SplitBox: Toward Efficient Private Network Function Virtualization

Hassan Jameel Asghar[1], Luca Melis[2], Cyril Soldani[3],
Emiliano De Cristofaro[2], Mohamed Ali Kaafar[1], Laurent Mathy[3]
[1]Data61, CSIRO, Australia [2]University College London, UK [3]University of Liége, Belgium

ABSTRACT

This paper presents SplitBox, an efficient system for privacy-preserving processing of network functions that are outsourced as software processes to the cloud. Specifically, cloud providers processing the network functions do not learn the network policies instructing how the functions are to be processed. First, we propose an abstract model of a generic network function based on match-action pairs. We assume that this function is processed in a distributed manner by multiple honest-but-curious cloud service 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 achieve a throughput of over 2 Gbps with 1 kB-sized packets on average, traversing up to 60 firewall rules.

1. INTRODUCTION

Network function virtualization (NFV) is increasingly being adopted, aiming to move 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., firewall rules) are revealed to the cloud, whereas in the traditional setting, such policies would only be known to the client's network administrators.

This motivates the need to let the cloud process network functions on behalf of a client without revealing the policies, a problem denoted as *Private Network Function Virtualization* (PNFV) [11]. Naturally, 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: several attempts have recently been made to instantiate PNFV [7,9,11,14] (reviewed in Section 2), however, we argue that 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 allow either allow or drop.

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 the cloud cannot corrupt all VMs simultaneously (for instance, the attacker cannot gain access to all compute nodes running the distributed VMs), we provide a scalable and secure solution. However, as discussed throughout the paper, this solution is not straightforward and, in the process, we overcome several challenges.

We start by presenting a mathematical definition of an abstract network function, then, we introduce a secure system, called SplitBox, geared to privately compute this abstract network function in such a way that the cloud, comprising of several middleboxes implemented as VMs, cannot learn the policies under certain trust assumptions. 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 [7] 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*. Furthermore BFs inevitably introduce false positives, i.e., packets might accidentally be matched against a firewall rule. Privately outsourcing firewalls is also considered by Shi et al. [14], who rely on CLT multilinear maps [4], which have been shown to be insecure [3]. More specifically, the `isZero` routine of CLT maps, used in [14] to check whether a packet matches a policy, is not secure. Additionally note that both [7, 14] 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. [6] introduce a secure multi controller architecture for SDNs based on secure multi-party computation, which can potentially be employed for NFV. They provide a proof of concept implementation for 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. [11] recently 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) [2], 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 [13] 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. 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. Lin et al. [10] also propose a privacy-preserving deep packet filtering technique (DPF-ET) where the packet data is hidden from the network owner and the users do not learn the filtering rules. Compared to BlindBox, DPF-ET significantly reduces the setup overhead and requires a filtering time for each packet of $5\mu s$ for matching a thousand rules with a 32-bit rule length. Both Blindbox and DPF-ET only consider middlebox actions that are limited to drop, allow or report to network administrator, without defining action as modifying packet contents (e.g., for a NAT) as is done in our paper.

Finally, Embark [9] enables a cloud provider to support middlebox outsourcing, such as firewalls and NATs, while maintaining confidentiality of an enterprise's network packets and policies. Embark employs the same architecture as APLOMB [12], where the middlebox functionalities (e.g. firewall) are outsourced to the cloud by the enterprises without greatly damaging throughput, but it encrypts the traffic going to the service provider (SP) in order to protect privacy. To this end, 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 cloud middleboxes at SP then process the encrypted traffic against the encrypted rules, and send back the produced encrypted traffic to the enterprise who, finally, performs the decryption. When compared to Blindbox, Embark achieves better performance, as it does not require per-user-connection overhead, and broader functionality. A key difference between Embark and our solution is that we allow complex actions (on top of allow/block) to be performed on the packet without revealing them to the cloud, e.g., NAT rules. Embark can only do so in the clear.

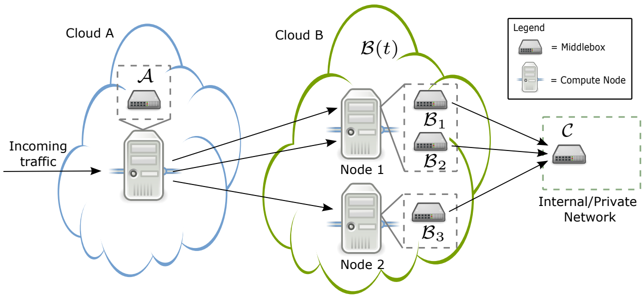


Figure 1: Our system model with Cloud A hosting MB $\mathcal{A}$ as a VM in one of its compute nodes. Cloud B hosts the MBs $\mathcal{B}(t)$ with $t = 3$ as VMs (not all t reside on the same compute node). The client MB $\mathcal{C}$ resides at the edge of the client's internal network. $\mathcal{A}$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

3. PRELIMINARIES

3.1 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 to $\mathcal{C}$, which reconstructs the network function's final result. There is also a direct link between $\mathcal{A}$ and $\mathcal{C}$.

Assumptions. We assume an honest-but-curious adversary which can corrupt 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 $\mathcal{B}(t)$ are on the same node. Since $\mathcal{C}$ is the client's MB, we do not assume it to be adversarial.

3.2 Network Functions

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

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

Algorithm 1: `Traversal`

```
   Input: Packet $x$, network function $\psi$.
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.
```

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. Examples of network functions are in Figure 2. 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*. 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. A network function is evaluated on input $x \in \{0,1\}^*$, denoted $\psi(x)$, using Algorithm 1. Note that the reason to create a separate writeable copy $x_{\mathtt{w}}$ of x is to ensure that the matching functions are applied on the "uncorrupted" 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, we say that the network function has terminated. 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, $(m_1, a_1), (m_2, a_2)$ 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.

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. For chaining to work, network function ψ_2 needs to know the output of network function ψ_1. However, if ψ_2 only needs the original input x, instead of the overwritten copy $x_{\mathtt{w}}$, network function 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.

3.3 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 $y(i) \neq *$. In other words, if the two strings match at every position except for the don't care positions we consider the two strings to be equal. Given $x \in \{0,1\}^*$, matching function m is defined as

$$m(x) = \begin{cases} 1, & \text{if } x(1,n) = \mu \\ 0, & \text{otherwise} \end{cases}, \tag{1}$$

where $\mu \in \{0,1,*\}^n$. We call μ the *match* of m.

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 $i \in [n]$. Given $x \in \{0,1\}^*$, the action function a is defined as

$$a(x) = x(1,n) \leftarrow \alpha, \tag{2}$$

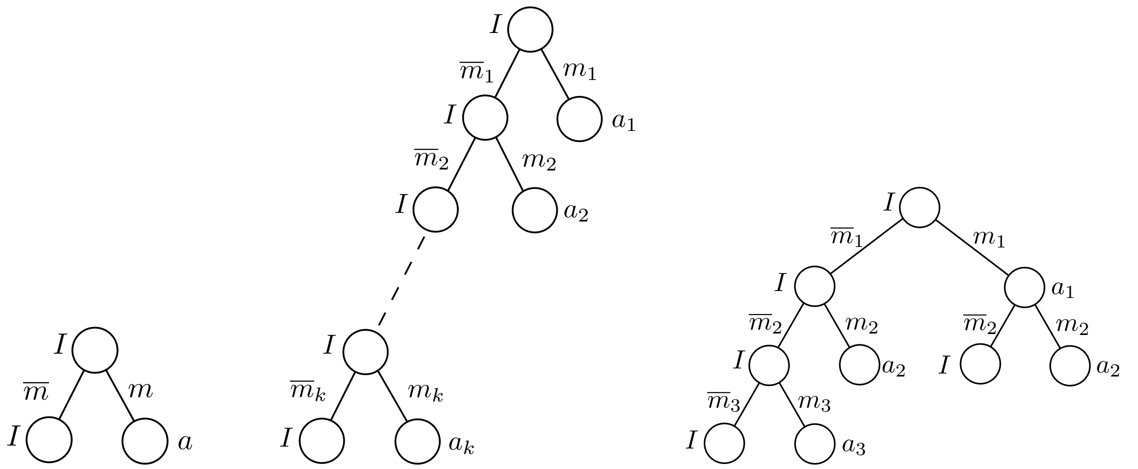
where $\alpha \in \{0,1,*\}^n$. We call α the *action* of a. With this definition, the identity action function I is $I(x) = x(1,n) \leftarrow \alpha$, where $\alpha = *^n$.

Definitions. Throughout the rest of the paper, we use the following definitions: let $z \in \{0,1,*\}^n$, the *projection* of z (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 $\pi_z(i) = 0$. Although we have broadly defined $\omega(\pi_z, x)$ for an $x \in \{0,1,*\}^n$, we use it exclusively for an $x \in \{0,1\}^n$. $\mathbb{H} : \{0,1\}^n \to \{0,1\}^q$ 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

4.1 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



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

(b) Network function with k distinct policies.

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

Figure 2: Network functions as binary trees.

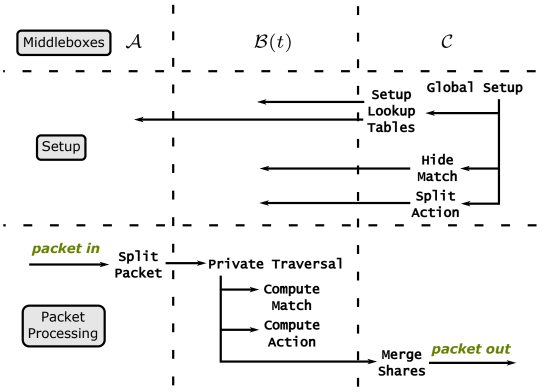


Figure 3: Breakdown of algorithms executed by each MB in SplitBox.

$m \in M$ 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, e.g., if it is a firewall, then it is common knowledge that the fields it operates on will include IP address fields.

4.2 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 introduces a high-level overview of all the algorithms computed by each 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 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

```
Algorithm 2: Global Setup ($\mathcal{C}$)
Input: Parameters $n$ and $l$, network function $\psi = (M, A)$.
for $j = 1$ to $t$ do
    Send $\psi_{\text{priv}}$ 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.
```

```
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 $\mu_j$ 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 $\mathcal{B}(t)$.
```

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

5.1 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 127.∗.∗.32 to 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 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 actions α_i and α_j do not overlap on their non-zero bits.

```
Algorithm 4: Hide Match ($\mathcal{C}$)
Input: Matching function $m \in M$ with match $\mu$.
Send $\pi_\mu$ to $\mathcal{B}(t)$.
```

```
Algorithm 5: Split Action ($\mathcal{C}$)
Input: Action function $a \in A$ with action $\alpha$.
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 $\alpha_j, \beta_j$ to $\mathcal{B}_j$.
```

5.2 Security

The proof of security of our construction is shown in Appendix A. 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. This is the reason why we use the hash function $\mathbb{H}$. It does not allow $\mathcal{B}(t)$ to learn more than the output of m. 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 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 the result corresponds to two actual packets (the probability is ρ^2) or not. The downside is that this reduces the (effective) throughput by a factor of ρ. Nevertheless, with this strategy we can use a feasible value of l. Of course $\mathcal{A}$ has to indicate to $\mathcal{C}$ which packet is a dummy packet. This can be done by sending a bit through $\mathcal{B}(t)$ to $\mathcal{C}$ by once again 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 [1], an extension of the Click Modular Router [8] which provides fast user-

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

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

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 $\mathcal{C}$.
```

space packet I/O and easy configuration via automatic handling of multi-threading and multiple hardware queues. We also use Intel DPDK [5] 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. An element `Entry` corresponds to MB $\mathcal{A}$. This element is responsible for the `Split Packet` algorithm. An element `Processor` is used for the $\mathcal{B}_j$ MBs. It is responsible for the `Private Traversal`, `Compute Match` and `Compute Action` algorithms. Finally, an `Client` element corresponds to $\mathcal{C}$ and is responsible for 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. 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 (as discussed in Section 7, the hashing speed is an important factor of the performance of our solution). `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(b). 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+ cables. 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.

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.
```

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 $\mathcal{B}_j$.
Compute $\alpha'_j \leftarrow \alpha'_j \oplus \alpha_j$.
Compute $\beta'_j \leftarrow \beta'_j \oplus \beta_j$.
Output $\alpha'_j$, $\beta'_j$.
```

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 re-

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 $\mu_j$.
Lookup table $\tilde{S}$ at index $(i, j)$ to obtain $\mathbb{H}(\tilde{s}_{i,j})$.
Extract $\tilde{x}_{\mathtt{r}} \leftarrow \omega(\pi_{\mu_j}, x_{\mathtt{r}})$.
Compute $\mathbb{H}(\tilde{x}_{\mathtt{r}})$.
if $\mathbb{H}(\tilde{x}_{\mathtt{r}}) = \mathbb{H}(\mu_j \oplus \tilde{s}_{i,j})$ then        // $m(x) = 1$
    Output 1.
else                                                                   // $m(x) = 0$
    Output 0.
```

constructs 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. To store the sequence number, we need to extend very small packets (e.g. TCP ACKs). We prefer that to not accounting for small packets in our delay measurements.

The SplitBox setup is compared against a simpler setup using a `IPFilter` element, with the same filtering rules, to act as a non-private firewall. In that configuration, a single machine is used. The `IPFilter` element replaces the `Entry` element and 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. `IPFilter` is also sensitive to the number of match attempts, but much less so thanks to cheaper comparisons on a hot 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.

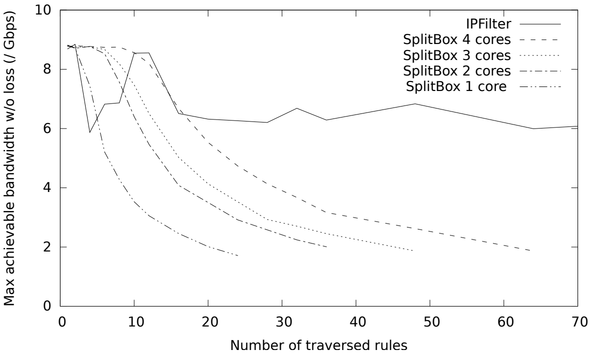


Figure 4: Achievable bandwidth drops sharply with the number of traversed rules.

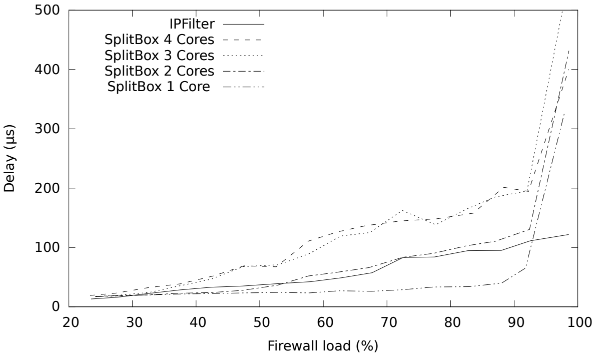


Figure 5: Delay increases with the firewall load.

Finally, in Figure 5, we plot the delays as a function of firewall load (i.e., current input bandwidth over maximum achievable bandwidth). Surprisingly, the delays do not follow 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 the NICs rings, or in 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 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. Our implementation using firewall as a test case achieves throughput a in the order of 2 Gbps, with packets of average size 1 kB traversing about 60 firewall rules. Our work can be improved in several ways, such as considering more diverse types of matches (that allow matching on arbitrary ranges) and actions (that allow overlapping non-zero bits), or by introducing k-out-of-t secret sharing schemes rather than t-out-of-t.

9. REFERENCES

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

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

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

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

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

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

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

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

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

[10] Y.-H. Lin, S.-H. Shen, M.-H. Yang, D.-N. Yang, and W. T. Chen. Privacy-Preserving Deep Packet Filtering over Encrypted Traffic in Software-Defined Networks. In *ICC '16*.

[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, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM*, 2012.

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

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

APPENDIX

A. SECURITY PROOFS

We assume a passive (honest-but-curious) adversary $\mathcal{E}$ which can either corrupt $\mathcal{A}$, or up to $t-1$ parties (MBs) from $\mathcal{B}(t)$.[1] Let Π denote our PNFV scheme (SplitBox). 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 of matching functions. Note that 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 party $\mathcal{B}_j \in \mathcal{B}(t)$ 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 learn-able 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 $\mathrm{wt}(\pi_\mu) = 1$). That is the reason for using the hash function $\mathbb{H}$ in the scheme.
- Any coalition of $t-1$ parties 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 I and O denoting the input and output of a party (or a subset of parties) corrupted by $\mathcal{E}$.[2] 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.

IDEAL$(\psi, \mathcal{S})$. We assume a trusted third party $\mathcal{T}$, which communicates with each of the parties via a secure and

[1] We will use the word 'party' instead of middlebox or MB.

[2] Abusing notation, in this section, we use the same symbol I for input that was previously reserved for the identity action function.

private link. $\mathcal{T}$ is given the network function $\psi = (M, A)$. Parties $\mathcal{B}(t)$ 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. Party $\mathcal{A}$ 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, $\text{IDEAL}(\psi, \mathcal{S})$ is the tuple $(I, O, X, \psi(X), D)$, where the random variables correspond to the party (or subset of parties) controlled by $\mathcal{S}$.

Our real setting, denoted $\text{REAL}(\Pi, \mathcal{E})$ is simply the execution of our scheme in the presence of the adversary $\mathcal{E}$. It again represents the tuple $(I, O, X, \psi(X), D)$ where each random variable corresponds to the party (or subset of parties) corrupted by $\mathcal{E}$. Naturally, depending on whether $\mathcal{E}$ corrupts party $\mathcal{A}$ or upto $t-1$ parties 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 $(I, O, X, \psi(X), D)$). 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 $\approx_c$ 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 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 PNFV solution.

Our main results are as follows.

THEOREM 1. *The PNFV 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 $I = X$ (which holds both in the ideal and real setting). The output 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 5.2, 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 Section 5.2 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 PNFV scheme Π privately processes ψ against an honest-but-curious $\mathcal{E} \subset \mathcal{B}(t)$ in the random oracle model.*

PROOF. Let $\mathcal{R} : \{0,1\}^* \rightarrow \{0,1\}^q$ 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 party 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 player 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 party in $\mathcal{E}$, together with the current counter value i.

For any oracle query from a party $\mathcal{B}_j \in \mathcal{E}$, $\mathcal{S}$ 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 $(I, O, X, \psi(X), D)$ for each party 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 $\text{REAL}(\Pi, \mathcal{E}) \approx_{\text{c}} \text{IDEAL}(\psi, \mathcal{S})$. □