the current node is a leaf node then
5      Send $i$, $\alpha'_j$ and $\beta'_j$ to party $\mathcal{C}$ and stop.
6  else
7      Run Compute Match algorithm on $i$, $m$ and $x_{\mathtt{r}}$, where $m$ is the right hand
         side edge.
8      if Compute Match outputs 1 then
9          Go to the right child node.
10     else
11         Go to the left child node.
12 Go to step 3.
```

α'_j and cumulative action projection strings β'_j as strings of all zeros. Within the `Private Traversal` algorithm, $\mathcal{B}_j$ executes the action functions using Algorithm 9 and matching functions using Algorithm 10. The `Compute Action` algorithm essentially updates α'_j and β'_j by XORing with the action share and action projection share of the current action. The `Compute Match` algorithm uses the read-only copy $x_{\mathtt{r}}$. It extracts the bits of $x_{\mathtt{r}}$ corresponding to the current match projection π_μ. It then looks up the counter value i (sent by $\mathcal{A}$) and the index of the matching function in the lookup table $\tilde{S}$ and extracts the hashed match. This is then compared with the hash of the relevant bits of $x_{\mathtt{r}}$.

3.3. Analysis

Correctness. Given $\psi = (M, A)$, for a matching function $m \in M$, SplitBox correctly computes the match as long as m is represented as substring matching. If m is an equality or range test for powers of 2 in binary (e.g., IP addresses in `127.`$*$`.`$*$`.32` to `127.`$*$`.`$*$`.64`), then it can also be computed. Our model also allows for arbitrary ranges by dividing m into smaller matches that check equality matching of individual bits. SplitBox can correctly compute action functions as long as: (a) they are applied to the initial packet x only, and not on its transformed versions; (b) any two action projections β_i and β_j do not

```
Algorithm 9: Compute Action ($\mathcal{B}(t)$)
  Input: Pair of cumulative action and cumulative action projection shares
         $(\alpha'_j, \beta'_j)$ of $\mathcal{B}_j$, pair of action and action projection shares $(\alpha_j, \beta_j)$ of
         action function $a \in A$ of $\mathcal{B}_j$.
1 Compute $\alpha'_j \leftarrow \alpha'_j \oplus \alpha_j$.
2 Compute $\beta'_j \leftarrow \beta'_j \oplus \beta_j$.
3 Output $\alpha'_j$, $\beta'_j$.
```

```
Algorithm 10: Compute Match ($\mathcal{B}(t)$)
  Input: Read-only copy $x_{\mathtt{r}}$, index $i \in [l]$, lookup table $\tilde{S}$, index $j \in [|M|]$ of
         $m_j \in M$ with match $\mu_j$.
1 Lookup table $\tilde{S}$ at index $(i, j)$ to obtain $\mathbb{H}(\tilde{s}_{i,j})$.
2 Extract $\tilde{x}_{\mathtt{r}} \leftarrow \omega(\pi_{\mu_j}, x_{\mathtt{r}})$.
3 Compute $\mathbb{H}(\tilde{x}_{\mathtt{r}})$.
4 if $\mathbb{H}(\tilde{x}_{\mathtt{r}}) = \mathbb{H}(\mu_j \oplus \tilde{s}_{i,j})$ then          // $m(x) = 1$
5     Output 1.
6 else                                                   // $m(x) = 0$
7     Output 0.
```

overlap on their non-zero bits. However, this does not restrict the number of times the identity function I can be applied, as its action projection is 0^n.

Security. We divide our security analysis in two parts. We first prove that an honest-but-curious $\mathcal{A}$ does not learn the network function $\psi = (M, A)$ including the number of matching and action functions, i.e., $|M|$ or $|A|$. Then we prove that any coalition of less than t MBS in $\mathcal{B}(t)$ does not learn the match μ of each matching function in M, and the action or the action projection of every action function in A for up to l packets (where l is the number of blinds in the lookup table). The coalition may learn the match projection (the bits of the packet the match applies to), or the result of the unknown match (i.e., whether it evaluates to 1 or not). We also discuss additional security strategies, e.g., how to ensure security for greater than l executions. Overall this ensures security of our proposed solution with the assumption that clouds A and B do not collude (as required). In the following, we make these assumptions more rigorous and provide formal proofs of security under these assumptions.

As explained in Section 2, we assume a passive (honest-but-curious) adversary $\mathcal{E}$ which can either corrupt $\mathcal{A}$, or up to $t-1$ MBs from $\mathcal{B}(t)$. Let Π denote our SplitBox scheme. Before a formal security analysis, we first discuss the assumptions and privacy requirements of the scheme Π:

- The parameter n is public.
- $\mathcal{A}$ should not know the network function $\psi = (M, A)$ (not even $|M|$ or $|A|$). It does however see x in clear.
- Each $\mathcal{B}_j \in \mathcal{B}(t)$ knows the projection π_μ of the match μ of each matching function $m \in M$. It should not, however, learn the match μ of any

matching function $m \in M$ (beyond what is learnable through π_μ). It also knows the result of all the matching functions; this may include matching functions that are not necessary to compute $\psi(x)$ for each packet x, i.e., the subset of matching functions that are in the path that exit the graph ψ given x. Since $\mathcal{B}_j$ can always access the hash function $\mathbb{H}$ offline, it can check all matching functions $m \in M$ for their output (not necessarily in the path of ψ). We therefore need to make this explicit.

- Each MB $\mathcal{B}_j \in \mathcal{B}(t)$ should not know x. Furthermore, for any two packets x_1 and x_2, it should not know which bits of x_1 and x_2 are the same, beyond what is learnable through the result of the subset of the matching functions used in $\psi(x_1)$ and $\psi(x_2)$. In particular, if a matching function m has projection π_μ for its match μ, it should only learn that the bits corresponding to π_μ are the same if $m(x_1) = m(x_2) = 1$. If $m(x_1) \neq m(x_2)$, $\mathcal{B}_j$ should not learn whether individual bits corresponding to π_μ are the same or different (except when $\text{wt}(\pi_\mu) = 1$). This is the reason for using the hash function $\mathbb{H}$ and the encryption of packet through XORing with the blinds in our scheme. We call this property, *indistinguishability of packet contents.*

- Any coalition of $t-1$ MBs in $\mathcal{B}(t)$ should not be able to learn the action α and the action projection β of every action $a \in A$.

Let us denote random variables $\mathcal{I}$ and $\mathcal{O}$ denoting the input and output of an MB (or a subset of MBs) corrupted by $\mathcal{E}$.Further denote the random variable X representing the packet x, and D representing the description of the network function ψ. The output of the network function ψ on input from X is denoted $\psi(X)$. We first describe the ideal functionality, denoted IDEAL, followed by the real setting, denoted REAL.

IDEAL$(\psi, \mathcal{S})$. We assume a trusted third party $\mathcal{T}$, which communicates with each of the MBs via a secure and private link. $\mathcal{T}$ is given the network function $\psi = (M, A)$. MBs $\mathcal{B}(t)$ are given the "index set" of M (i.e., $\{1, 2, \ldots, |M|\}$) together with the matching projections π_μ, for the match μ of each matching function $m \in M$. Notice that, since in our protocol, we leak this information, we need to make this explicit. MB $\mathcal{A}$ receives a packet x and hands it over to $\mathcal{T}$. $\mathcal{T}$ computes $x' = \psi(x)$. It hands over x' to $\mathcal{C}$. Since in our protocol, we leak the information about the output of the matching functions, $\mathcal{T}$ also hands over the result of each matching function $m \in M$ to the parties $\mathcal{B}(t)$. The simulator $\mathcal{S}$ serves as the adversary in the IDEAL setting. Succinctly, IDEAL$(\psi, \mathcal{S})$ is the tuple $(\mathcal{I}, \mathcal{O}, X, \psi(X), D)$, where the random variables correspond to the MB (or subset of MBs) controlled by $\mathcal{S}$.

REAL$(\Pi, \mathcal{E})$. Our real setting is simply the execution of our scheme in the presence of the adversary $\mathcal{E}$. It again represents the tuple $(\mathcal{I}, \mathcal{O}, X, \psi(X), D)$ where each random variable corresponds to the MB (or subset of MBs) corrupted by $\mathcal{E}$. Naturally, depending on whether $\mathcal{E}$ corrupts MB $\mathcal{A}$ or upto $t-1$ MBs in $\mathcal{B}(t)$, the simulator $\mathcal{S}$ in the ideal setting will be different (and so will be the random variables in the tuple $(\mathcal{I}, \mathcal{O}, X, \psi(X), D)$).

With these two settings, we want to show that for every probabilistic polynomial time adversary $\mathcal{E}$ there exists a probabilistic polynomial time adversary $\mathcal{S}$, such that

$$\text{REAL}(\Pi, \mathcal{E}) \approx_{\text{c}} \text{IDEAL}(\psi, \mathcal{S}),$$

where $\approx_{\text{c}}$ denotes computational indistinguishability. If the above holds, we say that Π privately processes ψ. In our proofs, we implicitly use the assumption that given binary strings c and $c_1, \ldots, c_t$ such that $c_1, \ldots, c_{t-1}$ are random binary strings in $\{0,1\}^n$, and $c_t = c_1 \oplus \cdots \oplus c_{t-1} \oplus c$, then for any subset of strings from $c_1, \ldots, c_t$, denoted $C(t-1)$, with cardinality $\leq t-1$, the following holds: $\mathbb{P}[c|C(t-1)] = \mathbb{P}[c] = 2^{-n}$. The proof of this assumption is standard. We use this result whenever we talk about t-out-of-t shares in our proposed solution.

Our main results are as follows.

Theorem 1. *The scheme* Π *privately processes* ψ *against an honest-but-curious* $\mathcal{E} = \mathcal{A}$*.*

Proof. Before receiving any packet, the simulator $\mathcal{S}$ samples l uniformly random strings $s_i \in \{0,1\}^n$ to construct the lookup table S and gives it to $\mathcal{E}$. It initializes its counter to 0. Upon receiving a packet x, $\mathcal{S}$ forwards it to $\mathcal{T}$. For $\mathcal{E}$, $\mathcal{S}$ first gets the current value of the counter $i \in [l]$. It further samples a uniformly random $r \in \{0,1\}^n$ and constructs $x_{\mathtt{w}} \leftarrow x \oplus r$. It computes t shares of r, the jth share of which is denoted r_j. Finally it obtains $x_{\mathtt{r}} \leftarrow x(1,n) \oplus s_i$ by looking up the counter value i in the table S. Finally $\mathcal{S}$ gives $x_{\mathtt{r}}$, i, $x_{\mathtt{w}}$ and the t shares of r to $\mathcal{E}$. Once the counter i reaches l, $\mathcal{S}$ resets it to 0.

Since the input to party $\mathcal{A}$ is the same as the input packet x, we have that $\mathcal{I} = X$ (which holds both in the ideal and real setting). The output $\mathcal{O}$ is distributed in the exact same manner in the two worlds. Since the output is generated without any knowledge of the network function ψ, we have that D is the same in the ideal and real world. Finally, the output of ψ is not revealed in the two worlds. Hence $\text{REAL}(\Pi, \mathcal{E}) = \text{IDEAL}(\psi, \mathcal{S}) \Rightarrow \text{REAL}(\Pi, \mathcal{E}) \approx_{\text{c}} \text{IDEAL}(\psi, \mathcal{S})$. □

As discussed in Section 3.3, if the match of a matching function is small, the adversary can brute-force the hash function $\mathbb{H}$ to find its pre-image. Thus, our security proof for $\mathcal{E} \subset \mathcal{B}(t)$ requires that the minimum Hamming weight of a match μ in the set of matching functions M should be large enough for brute-force to be infeasible. Furthermore, our security proof applies only when the blinds are used once, i.e., for counter values $\leq l$ without reset. See the discussion following the next theorem for our proposed mitigation strategy for security, when the counter completes its cycle.

Theorem 2. *Suppose* $\delta = \min_\mu wt(\pi_\mu)$*, for all matching functions* $m \in M$*. The scheme* Π *privately processes* ψ *for up to* l *inputs (packets), against an honest-but-curious* $\mathcal{E} \subset \mathcal{B}(t)$ *in the random oracle model.*

Proof. Let $\mathcal{R} : \{0,1\}^* \to \{0,1\}^q$ denote the random oracle. Before receiving any packet, the simulator $\mathcal{S}$ simulates the lookup table $\tilde{S}$ as follows. For each

$m \in M$, given the projection π_μ of its match μ, it generates l binary strings by sampling a random bit where $\pi_\mu(i) = 1$ and placing a 0 otherwise. For each such string, $\mathcal{S}$ samples a uniform random binary string of length q. $\mathcal{S}$ creates two tables. One is the lookup table $\tilde{S}$, and the other its personal table $\hat{S}$. The table $\hat{S}$ contains the pre-images of the entries in $\tilde{S}$. It hands over $\tilde{S}$ to each MB in $\mathcal{E}$. For each policy $(m, a) \in \psi$, it generates $|\mathcal{E}|$ random binary strings α_j and β_j of length n, for $1 \leq j \leq |\mathcal{E}|$, and gives each pair (α_j, β_j) to a separate MB in $\mathcal{E}$. $\mathcal{S}$ initiates a counter i initially set to 0.

Upon receiving the result of the matching functions in M from $\mathcal{T}$, indicating the arrival of a new packet, $\mathcal{S}$ first generates a random binary string as $x_{\mathtt{w}}$ and $|\mathcal{E}|$ random binary strings of length n (to simulate the r_j's). $\mathcal{S}$ initializes an empty string $x_{\mathtt{r}}$. For each matching function m that outputs 1, $\mathcal{S}$ looks up its table $\hat{S}$ and the projection π_μ, where μ is the match of the matching function, and replaces the corresponding bits of $x_{\mathtt{r}}$ with the corresponding bits of the input string to the lookup table $\hat{S}$. Finally, for all bits of $x_{\mathtt{r}}$ that are not set, $\mathcal{S}$ replaces them with uniform random bits. It hands over $x_{\mathtt{w}}$, $x_{\mathtt{r}}$ and r_j to each MB in $\mathcal{E}$, together with the current counter value i.

For any oracle query from an MB $\mathcal{B}_j \in \mathcal{E}$, $\mathcal{S}$ first looks at its table $\hat{S}$ and sees if an entry exists. If an entry exists, $\mathcal{S}$ outputs the corresponding output from the table $\hat{S}$. If an entry does not exist, $\mathcal{E}$ outputs a uniform random string of length q, and stores the input and the output by appending it to the table $\hat{S}$.

It is easy to see that the distribution of the variables $(\mathcal{I}, \mathcal{O}, X, \psi(X), D)$ for each MB in $\mathcal{E}$ is the same as in the real setting, for any $\mathcal{E}$, such that $|\mathcal{E}| < t$, for any value of the counter $i \leq l$, and for a polynomial in δ number of oracle queries. Therefore $\textsc{real}(\Pi, \mathcal{E}) \approx_{\mathrm{c}} \textsc{ideal}(\psi, \mathcal{S})$. $\square$

Security Consequences. Here we highlight two important points. First, if SplitBox is used for match projections with low Hamming weight, then an adversarial MB in $\mathcal{B}(t)$ could brute-force $\mathbb{H}$ to find its pre-image. This would reveal $\mu \oplus s$ for some blind s, allowing the adversary to learn more than simply looking at the output of m. Namely, if $m(x) = 0$, she learns which relevant bits of an incoming packet x do not match with the stored match. This is why we use the hash function $\mathbb{H}$, as it does not allow $\mathcal{B}(t)$ to learn more than the output of m.

Second, the length of the look-up table l should ideally be large enough so that the same blind is not re-used before a long period of time. However, high throughput would require a prohibitively large value of l. Thus, we adopt the following strategy: with probability $0 < 1 - \rho < 1$, $\mathcal{A}$ sends a uniform random string from $\{0, 1\}^n$ (dummy packet), rather than the next packet in the queue. As a result, any middlebox in $\mathcal{B}(t)$ that attempts to compare two packets using the same blind (according to the value of the counter $i \in [l]$) does not know for certain whether or not the result corresponds to two actual packets (the probability is ρ^2). This however reduces the (effective) throughput by a factor of ρ. Naturally, $\mathcal{A}$ needs to indicate to $\mathcal{C}$ which packet is dummy. This can be done by sending a bit through $\mathcal{B}(t)$ to $\mathcal{C}$, using a t-out-of-t secret sharing scheme

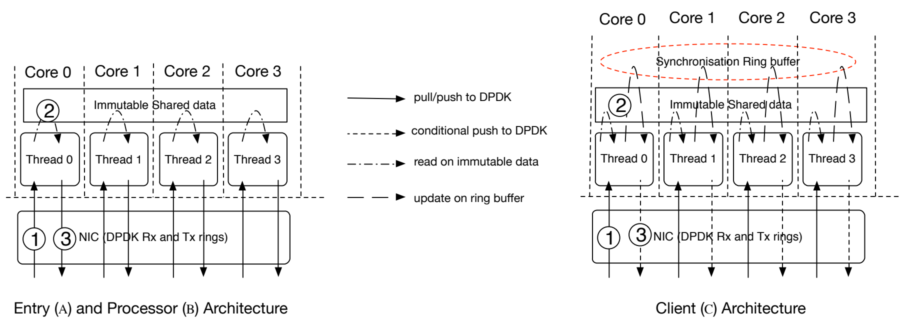


Figure 6: The setup of three SplitBox elements: `Entry`, `Processor`, and `Client`.

(XORing with random bits).

4. Implementation

We now present a prototype implementation of SplitBox.[3] Later in Section 5, we will also present a thorough evaluation of its performance in real-world settings. We implement SplitBox, in C++, inside FastClick [22], an extension of the Click modular router [23]. We choose FastClick as it provides fast user-space packet I/O and easy configuration via automatic handling of multi-threading and multiple hardware queues. We also use Intel DPDK [24] as the underlying packet I/O framework.

More specifically, we implement three FastClick elements: (i) an element `Entry` corresponding to $\mathcal{A}$; (ii) `Processor` realizing $\mathcal{B}$; and (iii) `Client` corresponding, to $\mathcal{C}$. Figure 6 outlines the setup of these elements as well as their interactions with concurrent shared data structures. The shared data structures are presented in Table 1, along with their associated sizes and location. Specifically, Figure 6 presents how each element is processing packets from the time they are pulled from, till the time they are pushed to DPDK rings.

In particular, note that all elements follow a Run-to-Completion model and that both `Processor` and `Entry` rely on the same architecture where they only perform read operations to shared data structures, while `Client` needs to write in a shared ring buffer in order to execute the `Merge Shares` algorithm.

Network Functions. We choose to implement and evaluate two network functions – namely, a stateless, ACL-based firewall and VLAN tagging network management – which follow a decision tree similar to the one in Figure 2(b). These two functions are implemented in a similar way, and rely on a "SplitBox Byte" (SByte), as shown in Figure 7. The purpose of SByte is to "mark" a packet, e.g., marking a packet for blocking (in the case of a firewall). SByte is used in the `Processor` element marking the packet from 0 to 255. Each value

[3]FastClick Elements are available on `https://bitbucket.csiro.au/users/jou008/repos/splitbox-source/`

Shared Data	**Size**	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$
Blinds	$l \times n$	✓	×	✓
Actions	$\|A\| \times n$	×	✓	×
Action Projections	$\|A\| \times n$	×	✓	×
Hashes	$l \times \|M\| \times q$	×	✓	×
Synchronization/ Merge Ring	$3 \times \|M\| \times \text{pkt\_size}$	×	×	✓

Table 1: Data structures shared among cores and their size. M is the set of matching functions, A the set of action functions, n the mask size, l the number of rows in the lookup table and q the hash size.

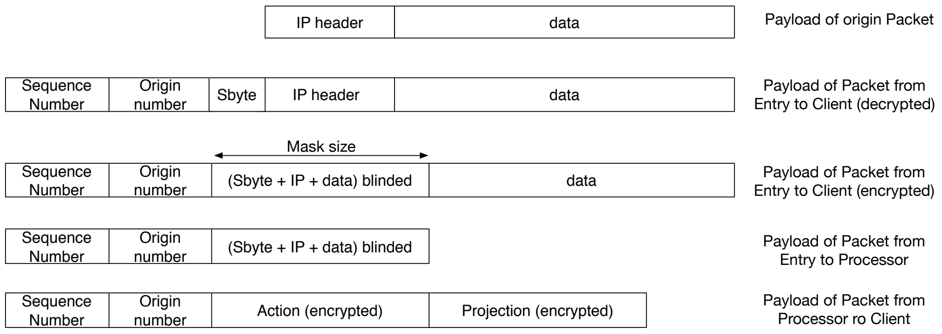

IP header
data
Payload of origin Packet
Sequence Number
Origin number
Sbyte
IP header
data
Payload of Packet from Entry to Client (decrypted)
Mask size
Sequence Number
Origin number
(Sbyte + IP + data) blinded
data
Payload of Packet from Entry to Client (encrypted)
Sequence Number
Origin number
(Sbyte + IP + data) blinded
Payload of Packet from Entry to Processor
Sequence Number
Origin number
Action (encrypted)
Projection (encrypted)
Payload of Packet from Processor ro Client

Figure 7: Packet structures in SplitBox.

corresponds to an action function configured by $\mathcal{C}$ using Algorithm 5. In the `Client` element, this algorithm configures up to $|M|$ ClickOS out-going pipes with various actions. For the firewall, one of these pipes corresponds to the blocking action, and implements ClickOS's `discard()` action, whereas, in the VLAN setting, each pipe encapsulates the packet in an Ethernet frame with the VLAN ID corresponding to the SByte number.

Entry. The `Entry` element is responsible for the `Split Packet` algorithm. It follows the processing model shown in Figure 6, where each thread pulls packets from the `Rx` ring from DPDK, obfuscates m bytes from the original packets, and then sends via the `Tx` ring the obfuscated packet to the `Client` and the obfuscation part to the `Processors`. In these packets, as shown in Figure 7, the `Entry` adds three fields at the beginning of the payload: the sequence number, the origin number, and the SByte. The sequence number identifies packets from each element during the merging (synchronization) algorithm and is incremented once per packet in the `Entry`. The origin number allows the `Client` to identify the provenance of a packet with a given sequence number. We use 0, 1, and 2 to identify, respectively, the `Entry`, `Processor` 0, and `Processor` 1. Finally, we add one more byte (SByte), and blind it in order to mark and perform any kind of policies as explained above. Overall, during this process, an `Entry` thread requests a read per packet from the shared structure Blind.

Processor. The `Processor` element implements the $\mathcal{B}_j$ MBs, and executes the `Private Traversal`, `Compute Match`, and `Compute Action` algorithms. Similar to the `Entry`, each thread in the `Processor` pulls packets from DPDK `Rx` ring, performs the algorithms in a Run-To-Completion configuration, and sends to the `Client` a packet containing the sequence number, an updated origin number, and the correct action and projection. The structure of these packets is illustrated in Figure 7. Note that $\mathbb{H}$ is implemented using OpenSSL's SHA-1, aiming to achieve a compromise between security, digest length, and speed. While faster hashing functions are available, they are not cryptographic hash functions, thus they might be invertible and/or lead to larger amount of collisions. On the other hand, we do not want hash functions which have very large message digests (leading to overly large lookup tables), or which are more computationally expensive. Overall, this element performs a significant number of lookups in the shared data structures (Projection, Action, and Hashes from Table 1), however, multiple hash computations per packet yield relatively low overhead, as we show in Section 5.

Client. Finally, the `Client` element implements $\mathcal{C}$, executing the `Merge Shares` algorithm. This algorithm is based on a ring buffer (Merge Ring in Table 1) shared among all threads on this element. In our C++ prototype, this ring buffer is implemented as a `std:vector` class of fixed size 1024. Each entry in this structure stores the packet from the `Entry` and the two `Processor` elements. As shown in Figure 6, this corresponds to an update on the shared structure, per packet, and, potentially, one push to DPDK `Tx` ring when the three packets have been received. The other algorithms of $\mathcal{C}$, used to configure the above three elements, are executed outside the FastClick elements.

5. Performance Evaluation

We now discuss maximum achievable throughput of our prototype while maintaining a packet loss rate inferior to 0.1% (PLR $< 0.1\%$) and the associated latency per packet as a function of traversed rules. Note that a PLR of less than 0.1% is merely used as a benchmark for an acceptable limit of packet loss. This is well below what is normally considered a problematic packet loss rate. In all experiments, we present aggregated results from both VLAN and firewall network functions. Overall, based on a thorough analysis, we show that SplitBox's cryptographic layer does not significantly affect overall performance as opposed to shared-data structures look-ups.

5.1. Experimental Setup

Experiments were performed using a lab testbed of four identically configured commodity servers (Dell PowerEdge R210s), respectively for one `Entry`, two `Processors`, and one `Client` elements. Each server is equipped with a quad-core Intel X3430 2.4GHz CPU and 16GB of RAM, as well as a dual-port Intel X710 NIC controlled by Intel DPDK for packet transmission between the

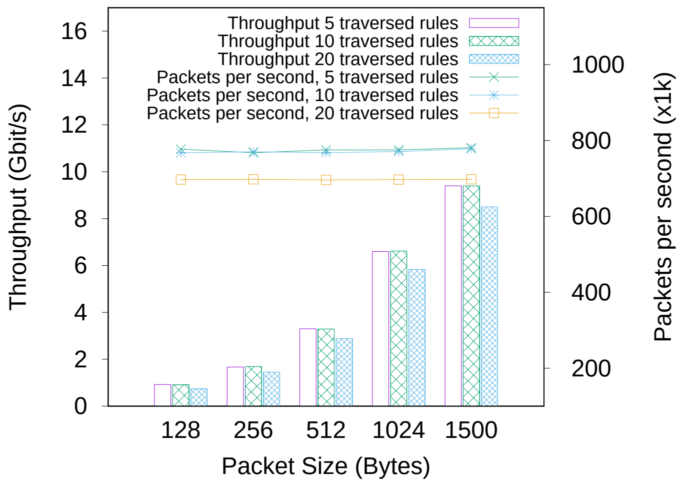


Figure 8: Maximum achievable throughput (PLR<0.1%) and packet per second in function of packet size and number of traversed rules.

elements (in addition to a separate Linux-controlled NIC for management access). The servers run Ubuntu 16.04.1 LTS with Linux kernel 4.4.0-57, and Intel DPDK version 2.2.0. We used a topology similar to that depicted in Figure 1, with two Cloud B middleboxes running as `Processor` elements. To generate test traffic, we used a Spirent TestCenter chassis equipped with a 2-port Hypermetrics CV 10G SFP+ module, running firmware version 4.24.1026. The Spirent generator acts as both traffic source and sink, allowing us to measure performance metrics very accurately, including throughput, loss, and latency for multiple concurrent flows, while generating traffic up to 10Gbps.

In all our tests, we set both our system and the baseline IP filter to always store 1000 possible rules and matches. We configured the Spirent traffic generator to create various types of traffic in order to stress-test the network functions. In particular, we considered two traffic flow distributions emulating both a worst-case scenario and a more realistic type of load. With the former, we generated uniformly distributed traffic where each flow accounts for the same amount of traffic, and matches a single rule in the VLAN tagging or the firewall scenarios. For instance, when considering 10 rules, we generated 10 different flows, each of them accounting for 10% of the total traffic. With the latter, each flow accounts for a different portion of the total traffic, following a Zipf distribution, since prior work has shown that, in a production firewall/network management infrastructure, only a few rules get matched most of the time [25].

5.2. Throughput and Delay

Our first experiment aimed to explore how packet size affects overall performance of SplitBox. Figure 8 shows performance in terms of maximum achievable throughput, as well as packets per second, as a function of packet size and number of traversed rules. In this experiment, we configure SplitBox to perform

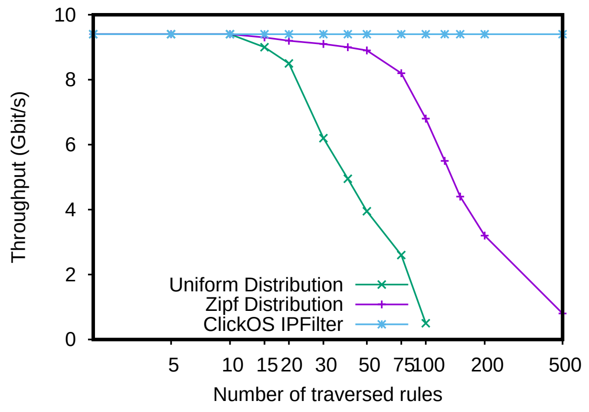


Figure 9: Maximum achievable throughput (PLR<0.1%) vs number of rules.

VLAN tagging on 5, 10, and 20 different flows using the uniform distribution. Overall, we observe that SplitBox can handle a constant number of packets/s, *regardless* of the packet size, which is crucial vis-à-vis our requirement to handle generic network functions. The number of packets per second degrades when the number of traversed rules is larger than 10, following the same proportion for all packet sizes. This implies that our system bottleneck does not reside, at 10Gbit/s, in the packet-handling function.

In order to further analyze the scalability of SplitBox, we studied the maximum achievable throughput as a function of the number of rules traversed. Figure 9 shows it, using packets of size 1500 bytes, vis-à-vis uniform and Zipf distribution (as explained above) and the ClickOS `IPFilter` elements. (Note that this element also uses Intel DPDK with FastClick.) Overall, as expected, the ClickOS element maintains line rate regardless of the number of rules. On the other hand, however, we observe that, with the uniform distribution, the line rate is achieved for rules between 2 and 15, the throughput starts to decrease quasi-linearly with the number of rules. The Zipf distribution behaves in a similar way, with the throughput decreasing with more rules following a similar pattern. However, in this case SplitBox is capable to sustain acceptable throughput (i.e. more than 8Gbit/s) for up to 100 traversed rules.

In Figure 9, we have identified the point in which SplitBox's throughput starts decreasing, leading to packet loss at very high speed. To further understand this issue, we measure the average and minimum latency when varying the number of traversed rules between 5 and 500, for both uniform and Zipf distributions – see Figure 10. We observe that the average latency increases with the number of rules, reaching a plateau for both distributions that corresponds to the time before we start losing packets either in (i) the synchronization ring buffer as an entry would be overwritten, or (ii) the Intel DPDK ring buffer on the `Processor`. This suggests that SplitBox's bottleneck either stems from numerous hash function evaluations, or potential cache misses in the shared data structure of the `Processor`, or both.

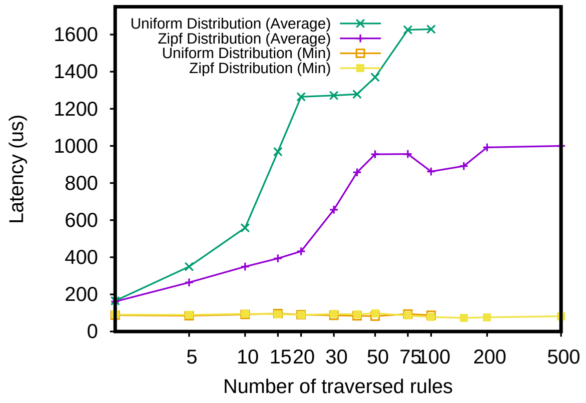


Figure 10: Latency vs number of rules at maximum achievable throughput (PLR<0.1%).

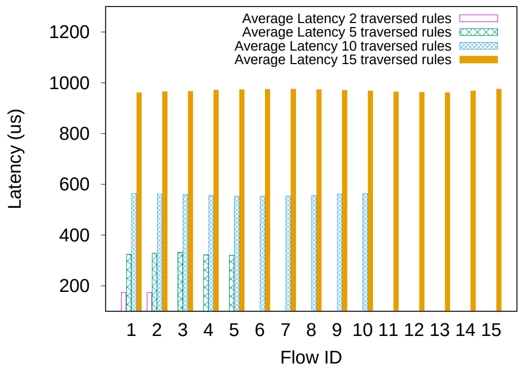


Figure 11: Average Latency per flow vs number of rules at maximum achievable throughput (PLR<0.1%).

In order to check if the hash function is contributing significantly to the latency, we present, in Figure 11, the average latency observed by each flow when increasing the number of rules from 2 to 15 in the uniform distribution scenario. Figure 11 shows that, on average, latency observed in all the flows is identical and monotonically increasing with the number of rules. This indicates that, with a relatively small number of rules, **hash function evaluations in `Processors` does not significantly impact latency**. Indeed, if hashing contributed significantly to the latency, in Figure 11, we would observe a difference in latency between the 2 rules and 15 rules setup, since the latter calls the hash function 14 more times for the flow number 15 compare to flow number 1.

We then turn to check whether cache access and potential cache misses are the reasons for the bottleneck in SplitBox. In Figure 12, we plot the number of hardware (RAM) cache references and misses observed per thread and per

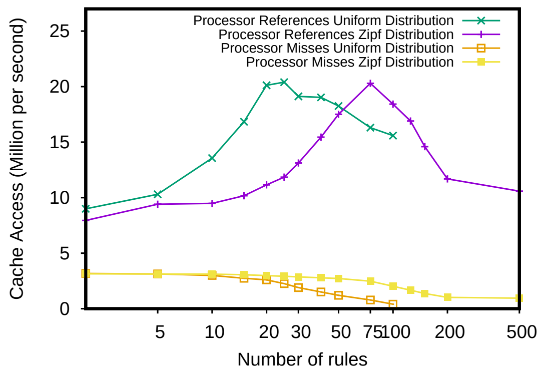


Figure 12: Cache reference vs number of rules at maximum achievable throughput (PLR<0.1%).

Table 2: Efficiency of `Processor`'s shared structure cache.

	5 traversed rules	10 traversed rules	20 traversed rules	50 traversed rules
Hashes hits	648	611	425	358
Action hits	58	69	54	95
Hashes misses	48 (6.9%)	54 (8.1%)	52 (10.9%)	49 (12%)
Action misses	0	0	0	0

second in `Processors`.[4] This coarse-grained plot presents both Intel DPDK cache references and our element cache references, highlighting that the number of memory references initially yields a linear increase in the number of rules for both flow distributions. The difference between the two curves can be explained as, in the Zipf distribution, the first rule is more likely to be matched and thus we do not reference, as much as in the uniform distribution, the Hashes structure (lookup table) from Table 1 since we are exiting the search loop after one iteration.

In Table 2, we present a sample of number of cache hits and misses for the `Processor` shared structures. We observe that, while the number of packets per second is more than halved going from 5 to 50 traversed rules, the percentage of cache misses increases from 6.9% to more than 12%. This increase in cache misses is explained by the size of the Hashes structure that is, as shown in Table 1, proportional to both the number of blinds and number of rules.

From these results we conclude that the performance bottleneck in SplitBox is mainly due to cache misses as the number of rules increases. We discuss different strategies to overcome this and other limitations and avenues for further

[4]Cache references and misses in `Entry` and `Client` are proportional to the number of packets per second, thus, we omit them.

performance optimization in SplitBox in the next section.

6. Discussion

Hardware limitations. As discussed earlier, our SplitBox prototype achieves near line rate throughputs under realistic loads. Performance starts decreasing when the number of traversed rules is larger than 50 for both VLAN tagging and the firewall scenarios, mainly due to the shared data structures management in the `Processor` element. We now discuss a few possible strategies to improve it. First, we observe that, since we do not perform any update on the data structures, increasing the number of cores and the amount of local core memory would have a significant impact on performances. Also, our experiments are run on relatively outdated CPUs – namely, Intel X3430, launched in late 2009 – with limited cache size (8MB shared cache size). Therefore, upgrading our testbed to state-of-the-art cloud technologies should result in maintaining line rate throughputs for larger number of traversed rules and/or smaller packets.

Next, better spatial locality of caching would also improve performance. Indeed, the number of blinds affects the size of all shared structures, and it is given to all elements as a function of the sequence number. To reduce this redundancy, one can extend the current Intel DPDK hashing function – Receive Side Scaling (RSS) – to always send the packet with the same blind number to the same core, thus reducing the size of all data structures by the number of cores. Unfortunately, this is not supported in the current version of the Intel DPDK framework since RSS does not give access to UDP payloads.

Increasing cloud MBs ($\mathcal{B}(t)$). Increasing the number of cloud MBs in $\mathcal{B}(t)$, i.e., increasing t, naturally strengthens the security of SplitBox. On the other hand, increasing t has an adverse impact on the packet loss ratio (PLR) to maintain a given level of throughput. The main performance bottleneck in terms of throughput at each MB in $\mathcal{B}(t)$ is table lookup. As shown previously, if the packet arrival rate is higher than table lookup speed, the receiving packet ring fills up quickly inducing end-to-end delay and driving the overall packet loss rate in the system. Indeed, in our performance evaluation we have considered the maximum achievable throughput while maintaining an overall successful delivery rate greater than 99.9%, i.e., PLR $< 0.1\%$. Let ρ be the probability of packet loss at a cloud MB $\mathcal{B}$ for a given throughput and number of traversed rules. Let $\Pr[\mathsf{no\ loss}]$ be the probability of not losing any packets from all the $\mathcal{B}$'s. Then, since the probabilities ρ are independent, we get $\Pr[\mathsf{no\ loss}] = \rho^t$. Thus, increasing the number of cloud MBs will result in proportional packet loss. This represents a tradeoff between increased security and reliability.

Real World Implications. Our experiments are conducted by simulating the execution environment of the real world in which the MBs (entry, cloud and client MBs) all reside on different compute nodes (hence the use of four commodity servers). Thus, the environment faithfully simulates execution in a real cloud environment, except for perhaps the additional latency expected in a setting where the compute nodes are relatively highly remote. However, note that this would also have a proportional performance impact in the traditional NFV setting (where no security is provided against an honest-but-curious adversary). As long as the latency between clouds A and B is reasonable, the overhead in a real setting beyond the results of our experiments would be minimal. Nevertheless, as we discuss in Section 8, in the future, we plan to test SplitBox in a more real-world cloud environment.

Removing lookup tables. We note that performance does not depend on the number of traversed rules per se, i.e., the number of different matching functions evaluated, but on the number of *different match projections* of these matching functions. For many middleboxes, the number of different match projections might be limited. With this observation, SplitBox could be substantially optimized. A special case of this is when there is a single match projection, e.g., a network function whose policies span the whole IP 5-tuple. In this case, we do not need any lookup tables! Omitting details, this is achieved by $\mathcal{A}$ only sending the hash of the relevant packet content to the $\mathcal{B}$'s, and each $\mathcal{B}_j$ doing a string match in the hashed domain to detect any matches. This can be modified slightly to provide indistinguishability of packet contents (cf. Section 3.3), without compromising efficiency.

Stateful network functions. A stateful network function is a function that processes packets based on its current state. The state itself can be modeled as dynamic policies that are generated and deleted as packets flow. An example is that of a stateful firewall, that keeps the state of a current TCP/IP connection. We discussed before that states in our network function model can be handled by simply adding policies (policy tree) on top of the network tree. These can then be removed once the state is deleted. However, to implement this in SplitBox, we require $\mathcal{C}$ to send l new hashes to each party in $\mathcal{B}(t)$, one for each blind s_i, and requiring them to update their lookup tables. While this is one solution, it is not optimal in terms of communication complexity. A somewhat different solution is to require $\mathcal{C}$ to maintain the state table at its end. This means that while the static policies are kept at the cloud, any dynamically generated state is maintained by $\mathcal{C}$. Notice that prior approaches [16] have also used this solution to maintain state tables. In either case, our solution is generic enough to handle state tables. It remains an open problem to handle the case where the cloud dynamically generates and maintains states without knowing the contents of the state and without involving any communication with $\mathcal{C}$, except may be at the setup.

Chaining. SplitBox does not straightforwardly allow network function chaining, if the next network function requires the packet modified by the previous network function as its input. This is because the parties $\mathcal{B}(t)$ only possess shares of the action, from which the modified packet can only be constructed by receiving the original packet and all shares. Obviously, for privacy reasons, this is done by $\mathcal{C}$ in our scheme. However, network function chaining can be done if we then let $\mathcal{C}$ forward the modified packet to party $\mathcal{A}$ of the next network function. In this way, the traffic loops through the client MB $\mathcal{A}$ until the last of the network functions has been applied.

7. Related Work

The first work to study privacy-preserving network functions outsourcing is, to the best of our knowledge, by Khakpour and Liu [12], who propose a scheme based on Bloom Filters (BFs) to privately outsource firewalling. Besides only considering one use case (that of a firewall), their solution is not provably secure as BFs are not *one-way*. Shi et al. [13] focus on the same problem, using CLT multilinear maps [26], which were however shown to be insecure [27]. Also note that both [12] and [13] do not consider network functions that modify packet contents, whereas, we aim to cover a broader range of network functions including but not limited to firewalls. Jagadeesan et al. [28] introduce a secure multi-controller architecture for SDNs based on secure multiparty computation, which could be employed for NFV. They focus on identifying heavy hitters in a network consisting of two controllers, however, it takes more than 13 minutes to execute with 4096 flow table entries. Melis et al. [15] investigate the feasibility of provably-secure private NFV for generic network functions: they introduce two constructions based on fully homomorphic encryption and public-key encryption with keyword search (PEKS) [29], but with high overhead (e.g., it takes at least 250ms in their experiments to process 10 firewall rules), which makes it unfeasible for real-world deployment. Recent proposals [30, 31] also rely on homomorphic encryption for privately outsourcing, respectively, firewall policies [31] and image transcoding [31], again achieving poor performances due to the expensive cryptographic primitives they employ. Somewhat related but limited solutions to private NFV rely on auditing [32] and verification of correctness [33] of outsourced functions.

Yuan et al. [34] support deep packet inspection over encrypted traffic, so that inspection rules or the payloads are not disclosed, relying on an encrypted rule filter and on secret sharing to enable secure inspection on the rules. They evaluate their solution on an Amazon Web Service instance with 500 concurrent connections and achieve a throughput of up to 3.6K packets/s per connection. Note that the system only considers general actions that do not modify the content of the packets.

Asghar et al. [1] introduce the main idea behind SplitBox, presenting a brief evaluation of a proof-of-concept implementation using a simple firewall as a test case. They achieve an average 2Gbps throughput, with 1 kB packets while traversing up to 60 firewall rules. Compared to [1], this paper presents a full-blown prototype implementation of SplitBox and a thorough system evaluation on commodity hardware using a Spirent traffic generator to create various types of traffic in order to stress-test the network functions. The experimental evaluation is done vis-à-vis two applications, namely, firewall and VLAN tagging, considering an uniformly distributed traffic flow distribution, which emulates the worst case scenario, as well as a Zipf distribution resembling a more realistic production firewall [25]. Overall, our prototype achieves the same throughput of the non-private solution with 9.4Gbps and 1.5kB packets when up to 10 rules are fired, and a decrease in performance by up to 5% for 50 rules. We also demonstrate that the latency introduced by SplitBox's prototype implementation, with large rule sets, is not due to the cryptographic layer, but to the data structures in the `Processor` element provided by the FastClick framework.

Blindbox [17] considers a setting in which a sender (S) and a receiver (R) communicate via HTTPS through a middlebox (MB) which has a set of rules for packet inspection that only it knows. The MB cannot decrypt traffic between S and R, while S and R do not learn the rules. Authors achieve 166Mbps throughput, however, the connection setup requires around 1.5 minutes, thus suggesting that BlindBox may not be practical for applications with short-lived connections. It also operates in a different setting than ours, where R sets and knows the rules (policies), while S and MB do not. Moreover, Blindbox only considers middlebox actions limited to drop, allow or report to network administrator, without defining action as modifying packet contents (e.g., for a NAT), while we do support modifying packet contents too. Recently, Canard et al. [35] introduces an extension of Blindbox [17] based on public-key encryption, which suffers from the same limitations of the original scheme.

Embark [16] allows a cloud provider to support middlebox outsourcing, such as firewalls and NATs, while maintaining confidentiality of an enterprise's network packets and policies. It employs the same architecture as APLOMB [5], where the middlebox functionalities (e.g. firewall) are outsourced without greatly damaging throughput, but traffic going through the service provider (SP) is encrypted in order to protect privacy. Embark relies on symmetric-key encryption and introduces a novel scheme, PrefixMatch, used to encrypt a set of rules for a middlebox type. The encrypted rules are generated by the enterprise(s) and then provided to the SP at setup time. The other scheme used in Embark is KeywordMatch adopted from [17]. Both the KeywordMatch and PrefixMatch methods require the client gateway to effectively insert the "encrypted" match for the cloud which in SplitBox is outsourced to the cloud MBs in $\mathcal{B}(t)$. The cloud middleboxes at SP then process the encrypted traffic against the encrypted rules, and send back the produced encrypted traffic to the enterprise, which performs the decryption. Thus, Embark does not satisfy our requirement of a thin client. Furthermore, Embark loops the traffic through the client which receives the traffic, encrypts it, sends to the cloud, who processes it (in the encrypted domain), and sends the modifications to the client, which then decrypts the packet. Providing privacy for such a system model is easier than the model considered in this paper, in which the traffic goes straight to the cloud. More specifically, since the client already knows the rules, it can effectively compute part of the matching functionality before sending the encrypted packet to the cloud. Another advantage of our scheme over Embark is that we provide a formal proof of security.

SICS [36] enables secure service function chain outsourcing. It extends APLOMB to support private processing of traffic through a sequence (chain) of middleboxes. Similar to Embark, SICS relies on AES encryption. However,

Table 3: Comparison of existing private NFV solutions

	Stateful	Thin Client	Line Rate	VM Isolation	Standard Hardware
Melis et al. [15]	✓	×	×	×	✓
Blindbox [17]	✓	×	✓	×	✓
SICS [36]	✓	×	✓	×	✓
Embark [16]	✓	×	✓	×	✓
SplitBox (our solution)	✓	✓	✓	×	✓
Safebricks [37]	✓	✓	✓	✓	×
LightBox [38]	✓	✓	✓	✓	×

actions applied to the packets are determined by labels attached to the packets. These labels are attached by the gateway (similar to our client MB), which also encrypts the packet header. Thus, once the encrypted packet along with the assigned label is sent to the cloud, the cloud MB readily knows which action to apply based on the label received.

Overall, our work differs from Embark [16] and SICS [36] as we allow complex actions to be performed on the packet directly without involving the client MB to aid the cloud MB. Thus, the traffic in our setting enters directly into the outsourced network function without looping through the client.

Finally, a parallel stream of work on secure network function outsourcing is the use of trusted computing environments in the cloud. This includes Safebricks [37] and LightBox [38] to name a few. As discussed in Section 1, this approach, which assumes that an adversary does not have access to the code or data in the protected "enclave" (trusted computing environment), is similar to ensuring VM isolation for secure network function outsourcing. Hence, it is orthogonal to the approach taken in this paper as it requires specialised hardware.

We present in Table 3 a summary of the comparison between our work and state-of-the-art privacy-preserving and secure NFV techniques. The comparison is based on whether the private NFV solution handles stateful network functions, the degree of client MB's involvement in real-time processing (thin client), achieving line rate, whether the solution is based on virtual machine isolation (as opposed to a cryptographic solution), or if the solution works with standard commodity servers (standard hardware).

8. Conclusion

This paper presented the design and implementation of SplitBox, a scalable system that allows a cloud service provider to privately compute network functions on behalf of a client, in such a way that the cloud does not learn the network policies. It provides strong security guarantees in the honest-but-curious model, based on cryptographic secret sharing. We performed a thorough system evaluation on commodity hardware, and created various types of traffic in order to stress-test firewall and VLAN tagging as network functions. Our evaluation shows that SplitBox achieves the same throughput of the non-private solution with 9.4Gbps and 1.5kB-sized packets when up to 10 rules are fired, and a decrease in performance limited to 5% (i.e., 8.9Gbps) with 50 rules.

In future work, we intend to deploy and evaluate SplitBox on more powerful, cloud-grade hardware, rather than on our commodity servers, re-evaluating its bottlenecks when operating on middleboxes equipped with more cores, cache, and/or memory. We also expect to improve performance by means of better management of shared data structures and better spatial locality of caching. Finally, we plan to extend SplitBox for: (i) full support of stateful network functions, (ii) function chaining, which at the moment could be done if we replace the client MB $\mathcal{C}$ with the entry MB $\mathcal{A}$ of another cloud, and (iii) handling network functions involving actions overwriting previous actions.

Bibliography

[1] H. J. Asghar, L. Melis, C. Soldani, E. De Cristofaro, M. A. Kaafar, and L. Mathy, "SplitBox: Toward Efficient Private Network Function Virtualization," in *ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2016.

[2] J. Sherry, S. Ratnasamy, and J. S. At, "A survey of enterprise middlebox deployments," Technical Report, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-24.pdf, 2012.

[3] European Telecommunications Standards Institute, "NFV Whitepaper," https://portal.etsi.org/nfv/nfv_white_paper.pdf.

[4] G. Gibb, H. Zeng, and N. McKeown, "Outsourcing Network Functionality," in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2012.

[5] 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 *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2012.

[6] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[7] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: enabling innovation in middlebox deployment," in *ACM Workshop on Hot Topics in Networks*, 2011.

[8] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the Art of Network Function Virtualization," in *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[9] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," *ACM SIGCOMM Computer Communication Review*, 2015.

[10] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags." in *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[11] Privacy Rights Clearinghouse, "Chronology of data breaches: Security breaches 2005–present," https://www.privacyrights.org/data-breaches, 2009.

[12] A. R. Khakpour and A. X. Liu, "First Step Toward Cloud-Based Firewalling," in *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2012.

[13] J. Shi, Y. Zhang, and S. Zhong, "Privacy-preserving Network Functionality Outsourcing," http://arxiv.org/abs/1502.00389, 2015.

[14] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "NoHype: Virtualized Cloud Infrastructure Without the Virtualization," in *International Symposium on Computer Architecture (ISCA)*, 2010.

[15] L. Melis, H. J. Asghar, E. De Cristofaro, and M. A. Kaafar, "Private Processing of Outsourced Network Functions: Feasibility and Constructions," in *ACM Workshop on SDN-NFV Security*, 2016.

[16] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, "Embark: Securely Outsourcing Middleboxes to the Cloud," in *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[17] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "BlindBox: Deep Packet Inspection over Encrypted Traffic," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[18] A. Shamir, "How to share a secret," *Communications of the ACM*, 1979.

[19] G. Oded, *Foundations of Cryptography. Basic Applications, vol. 2.* Cambridge University Press, New York, 2004.

[20] M. Goodrich and R. Tamassia, *Introduction to computer security.* Addison-Wesley, 2010.

[21] Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon, "Supercloud: Opportunities and challenges," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2723872.2723892

[22] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *ANCS*, 2015.

[23] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, 2000. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[24] Intel, "Intel Data Plane Development Kit," http://dpdk.org/.

[25] E. W. Fulp, "Optimization of network firewall policies using ordered sets and directed acyclical graphs," in *IEEE Internet Management Conference*, 2005.

[26] J.-S. Coron, T. Lepoint, and M. Tibouchi, "Practical Multilinear Maps over the Integers," in *IACR CRYPTO*, 2013.

[27] J. H. Cheon, K. Han, C. Lee, H. Ryu, and D. Stehle, "Cryptanalysis of the Multilinear Map over the Integers," in *IACR Eurocrypt*, 2015.

[28] N. A. Jagadeesan, R. Pal, K. Nadikuditi, Y. Huang, E. Shi, and M. Yu, "A Secure Computation Framework for SDNs," in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014.

[29] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *Eurocrypt*, 2004.

[30] H. Sheng, L. Wei, C. Zhang, and X. Zhang, "Privacy-preserving cloud-based firewall for iaas-based enterprise," in *Networking and Network Applications (NaNA)*, 2016.

[31] G. Biczók, B. Sonkoly, N. Bereczky, and C. Boyd, "Private vnfs for collaborative multi-operator service delivery: An architectural case," in *Network Operations and Management Symposium (NOMS)*, 2016.

[32] C. Chen, P. Maniatis, A. Perrig, A. Vasudevan, and V. Sekar, "Towards verifiable resource accounting for outsourced computation," in *ACM SIGPLAN Notices*, 2013.

[33] S. K. Fayazbakhsh, M. K. Reiter, and V. Sekar, "Verifiable Network Function Outsourcing: Requirements, Challenges, and Roadmap," in *ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2013.

[34] X. Yuan, X. Wang, J. Lin, and C. Wang, "Privacy-preserving deep packet inspection in outsourced middleboxes," in *IEEE INFOCOM*, 2016.

[35] S. Canard, A. Diop, N. Kheir, M. Paindavoine, and M. Sabt, "Blindids: Market-compliant and privacy-friendly intrusion detection system over encrypted traffic," in *Asia Conference on Computer and Communications Security*, 2017.

[36] H. Wang, X. Li, Y. Zhao, Y. Yu, H. Yang, and C. Qian, "SICS: Secure In-Cloud Service Function Chaining," *arXiv preprint arXiv:1606.07079*, 2016.

[37] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, “Safebricks: Shielding network functions in the cloud,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI’18), Renton, WA*, 2018.

[38] H. Duan, X. Yuan, and C. Wang, “Lightbox: Sgx-assisted secure network functions at near-native speed,” *arXiv preprint arXiv:1706.06261*, 2017.