SplitBox: Toward Efficient Private Network Function Virtualization

Hassan Jameel Asghar
Data61, CSIRO
hassan.asghar@data61.csiro.au

Luca Melis
University College London
luca.melis.14@ucl.ac.uk

Cyril Soldani
University of Liége
cyril.soldani@ulg.ac.be

Emiliano De Cristofaro
University College London
e.decristofaro@ucl.ac.uk

Mohamed Ali Kaafar
Data61, CSIRO
dali.kaafar@data61.csiro.au

Laurent Mathy
University of Liége
laurent.mathy@ulg.ac.be

ABSTRACT

This paper presents SplitBox, a scalable system for privately processing network functions that are outsourced as software processes to the cloud. Specifically, providers processing the network functions do not learn the network policies instructing how the functions are to be processed. We first propose an abstract model of a generic network function based on match-action pairs, assuming that this is processed in a distributed manner by multiple honest-but-curious providers. Then, we introduce our SplitBox system for private network function virtualization and present a proof-of-concept implementation on FastClick – an extension of the Click modular router – using a firewall as a use case. Our experimental results show that SplitBox achieves a throughput of over 2 Gbps with 1 kB-sized packets on average, traversing up to 60 firewall rules.

CCS Concepts

•Security and privacy → Security protocols; •Networks → Middleboxes / network appliances;

Keywords

Middlebox Privacy; Secret Sharing; Network Function Virtualization; Firewalls

1. INTRODUCTION

Network function virtualization (NFV) is increasingly being adopted by organizations worldwide, moving network functions traditionally implemented on hardware middleboxes (MBs) – e.g., firewalls, NAT, intrusion detection systems – to flexible and easier to maintain software processes. Network functions can thus be executed on virtual machines (VMs), with cloud providers processing traffic destined to, or originating from, an enterprise network (the client) based on a set of policies governing the network functions. This, however, implies that confidential information as well as sensitive network policies (e.g., the firewall rules) are revealed to the cloud, whereas in the traditional setting, such policies would only be known to the client's network administrators. Disclosing such policies can reveal sensitive details such as the IP addresses of hosts, the topology of the client's private network, and/or important practices [8, 13].

This motivates the need to allow processing outsourced network functions without revealing the policies: we denote this problem as *Private Network Function Virtualization* (PNFV), as done in [11]. We argue that PNFV solutions should not only provide strong *security* guarantees, but also satisfy *compatibility* with existing infrastructures (e.g., not requiring third parties, sending/receiving traffic, take part in new protocols) as well as *high throughput* in order to match the quality of service expected of network functions. In practice, this precludes the use of some standard cryptographic tools as well as other approaches which we review in Section 2.

Several attempts have recently been made to support PNFV or similar functionalities [8, 10, 11, 13], assuming the cloud to be honest-but-curious (i.e., the cloud processes the network functions as instructed but may try to learn the underlying policies). However, none of these simultaneously achieve security, compatibility, and high throughput, or their coverage of network functions is limited as they are only applicable to firewall rules that either allow or drop a packet.

Our intuition is to leverage the distributed nature of cloud VMs: rather than assuming that a single VM processes a client's network function, we distribute the functionality to several VMs residing on multiple clouds or multiple compute nodes in the same cloud. Assuming that not all VMs in the cloud are simultaneously under the control of the adversary (for instance, a *passive* attacker cannot gain access to all nodes running the distributed VMs), we are able to provide a scalable and secure solution. As discussed throughout the paper, achieving this solution is not straightforward and, in the process, we overcome several challenges.

We start by presenting an abstract definition of a network

HotMiddlebox'16 August 22-26, 2016, Florianopolis, Brazil

DOI: http://dx.doi.org/10.1145/2940147.2940150

function. Then, we introduce a novel system, which we name SplitBox, geared to privately and efficiently compute this abstract network function in such a way that the cloud, comprising of several middleboxes implemented as VMs, cannot learn the policies. Finally, we implement and evaluate SplitBox on a firewall test case, showing that it can achieve a throughput of over 2 Gbps with 1 kB-sized packets, on average, traversing up to 60 rules.

2. RELATED WORK

Khakpour and Liu [8] present a scheme based on Bloom Filters (BFs) to privately outsource firewalls. Besides only considering one use case, their solution is not provably secure as BFs are not *one-way*. Privately outsourcing firewalls is also considered by Shi et al. [13], who rely on CLT multilinear maps [5], which have been shown to be insecure [4]. Jagadeesan et al. [7] introduce a secure multi controller architecture for SDNs based on secure multi-party computation, which can potentially be employed for NFV. However, their implementationtakes more than 13 minutes to execute with 4096 flow table entries. Melis et al. [11] investigate the feasibility of provably-secure PNFV for generic network functions: they introduce two constructions based on fully homomorphic encryption and public-key encryption with keyword search (PEKS) [3], however, with high computational and communication overhead (e.g., it takes at least 250ms in their experiments to process 10 firewall rules) which makes it unfeasible for real-world deployment.

Blindbox [12] 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 should not be able to decrypt traffic between S and R, while S and R should not learn the rules. Although Blindbox achieves a 166Mbps throughput, it operates in a different setting than ours, in which R should set and know the rules (policies), while S and MB should not. Also, it only considers actions limited to drop, allow, or report, while we also consider modifying packet contents. Furthermore, the HTTPS connection setup requires around 1.5 minutes with thousands of rules, which suggests that BlindBox may not be practical for applications with short-lived connections.

Finally, Embark [10] enables a cloud provider to support middlebox outsourcing, such as firewalls and NATs, while maintaining confidentiality of an enterprise's network packets and policies. Specifically, it uses symmetric-key encryption to allow communication between enterprises and third-parties or enterprise-to-enterprise. A key difference between Embark and our solution is that we allow complex actions (besides allow/block) to be performed on the packet without revealing them to the cloud, e.g., NAT rules, while Embark can only do so in the clear.

3. PRELIMINARIES

System and Trust Model. Figure 1 illustrates our PNFV model, consisting of two types of cloud middleboxes (MBs): an *entry* MB $\mathcal{A}$ and $t \geq 2$ cloud MBs $\mathcal{B}(t)$, which collaboratively compute a network function on behalf of a client. The client has its own MB, denoted $\mathcal{C}$, at the edge of its internal network. $\mathcal{A}$ receives an incoming packet, does some computations 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.

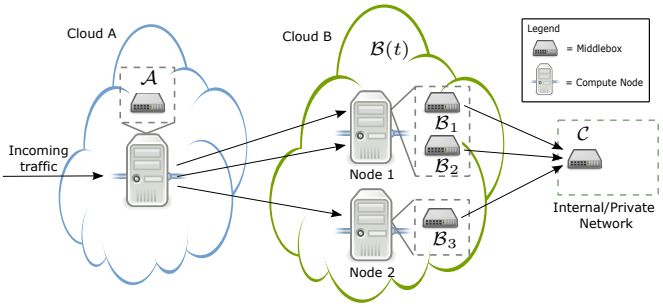


Figure 1: Our system model with Cloud A hosting MB $\mathcal{A}$ as a VM in one of its compute nodes. Cloud B hosts the MBs $\mathcal{B}(t)$ with $t = 3$ as VMs (not all t reside on the same compute node). The client MB $\mathcal{C}$ resides at the edge of the client's internal network. $\mathcal{A}$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

Assumptions. We assume an honest-but-curious adversary which can corrupt[1] either $\mathcal{A}$ or up to $t-1$ MBs from $\mathcal{B}(t)$, and it cannot corrupt $\mathcal{A}$ and any MB in $\mathcal{B}(t)$ simultaneously. In practice, one can assume $\mathcal{A}$ to be running on a different cloud provider than $\mathcal{B}(t)$ and that not all MBs in $\mathcal{B}(t)$ reside on the same node.

Network Functions. We define a packet x as a binary string of arbitrary length, i.e., $x \in \{0,1\}^*$. Our network functions will be applicable to the first n bits of x. A *matching* function is a boolean function $m : \{0,1\}^n \to \{0,1\}$. Its complement, i.e., the function $1-m$, is denoted by $\overline{m}$. An *action* function is a transformation $a : \{0,1\}^n \to \{0,1\}^n$. $m(x)$ (resp., $a(x)$) denote evaluating m (resp., a) on the substring $x(1,n)$ (i.e., the first n bits of x). If $|x| > n$, a keeps the part $x(n+1,*)$ of x unaltered. We also define the identity action function $I(x) = x$.

Let M and A be finite sets of matching and action functions, with $I \in A$. A *network* function $\psi = (M, A)$ is 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 node or a parent node. A parent node has two child nodes. The left child node is the identity action function I. The edge connecting the right child node is a matching function $m \in M$, whereas the edge connecting the left child node is its complement $\overline{m}$. The root node is the identity action function I. Clearly, there exists a binary relation from M to A, such that for each (m, a) from this relation there exists a parent node in ψ such that the left child is connected via the edge $\overline{m}$ and the right child via the edge m, and the right child is a.

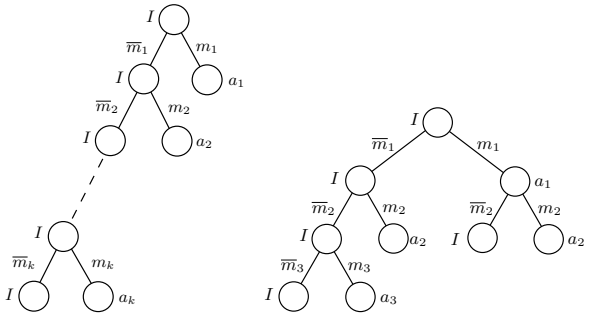
We call each pair (m, a) in ψ a *policy*. Policies serve

[1]The adversary may change the behavior of a MB from honest to honest-but-curious.

Algorithm 1: `Traversal`

Input: Packet x, network function ψ.

1 Make a read-only copy $x_{\mathtt{r}}$ and a writeable 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.



(a) Network function with k distinct policies.

(b) Network function with 3 distinct policies. Policy (m_2, a_2) is repeated twice.

Figure 2: Network functions as binary trees.

as building blocks of a network function. The set of policies of ψ is the set of *distinct* policies (m, a) in ψ. A network function is evaluated on input $x \in \{0,1\}^*$, denoted $\psi(x)$, using Algorithm 1. Figure 2(a) 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(b) has 3 distinct policies, $(m_1, a_1), (m_2, a_2)$ 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.

Coverage. Our abstract definition of network functions captures many network functions used in practice. These include firewalls, NAT and load balancers. Such functions usually perform a matching step to inspect some parts of a packet and modify contents of the packet subsequently. In the case of firewalls, modifications may also include dropping a packet.

Branching and chaining. Our definitions support branching, i.e., network functions that do not necessarily apply all policies on a packet. This is achieved by including multiple exit points, i.e., leaf nodes. Definitions also support *chaining*, e.g., ψ_1's output is ψ_2's input, however, in our proposed privacy-preserving solution chaining is not possible, since outputs of the MBs in $\mathcal{B}(t)$ need to be combined to reconstruct a transformed packet.

Policies. We restrict m to substring matching and a to be substring substitution. We also introduce the *don't care bit* denoted by $*$ in our alphabet. Given strings $x \in \{0,1\}^n$ and $y \in \{0,1,*\}^n$, we say $x = y$ if $x(i) = y(i)$ for all $i \in [n]$ such that $y(i) \neq *$. Given $x \in \{0,1\}^*$, 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. 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 $i \in [n]$. Given $x \in \{0,1\}^*$, the action function 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 function I, $\alpha = *^n$.

Definitions. Throughout the rest of the paper, we use the following definitions: let $z \in \{0,1,*\}^n$, 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 $z(i) = *$. The *masking* of a $x \in \{0,1\}^n$ using $\pi_z \in \{0,1\}^n$, denoted $\omega(\pi_z, x)$, returns x' s.t. $x'(i) = x(i)$ if $\pi_z(i) = 1$ and $x'(i) = 0$ if $\pi_z(i) = 0$. $\mathbb{H} : \{0,1\}^n \rightarrow \{0,1\}^q$ denotes a cryptographic hash function; $\oplus$ denotes bitwise XOR. The Hamming weight of a string $x \in \{0,1\}^n$ is $\mathrm{wt}(x)$. Finally, $x \leftarrow_{\$} \{0,1\}^n$ means sampling a binary string of length n uniformly at random.

4. INTRODUCING SPLITBOX

Privacy Requirements. We start by describing an *ideal* setting in which a trusted third party, $\mathcal{T}$, computes a network function ψ for the client. Upon receiving a packet x, $\mathcal{A}$ forwards it to $\mathcal{T}$, which provides the result of $\psi(x)$ to $\mathcal{C}$. Here $\mathcal{A}$ learns x but not $\psi(x)$ and $\mathcal{B}(t)$ neither x nor $\psi(x)$. In this section, we introduce our private NFV solution, SplitBox, aiming to simulate this ideal setting. However, we fall slightly short in that 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 is learnable from π_μ. Although this could reveal information such as which field of the packet the current matching function corresponds to, we do not consider it to be a strong limitation since this might be obvious from the type of NFV considered anyway. For example, if it is a firewall, then it is common knowledge that the fields it operates on will include IP address fields.

Design Aims. We consider the following design aims, i.e., the solution should: (a) be secure; (b) be computationally fast; (c) limit MB-to-MB communication complexity.

High-Level Overview. In a nutshell, if we assume that ψ includes a single policy (m, a), our strategy 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 $\mathcal{B}(t)$; whereas, 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 $\mathcal{B}_j$. In addition, $\mathcal{A}$ encrypts the contents of a packet x by XORing it with the blind s, and sends it to the MBs in $\mathcal{B}(t)$, which can then compute matches and actions on this encrypted packet. We present the details of SplitBox using a set of algorithms, grouped based on the MB executing them. Figure 3 shows a high-level overview of all the algorithms computed by each

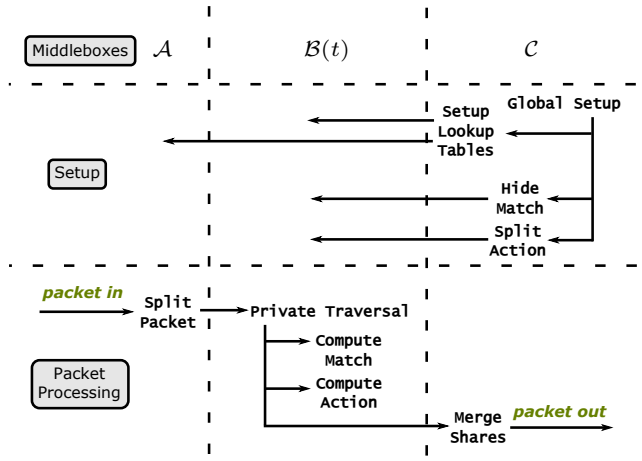


Figure 3: Breakdown of algorithms executed by each MB in SplitBox.

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

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

MB. We assume ψ_{priv} to be the private version of the network function 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 by XORing. For each blind $s \in S$ and for each $m \in M$, 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 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 we have modelled dropping a packet as setting $x(1, n)$ to 0^n.

Middlebox $\mathcal{A}$. This MB only runs Algorithm 7, which

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

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

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

Input: Matching function $m \in M$ with match μ.
Send π_μ to $\mathcal{B}(t)$.

maintains a counter initially set to 0 and incremented every time a new packet x arrives. The value of the counter corresponds to a blind in the lookup table S. Therefore 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 $\mathcal{C}$.

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 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 $x_{\mathtt{r}}$.

5. ANALYSIS

Correctness. Given $\psi = (M, A)$, for a matching function $m \in M$, as long as m can be represented as substring matching, SplitBox correctly computes the match. That is, if m is an equality test or range test for powers of 2 in binary (e.g., IP addresses in the range $\mathtt{127.*.*.32}$ to $\mathtt{127.*.*.64}$), then it can be successfully computed by SplitBox. Our model also allows for arbitrary ranges by dividing m into smaller matches that check equality matching of individual bits. However, such a representation can potentially make ψ very large. We can correctly compute action functions as

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

Input: Action function $a \in A$ with action α.

Sample $\alpha_1, \alpha_2, \ldots, \alpha_{t-1} \leftarrow_{\$} \{0,1\}^n$.
Let $\tilde{\alpha} = \omega(\pi_\alpha, \alpha)$. Compute $\alpha_t = \tilde{\alpha} \oplus \alpha_1 \oplus \cdots \oplus \alpha_{t-1}$.
Sample $\beta_1, \beta_2, \ldots, \beta_{t-1} \leftarrow_{\$} \{0,1\}^n$.
Compute $\beta_t = \pi_\alpha \oplus \beta_1 \oplus \cdots \oplus \beta_{t-1}$.
for $j = 1$ **to** t **do**
    Give α_j, β_j to $\mathcal{B}_j$.

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

Input: Index i, packet copy $x_{\mathtt{w}}$, α'_j and β'_j from $\mathcal{B}_j$ for $j \in [t]$.

Compute $\alpha' \leftarrow \alpha'_1 \oplus \cdots \oplus \alpha'_t$.
Compute $\beta' \leftarrow \beta'_1 \oplus \cdots \oplus \beta'_t$.
Compute $x \leftarrow x_{\mathtt{w}} \oplus s_i$, where $s_i \in S$.
for $i = 1$ **to** n **do**
    if $\beta'(i) = 1$ **then**
        $x(i) \leftarrow \alpha'(i)$
if $x(1,n) = 0^n$ **then**
    Drop x.
else
    Forward x.

long as they satisfy two properties: (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 overlap on their non-zero bits. Note that this does not restrict the number of times the identity function I can be applied, as its action projection is 0^n.

Security. While we refer to the full version of the paper for the security proofs [1], here we mention two important points: if SplitBox is used for match projections whose Hamming weight is low, then the $\mathcal{B}(t)$ can brute-force $\mathbb{H}$ to find its pre-image. This reveals $\mu \oplus s$ for some blind s, which allows the adversary to learn more than simply looking at the output of m. Namely, if $m(x) = 0$, the adversary learns which relevant bits of an incoming packet x do not match with the stored match. The second point relates to the length of the look-up table l: ideally l should 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. Therefore, we propose the following mitigation 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. Thus, any MB in $\mathcal{B}(t)$ does not know if the two packets corresponding to the same blind are two actual packets (the probability is ρ^2). The downside is that this reduces the (effective) throughput by a factor of ρ. $\mathcal{A}$ can indicate to $\mathcal{C}$ if the current packet is a dummy packet by sending a bit through $\mathcal{B}(t)$ to $\mathcal{C}$ using a t-out-of-t secret sharing scheme (XORing with random bits).

6. IMPLEMENTATION

In this section, we discuss our proof-of-concept implementation of SplitBox inside FastClick [2], an extension of the Click modular router [9] which 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 [6] as the underlying packet I/O framework. We implemented three main FastClick elements: element `Entry` corresponding to MB $\mathcal{A}$, `Processor` corresponding to MBs $\mathcal{B}$, and `Client` to $\mathcal{C}$. `Client` implements the `Merge Shares` algorithm. The other algorithms of $\mathcal{C}$ are executed outside the FastClick elements, and used to configure the above three elements. The hash function $\mathbb{H}$ is implemented using OpenSSL's SHA-1, aiming to achieve a compromise between security, digest length, and computation speed, as hash functions which have larger message digests will lead to overly large lookup tables. `Client` uses a circular buffer to collect packet shares until all have been received and the final packet can be reconstructed. For communication between our elements, we use UDP packets: UDP and L2 processing relies on standard Click elements such as `UDPIPEncap`. Finally, we also add a few elements to help in our delay measurements, as explained below.

To evaluate our implementation, we focus on a firewall use case, using a network function tree similar to that in Figure 2(a). A single action is applied, either the identity action, if the packet is allowed, or marking the packet with a drop message (0^n), if it should be dropped. We use three commodity PCs for our experiments (8-core Intel Xeon E5-2630 with 2.4GHz CPU and 16 GB of RAM): one for both `Entry` and `Client`, in order to use the same clock for delay measurements, and the other two as two `Processors`. The four nodes (including the two on the same machine) are connected through Intel X520 NICs, with 10-Gbps SFP+ ca-

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

Input: Packet x, lookup table S.

Get the index $i \in [l]$ corresponding to the current value of the counter.
Let $x_{\mathtt{w}} \leftarrow x \oplus s_i$ (writeable copy), where $s_i \in S$.
Compute $x_{\mathtt{r}} \leftarrow x(1,n) \oplus s_i$ (read-only copy), where $s_i \in S$.
for $j = 1$ **to** t **do**
    Send $x_{\mathtt{r}}$, i to $\mathcal{B}_j$.
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 ψ_{priv}.

Initialize empty strings $\alpha'_j \leftarrow 0^n$ and $\beta'_j \leftarrow 0^n$.
Start from the root node.
Update α'_j and β'_j by running the `Compute Action` algorithm on the current node a.
if *the current node is a leaf node* **then**
    Send i, α'_j and β'_j to party $\mathcal{C}$ and stop.
else
    Run `Compute Match` algorithm on i, m and $x_{\mathtt{r}}$, where m is the right hand side edge.
    if `Compute Match` *outputs 1* **then**
        Go to the right child node.
    else
        Go to the left child node.
Go to step 3.

Algorithm 9: `Compute Action` $(\mathcal{B}(t))$

Input: Pair of cumulative action and cumulative action projection shares (α'_j, β'_j) of $\mathcal{B}_j$, pair of action and action projection shares (α_j, β_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 α'_j, β'_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 μ_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.

bles. The topology is thus very similar to the one in Figure 1, except that we only have $t = 2$ in $\mathcal{B}(t)$, and that $\mathcal{A}$ and $\mathcal{C}$ share the same physical machine. Another difference is that our machines are connected directly, without intermediate routers between them. We use a trace captured at one of our campus border router (pre-loaded into memory) as input for the `Entry` element, which executes the `Split Packet` algorithm on a single core. Then, each output of `Entry` (one for $\mathcal{C}$ and one per $\mathcal{B}_j$) is encapsulated inside an UDP packet and sent to the corresponding output device, using one core per device.

On each $\mathcal{B}_j$ machine, the packets are read from the input device, decapsulated, and then passed to a `Processor` element which does the actual filtering. The resulting action packets are then re-encapsulated and sent through the NIC towards the client. This operation is done on a single core, but several cores can easily be used in parallel.With FastClick, it suffices to launch Click with more cores, and the system will automatically create the corresponding number of hardware queues on the NICS, and assign a core to each queue. On the client side, each of the three input NICs has an associated core. Incoming packets are decapsulated, and then passed to the `Client` element, which reconstructs the final packets (on its own core). Reconstructed packets which are not marked as dropped are then passed to a receiver pipeline, which computes the entry-to-exit delay, counts packets and measures reception bandwidth. To measure delays, the packets in the in-memory list are tagged with a sequence number in the packet payload, before the transmission begins. This number allows to match the exit timestamp with an entry timestamp, which is kept in memory. This allows to avoid storing the timestamp itself in the packet, which would increase the delay measured.

The SplitBox setup is compared against a simpler setup using an `IPFilter` element on a single machine, to act as a non-private outsourced firewall with the same rules. The

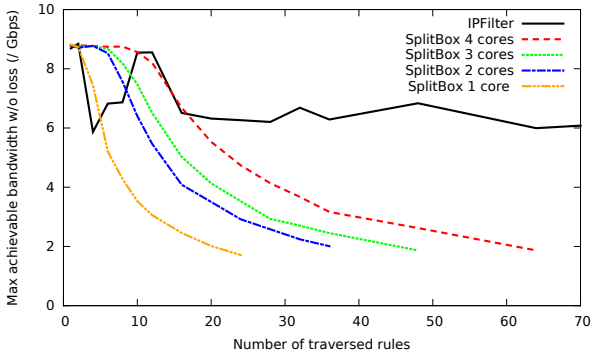


Figure 4: Achievable bandwidth drops sharply with the number of traversed rules.

`IPFilter` element sends only the non-dropped packets (without encapsulation) directly to an output device, which is connected to an input device feeding the receiver pipeline.

7. PERFORMANCE EVALUATION

We now present the results of the experiment described above, with various input bit-rates and different number of rules, while measuring loss rate and delays. While we have to forward all packets to the client, a non-private outsourced firewall can drop the rejected packets immediately. Thus, its achievable bit-rate will depend on a combination of the input traffic and the ruleset. To normalize results in our analysis, we craft rulesets such that all packets are accepted. While it changes nothing for SplitBox, it is a worst-case for the `IPFilter`-based testcase. At the same time, we tightly control the number of match attempts per packet, in order to evaluate the impact of the average number of rules traversed by a packet before it matches.

Figure 4 illustrates the evolution of the maximum achievable bandwidth (taken as inducing less than 0.001% losses), as a function of the number of traversed rules (i.e., the number of match attempts per packet). Our trace packets are about 1 kB on average, so that 8 Gbps corresponds to about 1 Mpps. We observe that the bandwidth decreases significantly with more traversed rules with SplitBox (PNFV), mainly due to the hashing function, which is called on the packet header once per match attempt. Not only is this more computationally expensive than simpler comparisons, but it is also done each time on different data (as we need to first XOR packet header with match projection), taking no advantage of the cache. Fortunately, the `Processor` operation is inherently parallelizable, thus, allocating more cores speeds things up. Note that the average number of traversed rules in a real firewall is significantly lower than the total number of rules. Therefore, it is particularly important to choose the order of match attempts according to the traffic distribution, and/or to use a more complex tree structure minimizing the number of match attempts.

Finally, in Figure 5, we plot the delays as a function of firewall load (i.e., current input bandwidth over maximum achievable bandwidth). Note that the delays do not follow

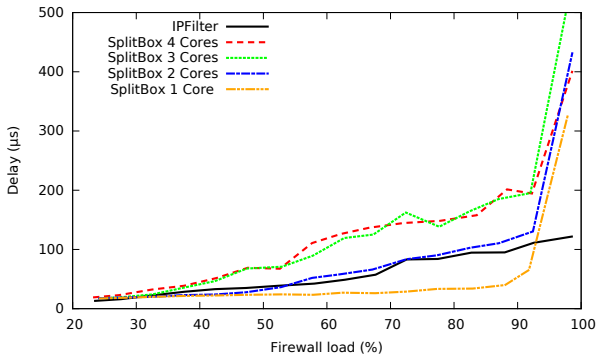


Figure 5: Delay increases with the firewall load.

the same dependency w.r.t. the number of match attempts per packet. Although these increase slightly with the number of traversed rules, they are mostly governed by queuing delays in the system (in NICs rings, or in-memory rings exchanging packets between the different processing cores). The number of blinds l seems to have little impact on the performance: with l ranging from 64 to 65,536, we observe no noticeable difference, except for additional memory consumption.

In conclusion, our SplitBox proof-of-concept implementation for a firewall use case achieves comparable performance to a non-private version, providing acceptable throughput and delays for small rulesets. Larger rulesets should be carefully laid out in order to minimize the number of match attempts per packet.

8. CONCLUSION & FUTURE WORK

This paper presented SplitBox, a novel 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 experiment with our implementation using firewall as a test case, and achieve a throughput in the order of 2 Gbps, with packets of average size 1 kB traversing about 60 firewall rules. In future work, we plan to consider more diverse types of matches (that allow matching on arbitrary ranges) and actions (that allow overlapping non-zero bits), as well as using k-out-of-t secret sharing schemes rather than t-out-of-t.

9. REFERENCES

[1] H. J. Asghar, L. Melis, C. Soldani, E. De Cristofaro, M. A. Kaafar, and L. Mathy. SplitBox: Toward Efficient Private Network Function Virtualization (Full Version). http://arxiv.org/abs/1605.03772, 2016.

[2] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *ANCS*, 2015.

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

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

[5] J.-S. Coron, T. Lepoint, and M. Tibouchi. Practical Multilinear Maps over the Integers. In *CRYPTO*, 2013.

[6] Intel. Intel Data Plane Development Kit. http://dpdk.org/.

[7] N. A. Jagadeesan, R. Pal, K. Nadikuditi, Y. Huang, E. Shi, and M. Yu. A Secure Computation Framework for SDNs. In *HotSDN '14*, 2014.

[8] A. R. Khakpour and A. X. Liu. First Step Toward Cloud-Based Firewalling. In *SRDS*, 2012.

[9] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3), 2000.

[10] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In *NSDI*, 2016.

[11] 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.

[12] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *SIGCOMM*, 2015.

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