SICS: Secure and Dynamic Middlebox Outsourcing

Huazhe Wang, Xin Li, Yang Wang, *Member, IEEE*, Yu Zhao, *Student Member, IEEE*
Ye Yu, Hongkun Yang, Chen Qian, *Senior Member, IEEE*,

***Abstract*—There is an increasing trend that enterprises outsource their middlebox processing to a cloud for lower cost and easier management. However, outsourcing middleboxes brings threats to the enterprise's private information, including the traffic and rules of middleboxes, all of which are visible within the cloud. Existing solutions for secure middlebox outsourcing either incur significant performance overhead or do not support incremental updates. In this paper, we present a secure and dynamic middlebox outsourcing framework, SICS, short for Secure In-Cloud Service. SICS encrypts each packet header and uses a label for in-cloud rule matching, which enables the cloud to perform its functionalities correctly with minimum header information leakage. Evaluation results show that SICS achieves higher throughput, faster construction and update speed, and lower resource overhead at the enterprise and in the cloud when compared with existing solutions.**

***Index Terms*—Middlebox outsourcing; Stateful middlebox; Packet transformer; Privacy-preserving; Label matching**

I. Introduction

MIDDLEBOXES are vital parts of modern networks, ranging from security appliances (e.g. firewalls and Intrusion Detection Systems (IDSes) [1]) to performance boosting devices (e.g. Web proxies and WAN optimizer [2]). Reported in a study, enterprise networks employ a large number of middleboxes [3]. While traditionally, middleboxes have been deployed as dedicated hardware devices inside an enterprise, the introduction of the Network Functions Virtualization (NFV) technology [4] and the cloud services has opened a new opportunity to outsource middleboxes to third-party clouds. An initial effort [3] indicates that middlebox outsourcing can be achieved without significantly impacting performance. Recently, there are also some industrial companies and communities working on providing in-cloud traffic processing capabilities [5][6][7].

However, it brings up an obvious concern about privacy, because in the new model, both the cloud provider and the middlebox provider may see the user's traffic and the middlebox rules, which may contain sensitive user information. For example, rules of a firewall contain sensitive information such as what traffic is not welcome, and its leakage could expose a severe security hole.

How to perform generic computing in the cloud while keeping the privacy of data has been studied extensively. The introduction of the hardware enclaves (e.g., Intel SGX [8]) provides a way to perform generic private computation; it can verify the binary before running it and can encrypt data before storing the data to enclave memory. However, this approach assumes one knows the hash of a correct binary [9] [10] and thus cannot prevent a *curious* middlebox provider from leaving a backdoor in the middlebox. Moreover, current implementations of enclaves still suffer from side channel attacks [11].

In another approach, the user can encrypt packets before sending them to the cloud/middleboxes, and previous works have studied how middleboxes can perform computation over encrypted data. These solutions are usually not generic, but it turns out that most middlebox functionalities only need a limited number of operations. For example, keyword matching, which is widely used for intrusion detection, can be performed efficiently over encrypted data [12][13].

One key challenge of the cryptographic approaches is how to handle packet headers. Headers are involved in both middlebox processing and traffic steering [14][15] (e.g., route all HTTP traffic through firewall-IDS-proxy), which need to detect whether or not an address lies within a range of values (e.g., if a header belongs to a prefix). With traditional IP addresses, one can implement such rule matching efficiently by aggregating IPs from the same subnet, because they share the same prefix. When headers are encrypted, however, such prefix property is lost, and building a lookup table using keyword matching, though possible, will create a memory explosion. Moreover, because of the dynamic nature of the network, the matching rules may change at runtime, and an ideal solution should not incur a high overhead when network configuration is changed. In summary, an ideal mechanism to handle packet headers should achieve the following properties:

1. Security guarantee. The cloud and middlebox should be able to fulfill its functionalities without learning the user's packet headers.

2. Low overhead. The mechanism should incur low processing overhead at both the enterprise side and the middlebox side, so that they can process packets at high speed. The mechanism should not consume much extra bandwidth because cloud providers usually charge traffic redirected to the cloud by volume.

3. Supporting incremental update. In modern networks, operators frequently modify network configurations (e.g., rerouting traffic to backup middlebox instances; changing the Access Control Lists (ACLs) of a firewall) to perform tasks, ranging from traffic engineering to patching security vulnerabilities [19]. SDN/NFV provides the ability to update a middlebox instance or launch a new one and reroute traffic to the new instance in a matter of milliseconds [20]. To support frequent rule updates, an ideal secure middlebox outsourcing

Huazhe Wang is with Microsoft. Chen Qian is with the Department of Computer Science and Engineering, University of California, Santa Cruz. Chen Qian (cqian12@ucsc.edu) is the corresponding author. Yang Wang is with The Ohio State University. Yu Zhao is with University of Kentucky. Xin Li, Ye Yu and Hongkun Yang are with Google.

This work was completed while Huazhe Wang, Xin Li and Ye Yu were with UC Santa Cruz and Univeristy of Kentukcy.

	Throughput	Minimum Overhead (per packet)	Incremental Update	Function Chain	Security Guarantee
Melis et al. [16]	very low	119 Bytes	✗	✗	high
Embark [17]	high	20 Bytes	✗	✓	Possible leakage of packet headers and rules
Splitbox [18]	medium	$> 2\times$ traffic	✗	✗	Possible leakage of packet headers and rules
SafeBricks [9]	$<$ Embark	-	✗	✓	Suffer from side-channel attack
SICS	$>$ Embark	4 Bytes	✓	✓	high

TABLE I: Comparison of existing secure middlebox outsourcing schemes.

mechanism should be able to update incrementally, i.e. the overhead of performing such an update should be proportional to the number of rules to be changed.

So far, none of the existing mechanisms can achieve all of the properties shown in Table I.

We design and implement a middlebox outsourcing scheme SICS, short for Secure In-Cloud Service chaining. SICS protects the private information of packet headers and rules by only allowing packets with encrypted headers into the cloud. However, encrypted headers cannot be used for forwarding and middlebox rule matching. Inspired by the concept of forwarding equivalence classes in packet forwarding networks [21][22][23], SICS assigns a label to each encrypted packet. Each label uniquely identifies the forwarding and rule-matching behavior of the packet. Switches and middleboxes in the cloud are also configured with label matching tables and processing incoming packets based on their labels. To apply forwarding equivalence classes for middlebox outsourcing, *there are key domain-specific challenges*. First, middlebox policies typically require a set of packets to go through a sequence of middleboxes, which is called a *service function chain* [24]. Those independently specified polices should be efficiently combined for packets that are subject to multiple requirements. Second, most middleboxes employ stateful processing (e.g., a firewall allows an inbound packet if it belongs to an established connection) and may modify packet headers (e.g., a source NAT converts internal addresses to external ones). However, forwarding equivalence classes can only analyze forwarding behavior of static networks [25] and cannot be directly used to handle the complexity and dynamics in middlebox chaining.

To address these challenges, we first logically group packets with the same middlebox processing chain and actions into *policy equivalence classes* and thus we eliminate the need to assign a unique label to every single flow. Second, building on configurations for header transformation, we propose a label-to-label replacement scheme. The new labels correspond to the new modified headers and are used for subsequent processing. Table I summarizes results from evaluations and compares SICS with the four recent secure middlebox outsourcing schemes in five desired properties: throughput, bandwidth overhead, incremental update, service function chaining, and security guarantee. SICS achieves all of the desired properties, while every other design contains several weaknesses.

Note that SICS focuses on how to handle packet headers securely. Middleboxes processing packet payloads can also take advantages of SICS if the header privacy becomes a concern. Similar to previous work [17], SICS is compatible to existing secure Deep Packet Inspection (DPI) over encrypted traffic and can be perfectly combined with existing methods [12][13] to handle the whole packet securely.

The rest of the paper is organized as follows. In §II we present related work. We introduce the system overview in §III. We present our detailed design in §IV (the enterprise side) and in §V (the cloud side). We show how SICS supports frequent rule updates in §VI and analyze its security in §VII. We present the implementation of SICS in §VIII and our evaluation results in §IX before concluding in §X.

II. Related Work

APLOMB [3] and Jingling [26] are the pioneer of middlebox outsourcing. APLOMB demonstrates that the latency inflation due to outsourcing is negligible. Neither of them takes privacy issues into consideration. Blindbox [12] enables equality based operations on encrypted payload of packets for a specific class of middleboxes, DPI; However, it cannot examine packet headers and/or perform range queries. Melis et al. [16] model the behavior of common middleboxes and proposed a privacy preserving middlebox outsourcing scheme based on fully homomorphic encryption [27], which has very poor performance. Meanwhile, it's not clear whether the mechanism in [16] can support function chaining. Embark [17] presents the method PrefixMatch to hide the packet header and rule information from the cloud. PrefixMatch uses the set of processing rules to divide each header field into multiple intervals and then it assigns a random IPv6 prefix to each interval. At a local gateway, every header field of a packet is mapped to a pseudorandom value of an IPv6 field separately and the entire IP packet header is mapped to a new IPv6 header. PrefixMatch does not support incremental rule updates and updating one rule requires all rules to be reconstructed, which may take as long as 100s. Before that, packets are still routed and processed as the old configuration which may incur unexpected packet loss and inaccurate processing. From a security perspective, such a field-by-field encryption scheme is vulnerable to certain types of attack, such as chosen plaintext attack. More details will be analyzed in §VII. Splitbox [18] distributes a single rule to several virtual machines (VMs), which reside on multiple clouds assuming an adversary cannot corrupt all VMs simultaneously. Computation results from all VMs are collected by a local middlebox and the final actions of the packets are calculated at the local middlebox. It is difficult for Splitbox to support service function chaining. Meanwhile, Splitbox increases bandwidth overhead several-fold as it needs

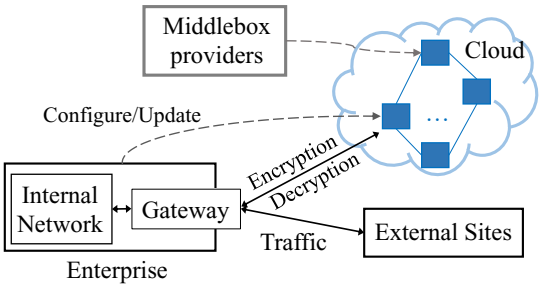


Fig. 1: The architecture of SICS

to send multiple copies of a packet to different VMs for the same network function.

SafeBricks [9] and Shieldbox [28] are state of the art *enclave-based* middlebox outsourcing solutions. Besides the potential security threats from *curious* middlebox providers and side channel attacks, they impact performance by around 15% across different in-cloud middleboxes due to the use of SGX enclaves [9] and do not support incremental update. Changing of service chains and provisions (e.g., number of deployed middlebox instances) requires rebuilding the whole enclave which takes a few minutes.

Some work [29][30][31] try to combine header based classifications and assign labels to packets for faster middlebox processing. They mostly only apply to a single service chain and focus on reducing processing time instead of traffic privacy.

III. Overview

In modern networks, most middleboxes choose the appropriate processing actions based on headers of incoming packets. When a middlebox processes a packet, it finds a rule that matches the packet header and follows the action of the rule. Hence, rule information specifies the packet processing policies of the middleboxes. Both packet headers and rules contain private information belonging to the enterprise network. To facilitate middlebox outsourcing without compromising privacy, we design and implement SICS, a Secure In-Cloud Service function chaining framework.

A. The SICS Outsourcing Architecture

As shown in Fig. 1, SICS contains three parties: an enterprise (middlebox user), middlebox providers, and a third party cloud that holds in-cloud middlebox processing. The middlebox providers set up middleboxes per request. The enterprise configures and updates rules in these middleboxes. The enterprise has a gateway that connects the internal and the external network. All incoming packets to the enterprise will be forwarded to the gateway. The gateway encrypts the packet headers and payload and sends the packets to the cloud for middlebox processing. The encryption can use symmetric-key algorithms, such as the Advanced Encryption Standard (AES), which can be performed in near line speed for 10Gbps links [17]. The encryption key is only known by the enterprise. The in-cloud middleboxes process packets following the service function chains and then the cloud transmits the packets back to the enterprise. The gateway decrypts the packets and sends them to the internal network.

The key challenge in this architecture is how the in-cloud middleboxes correctly match packets to rules given that the packet headers are encrypted. To enable correct rule-matching, SICS assigns each packet a label. The label represents all behavior of the packet in the cloud, including to which middleboxes the packet should be forwarded and in which order, as well as which rules the packet should match at a middlebox.

The operations on outgoing packets from the enterprise to an external site are similar: before being transmitted over the Internet, outgoing packets are encrypted at a gateway, redirected to the cloud, and sent back to the gateway.

Note the SICS gateway does not encrypt the checksum or TTL and instead adds a new checksum based on ciphertexts. Middleboxes can recompute checksums as usual.

An optimization that saves on bandwidth and latency can be adopted when communications are between two networks belonging to a same enterprise or two enterprises that have established a secure channel. After in-cloud processing, the traffic can directly go to the destination site without sending back since the same encryption key is shared by the two networks.

B. Security Model

In our security model, we assume the cloud and middlebox providers to be "honest but curious" [32]. They are honest to perform their services correctly. However, they might be curious to learn the user-configured processing policies at middleboxes or peek at the traffic received. This security model is practical and reflects the following real situations. First, the cloud or middlebox providers will not interrupt the normal cloud services because such an interruption will be detected [33][34]. However, it is possible that the customer data might be gathered and sold by disgruntled employees [35][36]. Additionally, hackers may try to steal the customer traffic and policy data [37][38]. SICS aims to protect the enterprise network privacy from all these attacks. We do not consider "active" attackers which manipulate customers' traffic maliciously.

SICS provides two security properties of middlebox outsourcing: (1) For an encrypted packet, the cloud and middlebox providers should not be able to infer its packet headers based on its in-cloud behavior. (2) The cloud and middlebox providers should not be able to learn the plaintext of header spaces specified by the enterprise's processing rules. In SICS, label assignment of packet headers does NOT need to be collision-resistant. Distinct packets can be assigned with the same label if they have identical behavior in the cloud. Distinct flows can still be differentiated based on their encrypted header fields if needed.

C. Middlebox with Label Matching

Label matching (known as label switching in layer 3 routing) is a technique of network relaying that is much faster than traditional IP-header switching. Each packet is assigned a label and the switching takes place after examination of the label assigned to each packet. SICS applies label matching

to middlebox outsourcing which provides two promising advantages. It can simultaneously achieve privacy protection and efficient packet processing.

Privacy protection of packet headers and rules. We name the service function chain and middlebox rule matching behavior of a packet as its *cloud-wide behavior*. A set of packets that have the same cloud-wide behavior form a *policy equivalence class*. In SICS, we assign the same label to all packets belonging to the same policy equivalence class, even if their packet headers are different. Given an encrypted packet with a label, SICS prevents an attacker from obtaining its original packet header. For example, h specifies a set of packet headers, and packets whose headers fall in h share the same cloud-wide behaviors. At the gateway, a packet is assigned a label (A label is represented as a binary string, e.g, "10110110", whose value has no relationship to the packet header) if its header belongs to h. The length of a label is determined by the total number of policy equivalence class. A label only includes two types of information: 1) which middlebox the packet should visit in the cloud, and 2) which action a middlebox should apply to this packet. The rule tables at the in-cloud middleboxes consist of label-matching entries as opposed to header matching entries. In this way, neither the cloud nor middlebox providers can learn the original middlebox processing policies with respect to the packet headers.

Note that label matching does not protect packet behavior, such as forwarding and middlebox actions. These are known to the cloud no matter what type of protection is used.

Efficient table lookup. Label matching can achieve better performance compared to the traditional header based matching (e.g., IPv4 header), especially in software middlexboxes running on general-purpose servers: (1) A label corresponds to a policy equivalence class and may cover multiple header ranges, the number of entries in a label matching table could be much smaller than that in a header matching table. In our experiment, a rule set with approximately 100K header matching rules of a function network is converted to less than 250 labels. (2) With a properly designed hash table, label matching can achieve $O(1)$ lookup time, without the use of specialized hardware such as TCAM. (3) Label matching adds little per-packet bandwidth overhead. In our experiments, a 16-bit long label is sufficient to represent cloud-wide behavior in a network with nearly one million rules. The label can be placed in the options field in IPv4 protocol header.

While the use of label matching is not new in a general networking, our specific contributions lie in the design of header space mapping in the context of secure middlebox outsourcing.

D. Design Framework

Fig. 2 shows the system model of SICS. Those modules run on a controller in the enterprise network. At runtime, the enterprise network administrator decides middlebox processing rules and the service function chaining requirements based on the business objective of the enterprise. The rule preprocessing module takes these rules and specifications as input and converts them into label-based rules. A SICS

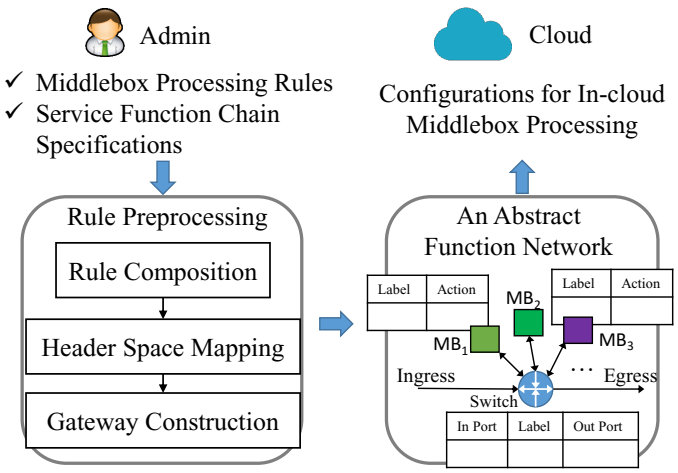


Fig. 2: The system model of SICS

gateway is constructed which assigns labels to packets based on the header space mapping relationship. To simplify in-cloud deployment, the controller then creates an *abstract function network* which includes configurations for all middleboxes and an *abstract switch* that is connected to all middleboxes. For each middlebox, there is a rule table identifying the action applied to each packet based on the label. The abstract switch is equipped with a forwarding table. Besides label and output port entry, the forwarding table has an extra entry classifying packets based on their input ports. The input ports are used to identify the segment in the service function chain that the packet is currently in. The abstract switch determines the next hop of a packet based on its label and input port.

The abstract function network can be easily mapped to the configurations of a practical deployment in the cloud that ensures packets are processed by required middleboxes in a specified sequence. Configurations are sent to the cloud from the enterprise using a VPN tunnel. When there exist processing policy or rule changes, this procedure is called repeatedly to update both the enterprise gateway and the middleboxes running in the cloud.

IV. Enterprise Modules of SICS

To enable secure middlebox outsourcing, SICS dynamically maps the header spaces specified by the middlebox processing policies to labels at the enterprise gateway. *To keep the complexity low and maintain scalability, the gateway performs only inexpensive per-packet operations, which are parallelizable.* In this section, we present the design of three key modules at the SICS enterprise side.

A. Rule Composition

The rule composition module takes the service function chain requirements and the middlebox processing rules as its input and implement its functionality in two steps.

It first combines different service function chain requirements and determines the overall service function chains for each set of packets. A service function chain requires that a class of packets must be processed by a number of middleboxes in a designated sequence. For example, all HTTP

Algorithm 1: Compute equivalence classes for middlebox chaining requirements.

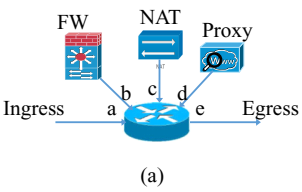
Input : Predicates of service function chain requirements (P_i for $i = 1,...,m$).
Output: A list of predicates $\mathcal{F} = \{f_1, f_2, ...f_n\}$.

$\mathcal{T}_1 = \varnothing$, $\mathcal{T}_2 = \varnothing$, $\mathcal{T}_1$.add(P_1), $\mathcal{T}_1$.add($\neg P_1$)
for $i = 2$ *to* m **do**
  for *each* $f \in \mathcal{T}_1$ **do**
    if $f \wedge P_i \neq false$ **then**
      $\mathcal{T}_2$.add($f \wedge P_i$)
    end
    if $f \wedge \neg P_i \neq false$ **then**
      $\mathcal{T}_2$.add($f \wedge \neg P_i$)
    end
  end
  $\mathcal{T}_1 = \mathcal{T}_2, \mathcal{T}_2 = \varnothing$
end
$\mathcal{F} = \mathcal{T}_1$
Return $\mathcal{F}$

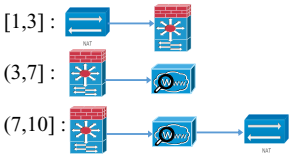
packets should go through IDS → Proxy. Packets from an internal site should be processed by NAT → Firewall. A service function chain is formulated with respect to a set of packets, specified by their packet headers, represented as a predicate P. P specifies the set of packets X for which $P(x)$ is *true* for a packet $x \in X$. A packet may relate to multiple service function chain requirements and needs to be processed by all the middleboxes included in those chains. Consider m service function chain requirements: (P_i, c_i, r_i), for $i = 1,...,m$. For the i-th requirement, let P_i be the predicate specifying the set of packets, c_i be the sequence of middleboxes, and r_i be the priority which is provided by administrators to determine the order of middlebox processing when two chains are combined. Requirements are listed in descending order of priorities. To ensure that packets are processed by all required middleboxes, SICS uses Algorithm 1 to calculate a set of middlebox chaining equivalence class, each of which specifies a set of packets with an identical service chain.

The output of Algorithm 1 is a list of predicates $\mathcal{F} = \{f_1, f_2, ...f_n\}$. The conjunction of any two predicates in $\mathcal{F}$ is *false* (referring to an empty set). Therefore packet sets specified by any two predicates have no intersection. Each predicate f_i corresponds to a service function chain, which can be obtained by concatenating c_i of P_i, if the conjunction of f_i and P_i is not false. The order is determined by their corresponding priorities.

Based on the combined service chain requirements, the rule composition module generates the forwarding table at the abstract switch to steer traffic along the required middleboxes in a sequence. Based on the input port field, we can partition the forwarding table into sub-tables. In each sub-table, we calculate one predicate for each output port by combining corresponding packet header prefixes or ranges. In our implementation, by representing packet sets as predicates, the merge operation can be performed efficiently using graph-based algorithms with Binary Decision Diagrams (BDDs) [39].



(a)



(b)

Input a

Header space	Output port
[1,3]	c
(3,7]	b
(7,10]	b
other	e

Input b

Header space	Output Port
[1,3]	e
(3,7]	d
(7,10]	d
other	e

Input c

Header space	Output Port
[1,3]	b
-	-
(7,10]	e
other	e

Input d

Header space	Output port
-	-
(3,7]	e
(7,10]	c
other	e

(c)

Header space	Output port
[1,3]	c
(3,10]	b
other	e

Header space	Output Port
(3,10]	d
other	e
-	-

Header space	Output Port
[1,3]	b
other	e
-	-

Header space	Output port
(7,10]	c
other	e
-	-

(d)

Fig. 3: (a) An abstract function network. (b) Service function chain requirements. (c) Original forwarding table. (d) Merged forwarding table.

With predicate aggregation, there exists at most one predicate per output port in each sub-table. We use the example shown in Fig. 3 to illustrate this process. Fig. 3(a) is an abstract function network with three middleboxes. All middleboxes are connected by an abstract switch with five ports. Port b, c and d are used to link the middleboxes and port a and e are ingress and egress ports. Fig. 3(b) shows three sample service function chains. The set of packets in each chain is specified by an integer range. [1] Fig. 3(c) is the original forwarding table at the virtual switch that steers traffic across the middleboxes according to the service chains in Fig. 3(b). From Fig. 3(c), we see that many items in each sub-table share the same output port. This allows us to reduce the size of each table by merging ranges which have the same output port. The resulting forwarding table is shown in Fig. 3(d). We reduce the total items in the forwarding table from 14 to 9.

The second step of the rule composition module is combining user-configured middleboxes processing rules which are created locally either by the network administrator or middlebox providers. We define the middlebox rules with the 3-tuple: (P_i, b_i, r_i), where P_i denotes the predicate from the i-th rule, b_i is the action performed on packets matching this rule and r_i is the priority. We sort all rules at a middlebox in descending order with respect to priorities. When a packet is checked against the rules at a middlebox, it is matched by the first rule whose predicate evaluates to true. We use Algorithm 2 to convert the rules of a middlebox to a list of predicates $\mathcal{F} = \{f_1, f_2, ...f_n\}$, each of which specifies the packets sharing the same behavior at the middlebox, where n is the total number of distinct behavior. For example, a firewall may have a predicate specifying packets allowed by the ACLs

[1] In our implementation, all packet sets are converted to predicates and represented by binary decision diagrams (BDDs) [39]. Here we use integer ranges for simplicity.

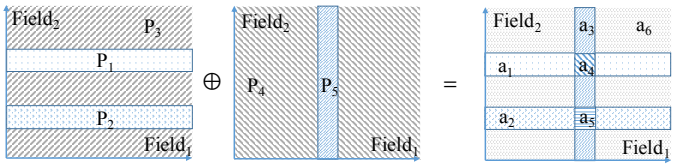


Fig. 4: Header space divided by predicates

and another predicate specifying the ones denied.

Algorithm 2: Compute a predicate for each action.

Input : Sorted processing rules at a middlebox (P_i for $i = 1, ..., m$)

Output: A list of predicates $\mathcal{F} = \{f_1, f_2, ...f_n\}$

```
for j = 1 to n do
    f_j ← false
end
valid ← false
for i = 1 to m do
    if P_i shares the same action as f ∈ F then
        f ← f ∨ (P_i ∧ ¬valid)
        valid ← valid ∨ P_i
    end
end
Return F
```

B. Header Space Mapping

After rule composition is performed, we obtain a list of predicates for each middlebox and the abstract switch. Predicates from a box can be seen as a partition which divides the packet header space into several disjoint sub-spaces, each with the same action. If we place predicates from all of the boxes together, the partition of the header space will become combinatorically finer due to the intersection of predicates from different boxes.

Fig. 4 shows an example illustrating the process of placing predicates from two boxes into a single header space. Each predicate is associated with two header fields[2]. Five predicates $P_1 \sim P_5$ from the two boxes are placed together in one packet header space. Then, the header space is partitioned into 15 blocks. Each block represents a set of headers belonging to the same set of predicates. The packet headers within one block will match the same set of predicates and exhibit identical behavior at all boxes. Therefore, they have the same cloud-wide behavior and hence belong to the same policy equivalence class. Note that a policy equivalence class is not necessarily a single block. Blocks that are specified by the same set of predicates belong to the same equivalence class. As shown in Fig. 4, the original predicate P_1 is divided into three segments. The right and left segments are only covered by P_1 and form an equivalence class a_1. The segment in the middle is covered by both P_1 and P_5 and forms an equivalence class a_4. In total, the partition of 15 blocks forms 6 equivalence classes represented by $a_1 \sim a_6$.

To obtain the policy equivalence classes, SICS reuses Algorithm 1 given a list of predicates. At this time, the input is the

[2]In practice, a predicate may be defined over multiple fields, e.g., 5-tuple in TCP/IP packets. Here, we use two dimension headers as an example.

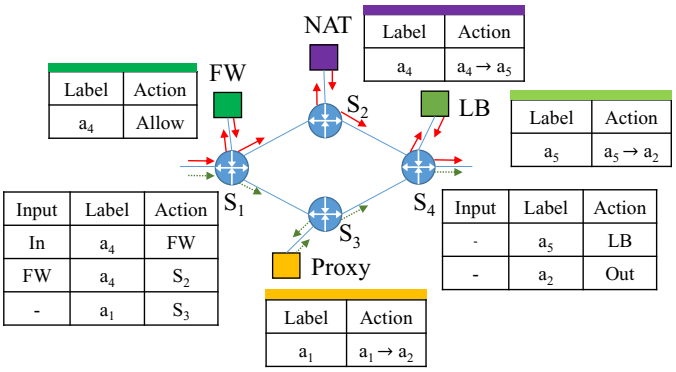


Fig. 5: An example Abstract Function Network

set of predicates from all middleboxes and the abstract switch. The set of policy equivalence classes has two key properties: (1) Packets within the same class have identical cloud-wide behavior. That is, these packets will traverse the same sequence of middleboxes and have same behaviors at each middlebox in the network. (2) Each input predicate is equal to the disjunction of a subset of policy equivalence classes, shown in Fig. 4 where $P_1 = a_1 \vee a_4$ and $P_5 = a_3 \vee a_4 \vee a_5$.

SICS maps packet headers within an policy equivalence class to one label. In the rule tables of the in-cloud boxes, predicate P is represented as *a set of labels*, which are determined by the subset of policy equivalence classes whose disjunction is P.

C. Example

We show an example abstract function network configured with labels in Fig. 5. The abstract switch is divided into four separate switch instances with each connecting to a single middlebox. We have two flows h_1 and h_2. Flow h_1 is required to go through a firewall, a NAT and a load balancer, while flow h_2 should go through a proxy. For simplicity, we assume the sets of predicates for all middleboxes and switches in Fig. 5 have a similar partition of the packet header space as in Fig. 4. For example, switch S_1 has two predicates that specify the same partition as P_4 and P_5. P_5 specifies the set of packets that are forwarded to the firewall and other packets specified by P_4 are forwarded to S_3. The NAT has three predicates which specify the same partition as $P_1 \sim P_3$. Packets matching P_1 are translated to packets specified by P_2. P_3 represents a default drop predicate. The set of policy equivalence classes are still $a_1 \sim a_6$ as in Fig. 4. h_1 and h_2 belong to the packet sets specified by a_4 and a_1, respectively. Relevant entries for the two flows are shown in the label matching tables of middleboxes. The two forwarding tables are for switch S_1 and S_4. From the figure, we can see packets in h_1 (red arrows) will be forwarded to and allowed by the firewall. After that, the label is changed to a_5 and then a_2 by the NAT and the load balancer sequentially based on label replacement actions. Details on label replacement are presented in §V-B. Finally the packets are forwarded to the egress by S_4 with the label a_2. Similarly, packets in h_2 (greed dotted arrows) are processed by the proxy before they are sent back to the gateway. Note the input port field at a switch is necessary when incoming and outgoing packets share the same label.

D. Packet Classification

To assign labels to packets, the gateway determines to which policy equivalence class a given packet belongs. Policy equivalence classes can be represented as the conjunction of input predicates. SICS uses all predicates obtained from the rule composition module to build a packet classifier, using the algorithms in [25][40]. The proposed classifier includes a binary tree whose root has a predicate P_1. At level i, the 2^i internal nodes each has a predicate P_i. Starting from the root, at each internal node, the input packet header is evaluated by the predicate of the node. If the result is true, the packet continues to be evaluated in the left sub-tree. Otherwise, it goes to the right sub-tree. A leaf node represents an policy equivalence class and the set of packets that can reach this leaf belong to the policy equivalence class. In practice, for a tree constructed by k predicates, its height is considerably lower than k and the number of leaves is significantly smaller than 2^k. The reason behind this observation is that conjunctions of a large number of predicates are are *false* and specify empty sets of packets, no new node will be created. More importantly, using the methods in [25], the classifier supports incremental updates when there exist policy changes. For example, new predicates can be added at the bottom of the tree with little overhead.

The gateway classifies packets into one of the policy equivalence classes, with each has a unique cloud-wide behavior. This corresponds to the provable coarsest refinement of packet header space and thus can be used to provide best computation time and space performance of the gateway.

V. In-Cloud Modules of SICS

SICS aims to protect the privacy of packet headers for in-cloud middleboxes. To also hide packet payload (e.g., keyword matching in intrusion detection), SICS can be combined with recent work of secure DPI [12][13].

Note that for very simple middleboxes, such as a stateless firewall blocking certain IPs, the gateway can fulfill its task when computing the label, packets that only traverse these middleboxes are processed locally and do not need to be redirected to the middleboxes running in the cloud. However, we observe many middleboxes involve expensive operations and for this reason enterprises tend to outsource them.

A. Stateful Middlebox

So far, SICS gateway identifies the cloud wide behavior of a packet solely based on an equivalence class obtained from the rule preprocessing phase. However, unlike switches or routers, common middleboxes conduct stateful functionalities (e.g., bidirectional firewall and address translation [41][42], stateful load balancing [43][44]) and use advanced statistical techniques to detect and prevent potential security threats (e.g., flood protection [45][7]). For an incoming packet, most stateful middleboxes first check if the packet belongs to an existing state, if it does, it will be applied with the action corresponding to the state. Otherwise it is searched against a rule table and processed based on the first entry it matches. Such functions can be resource-consuming since they need to maintain a separate state for every single connection. Since such functions rely on per-connection states, in-cloud middleboxes should be able to recognize packets of the same connection based on encrypted packet headers.

In SICS, all header fields are encrypted as a whole to provide high security guarantee and thus cannot be used to identify packets of the same connection. To support per-connection states, SICS adds a 32-bit connection identifier to each packet based on a pseudorandom function. In our prototype, prf is implemented using AES and truncated to 32 bits.

$$I_c = prf((IP_{src} \| port_{src}) * (IP_{dst} \| port_{dst}))$$

Using the equation above, the inbound and outbound packets of the same connection will have the same identifier. By conducting experiments using a real dataset [46], we observe that the probability that two packets from different connections having the same identifier is negligible. Note adding an identifier to recognize packets of the same connection is a general approach that can be applied to other middlebox outsourcing work, such as Embark [17] and Splitbox [18].

Algorithm 3: Compute equivalence classes after adding header transformers.

Input : A list of predicates $\mathcal{P}$ and a set of packet transformers $\mathcal{T}$
Output: A list of predicates $\mathcal{F} = \{f_1, f_2 ... f_n\}$

$\mathcal{F} \leftarrow \mathcal{EC}(\mathcal{P})$, $\mathcal{P} \leftarrow \mathcal{F}$
for $T \in \mathcal{T}$ *and* $f_i \in \mathcal{F}$ *that can be transformed by* T **do**
  $\mathcal{P} \leftarrow \mathcal{P} \bigcup T(f_i)$
end
$\mathcal{F} \leftarrow \mathcal{EC}(\mathcal{P})$, $\mathcal{P} \leftarrow \mathcal{F}$
for *each deterministic* $T \in \mathcal{T}$ *and* $f_i \in \mathcal{F}$ **do**
  Compute the set $\mathcal{B} = \{b_1, b_2, ... b_l\} \subseteq \mathcal{F}$ whose disjunction is $T(f_i)$
  $\mathcal{R} \leftarrow \{T^{-1}(b_j) |$ for each $b_j \in \mathcal{B}\}$
  $\mathcal{P} \leftarrow \mathcal{P} \bigcup \mathcal{R}$
end
$\mathcal{F} \leftarrow \mathcal{EC}(\mathcal{P})$
Return $\mathcal{F}$

B. Header Transformer

In SICS, a single label is sufficient to guide all rule matching behavior of a packet if it does not traverse middleboxes that modify packet headers. As shown in Fig. 5, header transformers such as NAT, load balancer may modify packet headers. When a packet goes through a header transformer, the behavior of the packet at downstream boxes is determined by its new header. With label matching, the subsequent packet behavior must be determined by the new label corresponding to the new header. Hence, middleboxes must be able to assign new labels to packets they have just modified without ever learning their headers.

To address the above problem, we design a label-to-label replacement scheme. A packet transformer maps an input packet set to an output packet set. For a packet transformer T and a predicate P specifying its input packet set, $T(P)$ denotes

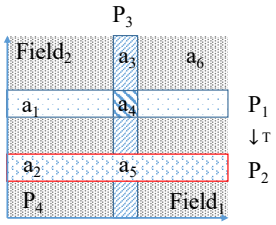


Fig. 6: Compute equivalence classes with a header transformer

the transformed predicate specifying the output packet set. More specifically, given a predicate P, $T(P)$ can be calculated by replacing constraints on corresponding header bits. For example, a transformer for a four-bit prefix $11\star\star$ modifies the second bit from 1 to 0. This operation can be modeled by applying existential quantification and conjunction of the new constraints to the second bit. The transformed predicate represents prefix $10\star\star$. Similarly, T^{-1} can be calculated using the inverse process. SICS supports both deterministic (e.g., one to one mapping from a prefix to another) or non-deterministic (e.g., randomly choose a new address from a given prefix) packet transformers.

Header transformers may produce new policy equivalence classes. Given a list of predicates $\mathcal{P}$, we extend Algorithm 1 to calculate the new set of policy equivalence classes, denoted as $\mathcal{EC}(\mathcal{P})$, when header transformers exist. As shown in Algorithm 3, a new set of policy equivalence classes is calculated after a set of transformed predicates are added (line 5). For a transformer T, the transformed predicate $T(f_i)$ for a policy equivalence class f_i is equal to the disjunction of a subset of equivalence classes $\mathcal{B} = \{b_1, b_2, ...b_l\}$. If T is non-deterministic, a packet in the packet set specified by f_i is randomly transformed into a packet that belongs to either one of equivalence classes within $\mathcal{B}$. However, if T performs a one-to-one mapping, a transformed packet must belong to a deterministic policy equivalence class. To decide into which equivalence class a packet should be transformed, lines 6-11 of Algorithm 3 calculate the inverse predicate for each $b_i \in \mathcal{B}$ and update the set of equivalence classes. Then, each deterministic transformer has a one-to-one mapping for all policy equivalence classes. With the refined set of policy equivalence classes, SICS can easily build a label replacement table for each header transformer. Upon receiving a packet with a label that can be processed by the transformer, a non-deterministic header transformer randomly modifies the label to one of the multiple labels, whereas a deterministic header transformer always conducts a unique label replacement action. We revisit the example in Fig. 4 to illustrate how Algorithm 3 works. As shown in Fig. 6, originally there are three predicates P_1, P_3 and P_4, producing four equivalence classes a_1, a_3, a_4 and a_6. We assume a NAT with a transformer T modifies headers from P_1 to P_2. Following the line 3 $\sim$ 5 in Algorithm 3, a new set of equivalence class is obtained $a_1 \sim a_6$. To build the label replacement table, line 6 $\sim$ 11 calculate the reverse of transformed predicates and check whether a new equivalence class is produced. In Fig. 6, no new equivalence class is added since $T^{-1}(a_2) = a_1$ and $T^{-1}(a_5) = a_4$,

In addition to replacing labels, the middlebox also assigns an index corresponding to the modified header, e.g., an index for an IP in a prefix stored at the gateway. When the gateway receives a packet with such an index, it restores the modified header fields.

To keep the connection identity, a header transformer maintains a mapping from the newly assigned header/index to the original connection identifier. For reverse packets, the gateway does not encrypt assigned header fields (e.g., random port number ranges assigned by a NAT). Upon receiving packets with the same assigned header fields, the transformer restores the connection identifier. So the same processing policy is applied in subsequent middleboxes.

C. Case Studies

Next, we use a proxy and a firewall-NAT chain as examples to discuss how SICS combines the two techniques above to support more complex real-world middleboxes.

Proxy. An HTTP proxy accepts a TCP connection from a client, decides whether it is a hit or miss by looking up the URL in the local cache. (a) Hit: The proxy creates a reply packet with encrypted headers from the request packet as well as the requested content. It also adds an address index pointing to its own address to indicate it is a proxy reply packet. When the gateway receives the reply packet, it decrypts the packet header and restores the source and destination addresses of the packet. (b) Miss. The proxy creates a new request with the same encrypted headers. The proxy adds an address index pointing to the server which is configured for the requested URL. When the packet bounces back to the gateway, the gateway decrypts the packet header and replaces the destination address with the server's address. In the reverse direction, reply packets from the server are encrypted and received by the proxy. The proxy caches the replied content and sends the content back, as in case (a). During this process, packets are forwarded and processed by the proxy in the cloud without exposing the headers.

Firewall-NAT. We consider a firewall-NAT chain that examines packets headers. The NAT function can be divided into two categories: source NAT and destination NAT. A source NAT translates the headers of connections initiated within internal networks, while a destination NAT applies to connections started from outside networks.

For a packet initiated within the inside network, the firewall first applies its label-based ACLs and stores the connection identifier if the packet is allowed. Then, the NAT adds an index for a reserved external IP, a random port number and assigns a new label to the packet based on the label replacement table. Note header transformers may break the connection identity between outbound and inbound packets. To make the connection reversible, the NAT maintains a mapping from the newly assigned port number to the packet's original encrypted headers as well as the connection identifier. Before packets are sent out to external networks, the gateway decrypts and restores the header fields assigned by the source NAT. For a reverse packet, if the destination port belongs to the range of random port numbers assigned by the source NAT, the gateway encrypts the packet and places the port number in the options field of the packet. Using the port number, the

NAT restores reverse packets with the corresponding original encrypted headers and the connection identifier. So the same processing policy is applied to reverse packets at the firewall. Packets initiated from outside networks have similar processing schemes, except that a destination NAT maintains a deterministic one-to-one mapping from a public address to a private address.

VI. Update operations

Overload is a common cause of middlebox failures [47]. Traffic should be steered across different middlebox instances dynamically. Service function chain requirements and middlebox processing rules are also changing constantly to meet the new costumers' needs or reduce security threats. All changes in traffic processing result in rule updates at the enterprise and on the cloud sides. To keep the correctness and performance of in-cloud processing, it is necessary for a middlebox outsourcing framework to support incremental rule updates with low latencies. A rule insertion or deletion can be converted to predicate changes [22]. If there are predicate changes after the rule updates, SICS performs the following methods to update both the enterprise side and the in-cloud boxes.

Update at the enterprise side. SICS starts by updating the packet classifier at the gateway. When a new predicate is added, SICS adds the new predicate to the bottom of the packet classifier. If the update produces new equivalence classes, the packet classifier starts to classify packets to the new set of equivalence classes. When existing predicates are deleted, SICS updates the set of equivalence classes by merging the equivalence classes if they identify the same cloud-wide behavior. Updates to the classifier can be executed very fast. In our experiments, the average cost of adding/deleting a predicate is less than 0.5 ms.

To figure out the update schemes of in-cloud boxes, the enterprise controller maintains a representation list for each predicate. This list includes all equivalence classes whose disjunction is equal to the predicate. In the example shown in Fig. 4, the representation list of P_5 is $\{a_3, a_4, a_5\}$ and for P_2 it is $\{a_2, a_5\}$. Representation lists of predicates are maintained dynamically, so when the list of a predicate is modified, the controller sends update instructions to the in-cloud box which produces the predicate.

Update in the Cloud. In SICS, a rule update in the cloud consists of the updating of the rule tables (hash tables) at each middlebox and the abstract switch. The forwarding table of the abstract switch is partitioned into several sub-tables which are updated independently. When a new equivalence class is added into the representation list of a predicate, its label-action pair is inserted into the rule table of the in-cloud box that produced the predicate. Here, the key is the label which corresponds to the policy equivalence class and the value is the action of the predicate. In contrast, a label-action pair is removed from the rule table when the corresponding equivalence class is deleted from the representation list of the predicate.

The connection states maintained in the stateful middleboxes will not be disrupted during an update since states are identified by encrypted packet headers or connection IDs.

Maintaining Processing Consistency. Rule updates need to be treated carefully. Any inconsistency in state between the gateway and the boxes in the cloud may lead to incorrect middlebox processing. To maintain per-packet consistency, the controller first calculates the incremental rule update schemes for the enterprise gateway and boxes involved in the cloud. During this time, the gateway and in-cloud middleboxes continue to encrypt and process traffic according to the old rules. Once the update schemes are determined, the gateway buffers incoming packets until all in-cloud packets finish processing in the cloud (The buffering time is bounded by the packet processing time, which is typically hundreds of milliseconds [3]). Then, the gateway and in-cloud boxes install updates and start processing new packets. To maintain flow consistency, ongoing flows should continue traversing the original sequence of middleboxes while they are updating. SICS employs the migration avoidance mechanism in [48]. New flows are steered to new middlebox instances while existing flows are still processed by old ones.

VII. Security Analysis

SICS converts IP prefixes and other header spaces from middlebox processing rules to a list of predicates. Each predicate is represented as a set of labels that are used as matching fields to enable in-cloud functionalities. Labels do not leak size, order or borders of header spaces specified in the rules. The cloud is unable to learn to which field of the packet header a match corresponds. Labels at in-cloud middleboxes are updated independently and the information about header spaces represented by these labels cannot be inferred from updates. A gateway encrypts packet headers and assigns a label to each packet in order to identify its in-cloud processing. In this case, given an encrypted packet with a label, its original packet header cannot be reversed from the label. For any two packets that are assigned the same label, the cloud is limited to learning that the two packets have the same cloud-wide behavior, but prevented from determining any other information about their orders or values.

Information leakage. From an information-theoretic point of view, information leakage of a communication system is at least $log_2 N$ bits, where N is the number of observable equivalence classes [49]. In the context of SICS, each equivalence class identifies a cloud-wide behavior, which is represented by one label. The label instructs the in-cloud boxes to process the packet as configured. With fewer number of cloud-wide behaviors, the cloud may not be able to correctly perform its functionalities. In this sense, SICS achieves minimal information leakage. On the other hand, Embark employs a field-by-field encoding to convey the information about how packets should be processed in the cloud. The set of cloud-wide equivalence classes are the Cartesian product of per-field equivalence classes. Consequently, Embark exposes a larger number of observable equivalence classes and hence more information leakage.

Next, we demonstrate that the security of SICS is stronger than the PrefixMatch in Embark [17] under two attacks.

Chosen Plaintext Attack. A chosen plaintext attack allows an attacker to determine which plaintext message is encrypted

into an input ciphertext message. We assume that an attacker (e.g., the cloud itself or a hacker) selectively sends sample packets to the gateway and observes their cloud-wide behavior, attempting to figure out the plaintext of the rules at a middlebox. PrefixMatch adopts a per-field encryption scheme where prefixes or ranges for each header field are encrypted separately. For an encrypted prefix or range, the attacker knows to which field of the packet header the prefix or range corresponds. The plaintext of the encrypted prefix or range can then be obtained by traversing the entire search space of that field.

An example of such attack is the following: for the destination port field in the packet header, PrefixMatch encrypts a port number interval $[s,e]$ to a random interval $[S,E]$. All port numbers falling in $[s,e]$ are encrypted to values in $[S,E]$. Knowing the interval $[S,E]$, it takes an attacker at most 2^{16} queries (e.g., sample packets with a destination port traversing from 0 to 2^{16}) to find all port numbers in $[s,e]$, where 16 is the length of the port field. Now the attacker has successfully deciphered the encrypted interval $[S,E]$ in the cloud. In addition, when a future packet matches the interval $[S,E]$, the attacker learns that the original destination port of the packet falls in $[s,e]$. Similarly, the attacker could learn mapping relationships for other fields. Since a chosen packet header can test each header field simultaneously, the number of required queries to decipher all header fields is determined by the length of the longest header field. For a 5-tuple, the longest header field is 32 bits. So it takes at most 2^{32} queries to decipher a 5-tuple based ruleset which is encrypted using PrefixMatch.

As described in §IV-B, SICS encrypts packet header fields as a whole. This means all packet header fields are involved in the header space mapping process, i.e., the label of a packet is determined by all of the bits in its header. When considering the same attack just described, we clearly see the benefit of SICS which require 2^{104} queries to decipher, a significant improvement over PrefixMatch's 2^{32}. PrefixMatch cannot be modified to encrypt all fields as a whole since the encryption in PrefixMatch is based on comparing per-field values of packets and the endpoints of rules.

Frequency Analysis Attack. Frequency analysis is a classic inference attack that has been historically used to recover plaintexts from substitution-based ciphertexts, and is known to be useful for breaking deterministic encryption. In frequency analysis, an adversary acquires knowledge of the frequency distribution of plaintext messages (e.g., via unintended data release or data breaches), counts the frequency of ciphertext messages and maps each ciphertext to the plaintext in the same frequency rank. To conduct frequency analysis, we assume the cloud is able to obtain the plaintext enterprise traffic from a previous time period and tries to infer the current encrypted traffic using the previous frequency distribution. To prevent frequency analysis, SICS adds randomness to the encryption of the original packet headers and the connection identifiers by changing the seed for symmetric key generation and the pseudorandom function after a certain time period. In SICS, it is not useful to add randomness to the labels of packets. For example, if a new label is assigned to a packet when the behavior of the packet does not change, the cloud can easily determine the new label is equivalent to the old label because they specify the same cloud-wide behavior. However, frequency analysis only achieves low inference accuracy in SICS. One reason is that because a label in SICS covers a range of packet headers, the cloud cannot infer the frequency of each single packet header using the frequency of the label. Another reason is that the frequency analysis is sensitive to label updates that occur during middlebox load balancing and the changes in processing policy over time. An update to a label can change the frequency rank of multiple labels, including the label itself as well as other labels with similar frequencies. In contrast, PrefixMatch uses a one-to-one deterministic header mapping which is less secure in terms of frequency analysis.

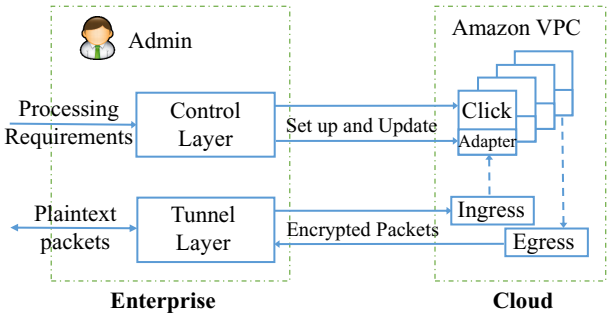


Fig. 7: SICS software architecture

VIII. Implementation

We have built a SICS prototype in our laboratory using middleboxes running in the Amazon Virtual Private Cloud (VPC) [50] and a gateway running on a general purpose desktop computer with eight cores, 3.20 GHz Intel Core i7-6700 Processor and 32GB memory. The gateway redirects traffic from another machine of the same model connected with a 10GbE link.

Fig. 7 shows the software architecture of SICS. The enterprise side consists of two layers: a control layer and a tunnel layer. The control layer takes the service function chain requirements and processing rules of the middleboxes as its input to calculate an abstract function network. When there are changes, the control layer updates the packet classifier in the tunnel layer and calculates the necessary updates in the cloud. Then, it sends batched update instructions to the middlebox instances running in the cloud. The tunnel layer, built on Intel's DPDK [51], acting as a gateway, performs packet manipulation, header encryption and VPN tunnels connecting remote instances in the cloud.

On the cloud side, the abstract function network can be easily converted into a practical deployment within the Amazon VPC. SICS supports all header-related in-cloud functions (e.g., firewall, NAT, traffic steering). We implemented middleboxes using Click [52] and rule tables using the Cuckoo hash table [53][54]. To enable in-cloud middlebox chaining, SICS adds an adapter layer which holds a sub-forwarding table from the abstract switch at each middlebox instance. Based on their labels, the adapter decapsulates incoming packets for current processing and encapsulates outgoing packets with the address of the next middlebox.

A possible limitation of SICS is that SICS employs label matching which requires modifications to the existing header matching based middlebox implementations.

IX. Evaluation

We now investigate the performance of SICS at both the enterprise side and in-cloud middleboxes.

A. Enterprise-side performance

1) Gateway: We first evaluate the performance of the SICS gateway. For most experiments, we use a synthetic workload generated by the Pktgen traffic generator powered by DPDK [55]. We create an abstract function network using Stanford dataset [46] with three types of middleboxes: firewalls, source NATs and destination NATs. A destination NAT is used to implement a L4 load balancer. The Stanford dataset has 16 routers (2 backbone routers connected to 14 zone routers) with 757170 IPv4 forwarding rules and 1584 ACL rules. Firewalls can be placed on any router. For each firewall, we randomly select ACLs from the ruleset and shuffle the order to achieve different security policies. NATs are added to the dataset connecting zone routers to private subnets. For each NAT added, we use a different public IP address for the newly created port of the zone routers and a different private prefix for the subnet. A subset of forwarding rules are used to steer traffic along middlebox chains. We vary the number of middleboxes from 0 to 16 with the total number of rules increasing from 100K to 800K to show how the performance of SICS is affected by the network size. We compare the SICS gateway with PrefixMatch in Embark [17] since PrefixMatch is the only existing *cryptographic* approach that supports service function chaining. We report the median of 10 iterations for each experiment.

Construction time. Table II shows the construction time of the gateway with respect to the network size. For SICS, rule composition accounts for the most of the overhead while computing equivalence classes and constructing the packet classifier can be finished in tens of milliseconds. In Embark, the time cost is the time to construct the data structure for PrefixMatch. The PrefixMatch structure in Embark works only on one header field, so PrefixMatch needs to be run for every header field, one after another. In Table II, we see that the time cost of PrefixMatch in Embark is at least 5 times larger than SICS for all six network sizes. The reason is that the total number of sub-intervals for each header field in PrefixMatch is much larger than the number of policy equivalence classes in SICS. For example, the test network with 100K rules produces approximately 200 equivalence classes; whereas the number of sub-intervals calculated using PrefixMatch is over 9000. This highlights the efficiency of the SICS approach compared with the process used by PrefixMatch when it finds the intervals pertaining to the same set of prefixes, especially when the size of the network is large. As shown in Table II, the construction of the gateway in SICS only uses 368.3ms for the network with 100K rules and it is still less than 10s when the size of the network increases to 800K.

Incremental rule update cost. In this set of experiments, we first construct the packet classifier using a subset of predicates and then keep adding new or deleting existing predicates. In Fig. 8, we measure the time cost to update each predicate. We find that the medium time cost for updating a predicate does not have a distinct difference when the network size increases. The medium time cost for updating a predicate is less than 0.5 ms for all networks.

No. of Rules (K)	Rule Composition (s)	Computing ECs (ms)	Packet Classifier (ms)	Embark (s)
100	0.3	14.9	53.4	7.2
200	1.1	15.2	83.2	12.6
400	2.9	22.4	129.0	18.8
600	7.1	25.2	148.2	50.3
800	9.4	30.5	249.8	76.43

TABLE II: Construction time of the gateway.

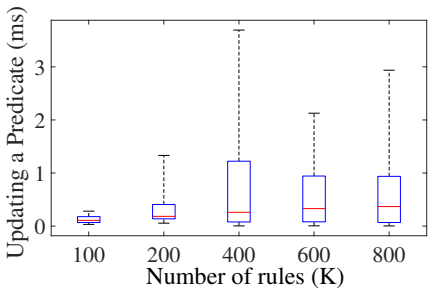


Fig. 8: Box plot of update cost.

PrefixMatch in Embark may need to be reconstructed when a rule changes and the reconstruction process costs nearly 100s. PrefixMatch can still process packets using old configurations during the reconstruction; however, the long update delay may incur packet losses and harm the accuracy of middlebox processing. The situation worsens when updates happen frequently.

Throughput.

We first measure how throughput of SICS gateway scales with network size. Packets used in the experiments are generated uniformly with respect to equivalence classes and results for various network sizes are shown in Fig. 9. From the figure, we find that the gateway in SICS can achieve 3.92 Mpps for the network with 100K rules. For the largest network with 800K rules, the throughput is 2.2 Mpps. For all networks, the throughput of the gateway in SICS is higher than Embark by approximately 20%.

To investigate the potential overhead introduced by SICS gateway, we measure throughput of SICS gateway when encrypting traffic to send to the cloud and a simple redirection [3] as the baseline. As shown in Table III, SICS gateway averages 8.49 Gbps and 8.80 Gbps for a mixed and a full size trace. No significant regression is observed when comparing the throughput of SICS gateway with a simple redirection. For minimal size traces, the throughput goes down when packet classification in SICS gateway becomes a bottleneck.

Thoughput (Gbps)	Min Size	Max size	Realistic (Mixed)
Redirection	7.25	8.82	8.56
SICS	1.41	8.80	8.49

TABLE III: Throughput on a single core at SICS gateway.

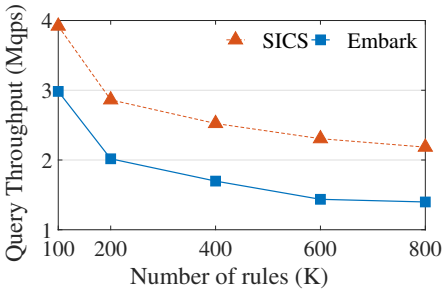


Fig. 9: Throughput as the number of rules increases.

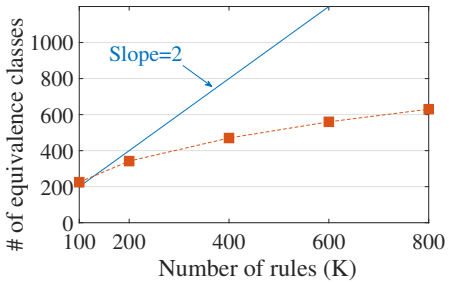


Fig. 10: Number of ECs as the number of rules increases.

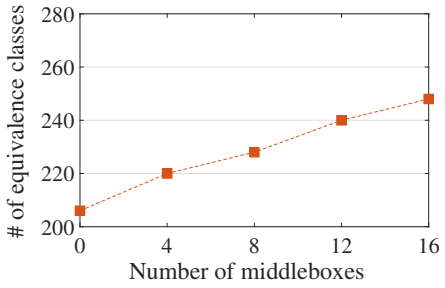


Fig. 11: Number of ECs as the number of middleboxes increases.

Memory usage. The SICS gateway only stores predicates, calculated by the rule composition module, instead of rules. Predicates are represented as BDDs in our implementation. For each predicate, the controller maintains a representation list recording a subset of equivalence classes and their corresponding labels whose disjunction is equal to the predicate. Each equivalence class is represented as a set of pointers to predicates which contain the equivalence class. With Embark, the memory cost of the data structure for PrefixMatch is also calculated. For all network sizes, the gateway of SICS uses less memory than Embark. The memory cost is 0.267MB for SICS and 0.274MB for Embark when the network size is 100K. For the largest network with 800K rules, SICS and Embark uses 0.349MB and 1.345MB respectively. Neither the gateway in SICS nor Embark consumes appreciable memory since they only store the classifier and not the rules.

Scalability of the gateway. As shown in previous results, the performance of Embark degrades sharply as the total number of rules increases. Compared with Embark, the performance of SICS mainly depends on the number of equivalence classes calculated from these rules, which is a much smaller value than the number of rules. Given processing rules and service chaining requirements, the number of equivalence classes is determined by the number of various possible actions at the middleboxes and the service function chains, not by the total number of rules. For example, a firewall with 10K ACL rules produces only two equivalence classes, with each one corresponding to the action deny and allow, respectively. Fig. 10 shows the number of equivalence classes with respect to the number of rules in the function network. With fewer equivalence classes, SICS is more likely to achieve high throughput and good scalability. In the figure, we can see that the number of equivalence classes grows at a rate of less than 2. To show how the number of equivalence classes increases when middleboxes are added to existing networks, we keep adding middleboxes into the network with 100K rules. Fig. 11 shows the number of equivalence classes versus the number of middleboxes added. The increase in the number of equivalence classes is about 2 for each middlebox on average.

2) Bandwidth Overhead: We evaluate the extra bandwidth overhead between the enterprise and the cloud. Embark introduces a 20-byte overhead per IPv4 packet because it converts them to IPv6. SICS only inserts a 16-bit label into the options field of IPv4 packets which encodes up to 65536 equivalence classes (cloud-wide behavior). For middleboxes that modify packet headers, SICS uses another 16 bits as the identifier to represent rewritten header fields. For stateful middelboxes, SICS adds a 32-bit connection ID. Hence, the total per-packet bandwidth overhead introduced by SICS is 64 bits or 8 bytes. This is placed in the options field of IPv4 protocol header.

3) Processing Delay: SICS employs a similar middlebox outsourcing architecture as Embark which involves encryption and redirection overhead. Compared with local processing, deploying SICS in the Amazon VPC incurs hundreds of milliseconds processing delay; whereas an ISP based deployment with a larger footprint with respect to the Amazon VPC can reduce the delay to tens of milliseconds [17].

B. In-cloud Middleboxes

In this section, we evaluate the performance of label matching based in-cloud middleboxes. We develop middleboxes with existing Click elements [52] and lookup tables using (2,4)-Cuckoo hash tables [54], which each uses 64 KB memory.

Throughput of in-cloud middleboxes. For comparison, we also implement prefix matching based firewall and NAT using raw click elements. The only difference between RAW-Click and SICS-Click is how middleboxes search for a match for an incoming packet. Each middlebox has 1000 IPv4 5-tuple rules. Fig. 12 shows the throughput in thousand of packets per second (kpps, log scale) for the two middleboxes. We see that the throughput of label matching based firewall and NAT in SICS is about 8000 kpps, which shows an improvement of two orders of magnitude over their header based pattern matching counterparts.

Reacting to middlebox failures and overload. We consider two dynamic scenarios: (1) a middlebox fails and (2) traffic overload at a middlebox. We measure the reaction time of SICS for each scenario and the results are shown in Fig. 13. When a middlebox fails, we need to migrate the state of the

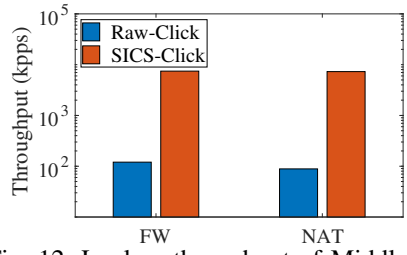


Fig. 12: Lookup throughput of Middleboxes.

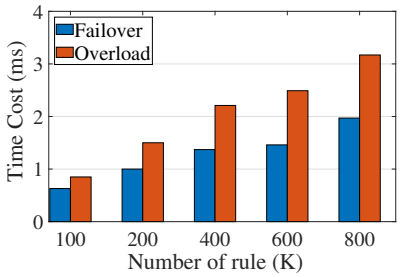


Fig. 13: Response time in the case of a middlebox failure and traffic overload.

failed middlebox to a new instance and configure the network to reroute packets with certain labels to the new instance. To prevent traffic overload at a middlebox, in addition to middlebox state migration, we need to add new predicates to split a portion of traffic on the current middlebox to another middlebox. This requires additional updating of the packet classifier at the gateway and representation lists at the controller. From Fig. 13, we see that the overall time to react to middlebox failure and traffic overload is low (several milliseconds) and in fact the overhead is negligible.

X. Conclusion

SICS is a middlebox outsourcing framework that protects the private information of packet headers and middlebox rules. Compared with existing methods, SICS has several unique advantages including a stronger security guarantee, high-throughput processing, and support for quick updates. SICS assigns each packet a label identifying its matching behavior in a service chain and all middlebox processing in the cloud is based on labels. We use a prototype implementation and evaluation on VPC and local computers to demonstrate the feasibility, high performance, and efficiency of SICS.

Acknowledgement

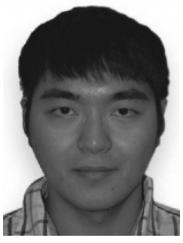
H. Wang, X. Li, Y. Zhao, Y. Yu, and C. Qian were partially supported by NSF Grants 1701681, 1717948, and 1932447. Y. Wang is partially supported by NSF grant CNS-1908020. We thank the anonymous reviewers for their comments.

References

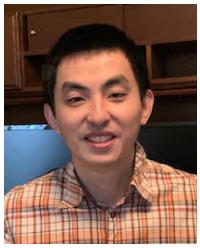
[1] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.
[2] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: enabling innovation in middlebox deployment," in *Proc. of ACM HotNets*, 2011.
[3] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," in *Proc. of ACM SIGCOMM*, 2012.
[4] R. Guerzoni *et al.*, "Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper," in *SDN and OpenFlow World Congress*, 2012.
[5] "WAN Optimization as-a-Service," http://www.routeviews.org.
[6] "Zscaler Cloud Firewall," https://www.zscaler.com/products/next-generation-firewall.
[7] "Palo Alto Networks," https://www.paloaltonetworks.com/products/secure-the-network/next-generation-firewall.
[8] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *Proc. of ACM HASP*, 2013.
[9] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, "Safebricks: Shielding network functions in the cloud," in *Proc. of USENIX NSDI*, 2018.
[10] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren, "Lightbox: Sgx-assisted secure network functions at near-native speed," *arXiv: 1706.06261v2*, 2018.
[11] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Security and Privacy (SP), 2015 IEEE Symposium on*, 2015.
[12] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep packet inspection over encrypted traffic," in *Proc. of ACM SIGCOMM*, 2015.
[13] X. Yuan, X. Wang, J. Lin, and C. Wang, "Privacy-preserving deep packet inspection in outsourced middleboxes," in *Proc. of IEEE INFOCOM*, 2016.
[14] Z. A. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using SDN," in *Proc. of ACM SIGCOMM*, 2013.
[15] Y. Zhang *et al.*, "StEERING: A Software-Defined Networking for Inline Service Chaining," in *Proc. of IEEE ICNP*, 2013.
[16] L. Melis, H. J. Asghar, E. De Cristofano, and M. A. Kaafar, "Private processing of outsourced network functions: Feasibility and constructions," in *Proc. of ACM SDN-NFV Security*, 2016.
[17] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, "Embark: Securely outsourcing middleboxes to the cloud," in *Proc. of USENIX NSDI*, 2016.
[18] H. J. Asghar, L. Melis, C. Soldani, E. De Cristofaro, M. A. Kaafar, and L. Mathy, "Splitbox: Toward efficient private network function virtualization," in *Proc. of ACM HotMiddlebox*, 2016.
[19] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. of ACM SIGCOMM*, 2012.
[20] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proc. of USENIX NSDI*, 2014.
[21] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proc. of USENIX NSDI*, 2013.
[22] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," in *Proc. of IEEE ICNP*, 2013.
[23] ——, "Scalable verification of networks with packet transformers using atomic predicates," *IEEE/ACM Transactions on Networking*, 2017.
[24] "Service Function Chaining," https://datatracker.ietf.org/wg/sfc.
[25] H. Wang, C. Qian, Y. Yu, H. Yang, and S. S. Lam, "Practical network-wide packet behavior identification by ap classifier," in *Proc. of ACM CoNEXT*, 2015.
[26] G. Gibb, H. Zeng, and N. McKeown, "Outsourcing network functionality," in *Proc. of ACM HotSDN*, 2012.
[27] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," in *Theory of cryptography*, 2005, pp. 325–341.
[28] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "Shieldbox: Secure middleboxes using shielded execution," in *Proc. of ACM SOSR*, 2018.
[29] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: A software-defined framework for developing, deploying, and managing network functions," in *Proc. of ACM SIGCOMM*, 2016.
[30] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr, and D. Kostić, "Snf: Synthesizing high performance nfv service chains," *PeerJ Computer Science*, 2016.
[31] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr, "Metron:NFV service chains at the true speed of the underlying hardware," in *Proc. of USENIX NSDI*, 2018.
[32] M. T. Goodrich and R. Tamassia, *Introduction to computer security*. Pearson, 2011.
[33] S. K. Fayazbakhsh, M. K. Reiter, and V. Sekar, "Verifiable network function outsourcing: requirements, challenges, and roadmap," in *Proc. of ACM HotMiddlebox*, 2013.
[34] X. Yuan, H. Duan, and C. Wang, "Bringing practical execution assurances to outsourced middleboxes," in *Proc. of IEEE ICNP*, 2016.

[35] "AT&T fined $ 25 million after call center employees stole customers," http://arstechnica.com/techpolicy/2015/04/att-fined-25-million-after-call-centeremployees-stole\\-customers-data/.
[36] "Radioshack sells customer data after settling with states." http://www.bloomberg.com/news/articles/2015-05-20/radioshackreceives-approval-to-sell-nameto-standard-general.
[37] "Chronology of data breaches." http://www.privacyrights.org/data-breach.
[38] "2015 data Breach Investigations Report." http://www.verizonenterprise.com/DBIR/2015/.
[39] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, 1986.
[40] H. Wang, C. Qian, Y. Yu, H. Yang, and S. S. Lam, "Practical network-wide packet behavior identification by ap classifier," *IEEE/ACM Transactions on Networking*, 2017.
[41] G. N. Purdy, *Linux iptables Pocket Reference: Firewalls, NAT & Accounting*, 2004.
[42] H. Pereira, A. Ribeiro, and P. Carvalho, "L7 classification and policing in the pfsense platform," *Atas da CRC*, 2009.
[43] "Haproxy-the reliable, high-performance tcp/http load balancer," http://haproxy.
[44] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer." in *Proc. of USENIX NSDI*, 2016.
[45] J. Frahim and O. Santos, *Cisco ASA: All-in-One Firewall, IPS, Anti-X, and VPN Adaptive Security Appliance*. Pearson Education, 2009.
[46] "Header Space Library and Netplumber," http://bitbucket.org/peymank/hassel-public/.
[47] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 350–361.
[48] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for NFV applications," in *Proc. of ACM SOSP*, 2015.
[49] P. Malacaria and J. Heusser, "Information theory and security: Quantitative information flow," in *Formal Methods for Quantitative Aspects of Programming Languages*. Springer, 2010, pp. 87–134.
[50] "Amazon Virtual Private Cloud." https://aws.amazon.com/vpc/?nc1=h_ls.
[51] "Dpdk. data plane development kit." https://www.dpdk.org/.
[52] E. Kohler, "The Click Modular Router," Ph.D. dissertation, Massachusetts Institute of Technology, 2000.
[53] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Proc. of ESA*, 2001.
[54] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. of ACM CoNEXT*, 2014.
[55] "Pktgen." http://pktgen.readthedocs.io/en/latest/.

Huazhe Wang (M'15) received the Ph.D. degree from the Department of Computer Science and Engineering, University of California, Santa Cruz in 2019. He received the B.Sc. degree from Beijing Jiaotong Univeristy in 2011 and the M.Sc. degree from Beijing Univeristy of Posts and Telecommunications in 2014. His research interests includes automated network validation and network security.



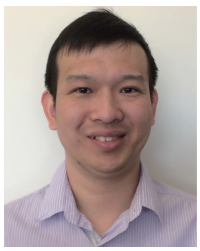
Xin Li (M'15) received the B.Eng. degree in telecommunication engineering from the University of Electronic Science and Technology of China, the M.S. degree in electrical engineering from the University of California Riverside, and the Ph.D. degree from the Department of Computer Science and Engineering, University of California Santa Cruz in 2018. His research interests include network security, software-defined networking, network functions virtualization, and the Internet of Things.



Yang Wang received the bachelor's and master's degrees in computer science and technology from Tsinghua University, in 2005 and 2008, respectively, and the doctorate degree in computer science from the University of Texas at Ausitn, in 2014. He is now an assistant professor in the Department of Computer Science and Engineering, Ohio State University. His current research interests include distributed systems, fault tolerance, and scalability.

Yu Zhao is a Ph.D. student at the University of Kentucky. He received his bachelor degree in Jilin University of information technology in 2009. He received his master degree in Changchun University of Science and Technology in 2012. In 2012, he joined the Research Center of Internet of Things, Third Research Institute of Ministry of Public Security located in Shanghai, China. His research interests include software testing, computer network, wireless sensor networks, cognitive radio networks, and signal processing. He is an IEEE and ACM member.

Ye Yu (M'13) received the B.Sc. degree from Beihang University and the doctorate degree in computer science from University of Kentucky, in 2013 and 2018, respectively. His research interests including data center networks and software defined networking.



Hongkun Yang (M'12) received the Ph.D. degree in the Department of Computer Science, University of Texas at Austin in 2015, where he is a recipient of the MCD Fellowship. He received the B.S.E. degree with Distinction and the M.S.E. degree from Tsinghua University in 2007 and 2010, respectively. His research interests include computer networks, protocol verification, network security, and formal methods. He has published research papers in a number of conferences and journals including IEEE ICNP, IEEE INFOCOM, IEEE Transactions on Mobile Computing. He is a student member of IEEE.

Chen Qian (M'08) is an Assistant Professor at the Department of Computer Engineering, University of California Santa Cruz. He received the B.Sc. degree from Nanjing University in 2006, the M.Phil. degree from the Hong Kong University of Science and Technology in 2008, and the Ph.D. degree from the University of Texas at Austin in 2013, all in Computer Science. His research interests include computer networking, network security, and Internet of Things. He has published more than 50 research papers in highly competitive conferences and journals. He is a member of IEEE and ACM.