On the Safety and Efficiency of Virtual Firewall Elasticity Control

Juan Deng[†*], Hongda Li[†*], Hongxin Hu[†], Kuang-Ching Wang[†], Gail-Joon Ahn[‡], Ziming Zhao[‡], and Wonkyu Han[‡]
[†] Clemson University
{jdeng, hongdal, hongxih, kwang}@clemson.edu
[‡]Arizona State University
{gahn, zzhao30, iamhwk}@asu.edu

***Abstract*—Traditional hardware-based firewall appliances are placed at fixed locations with fixed capacity. Such nature makes them difficult to protect today's prevailing virtualized environments. Two emerging networking paradigms, Network Function Virtualization (NFV) and Software-Defined Networking (SDN), offer the potential to address these limitations. NFV envisions to implement firewall function as software instance (a.k.a virtual firewall). Virtual firewalls provide great flexibility and elasticity, which are necessary to protect virtualized environments. In this paper, we propose to build an innovative virtual firewall controller, `VFW Controller`, to enable safe, efficient and cost-effective virtual firewall elasticity control. `VFW Controller` addresses four key challenges with respect to *semantic consistency*, *correct flow update*, *buffer overflow avoidance*, and *optimal scaling* in virtual firewall scaling. To demonstrate the feasibility of our approach, we implement the core components of `VFW Controller` on top of NFV and SDN environments. Our experimental results demonstrate that `VFW Controller` is efficient to provide safe elasticity control of virtual firewalls.**

I. Introduction

Firewall is the most critical and widely deployed network security function that protects networks against suspicious traffic and unauthorized access. Traditional hardware-based firewalls are often placed at fixed network entry points and have a constant capacity with respect to the maximum amount of traffic they can handle per time unit. Given such rigid nature, it is difficult to leverage hardware-based firewalls to protect today's prevailing virtualized environments. First, the perimeter of a network in virtualized environments becomes fluid, as VMs and applications may span across racks within a data center or even across multiple data centers, and they are often migrated for the purpose of flexible resource management and optimization [20]. Second, traffic volume in virtualized environments varies frequently and significantly [17] [18]. A study on network traffic characteristic in virtualized environments suggests that traffic volume depends on time-of-day and day-of-week, and the traffic volume variations are nearly an order of magnitude [17]. This leads to an *expensive* option of over-provisioning hardware-based firewalls with the capacity to process peak traffic load that occurs occasionally.

Two emerging networking paradigms, Network Function Virtualization (NFV) [6] and Software-Defined Networking (SDN) [39], push forward a new breed of firewalls, *virtual firewalls (VFW)*, which feature flexibility and elasticity, and are well suited to protect virtualized environments. NFV implements firewall function as software instance that can be created or destroyed quickly to handle traffic volume variations. For example, ClickOS [38], a high-performance and open-source NFV platform, can create a virtual instance in less than 30 milliseconds. SDN, recognized as complimentary technology to NFV [26], seamlessly provides dynamic traffic steering support toward flexible, on-demand placement of virtual firewalls. Given these benefits, major commercial virtualized environments (e.g., VMware vCloud [37], Amazon AWS [2] [9], VCE Vblock [11]) have recently started to embrace virtual firewalls. Virtual firewalls can also be used to protect traditional enterprise networks [47].

However, to fully take advantage of virtual firewall benefits, our study reveals that there are great challenges to enable virtual firewall elastic scaling. When a virtual firewall is overloaded due to a large amount of incoming traffic and needs to *scale out*, new instances via NFV are quickly created. Selective firewall rules and states (in case of stateful firewall) on the overloaded virtual firewall are migrated to new instances, and the corresponding traffic flow rules in SDN switches are updated alongside to redistribute traffic. When multiple virtual firewall instances are underloaded and need to *scale in*, some instances are destroyed, all firewall rules and states on them are migrated to remaining instances, and flow rules are also updated accordingly. The scaling of virtual firewalls must be *safe*, *efficient* and *optimal*. A *safe* scaling does not cause legal traffic to be dropped or illegal traffic to be allowed. An *efficient* scaling ensures that the latency overhead caused by scaling is bounded. An *optimal* scaling consumes minimum compute and network resources.

To achieve *safe*, *efficient*, and *optimal* virtual firewall scaling, the following key challenges should be addressed:

- **Semantic Consistency** The split and mergence of firewall rules in virtual firewall scaling must not change the semantics of security policies. Otherwise *safety* is violated (see §II B for examples). Keeping the semantic consistency after rounds of splits and

[*]The first two authors contribute equally to the paper.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-1891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23013

mergences is non-trivial, because firewall rules are often logically entangled with each other resulting in complicated rule dependencies. Sometimes rule dependencies are intentionally introduced by system administrators to obtain fewer firewall rules [29] [49].

- **Correct Flow Update** When migrating firewall rules and states to new/remaining firewall instances, network flow rules in SDN switches must be correctly updated to re-route corresponding traffic to new/remaining instances. Incorrect flow update may cause some traffic is missent to an instance that does not have the firewall rules intended for the traffic, hence violates *safety*. Finding correct flow update is difficult, since flow rules in an SDN switch may be dependent [31] [32] and the traffic space defined by the flow rules often does not exactly match that defined by the firewall rules.
- **Buffer Overflow Avoidance** A *safe* scaling also requires buffering in-flight traffic during a migration [27] [28]. In-flight traffic refers to the traffic that arrive at the source instance after the matching firewall rules and states have been migrated, or the traffic that arrive at the destination instance before corresponding firewall rules and states become ready. However, buffer space is not unlimited. We also observe that migration of different firewall rules incurs different amount of in-flight traffic. Therefore, care must be taken while selecting firewall rules to migrate so that buffer overflow [1] can be avoided.
- **Optimal Scaling** Compute and network resources for building virtual firewalls are neither unlimited nor free. Resource optimization is an important goal that needs to be achieved in virtualized environments. Creations of virtual firewalls consume compute resource, and flow rule updates are constrained by the limited capacity of the Ternary Content Addressable Memories (TCAMs) used for holding rules in each SDN switch. It is NP-hard to achieve optimization of resource usage during virtual firewall scaling.

In this paper, we propose a novel virtual firewall controller, `VFW Controller`, that enables *safe*, *efficient*, and *optimal* virtual firewall scaling. To address the challenge of *semantic consistency*, `VFW Controller` applies packet space analysis to identify intra-dependencies of firewall rules. A group-based migration strategy is applied to guarantee the semantic consistency. To find *correct flow update*, intra-dependencies of flow rules and inter-dependencies between firewall rules and flow rules are identified, which help locate the subset of flow rules to update and the correct update operations (e.g., change, insertion). To avoid *buffer overflow*, we model migration process and predict the amount of in-flight traffic generated during the migration. Firewall rules that may cause buffer overflow when migrating them are not selected to move. To achieve *optimal scaling-out* of virtual firewalls, `VFW Controller` adopts a three-step heuristic approach to minimize resource usage. To achieve *optimal scaling-in*, integer linear programming (ILP) is used to ensure maximum resources can be released. We design and implement the core components of `VFW Controller` on top of ClickOS [38]. Our evaluations in CloudLab [3] show that our `VFW Controller` provides efficient virtual firewall scaling control. To the best of our knowledge, `VFW Controller` provides the first solution for the safety and efficiency of virtual firewall elasticity control.

The rest of the paper is organized as follows. Section II presents the motivation and challenges of this paper. Section III gives an overview of `VFW Controller`. The following four sections present how `VFW Controller` guarantees semantic consistency, finds correct update, avoids buffer overflows, and achieves optimal scaling, respectively. We describe the implementation of `VFW Controller` and our experiments in Section VIII. Section IX discusses related work. Conclusion and future work are addressed in Section X.

II. Background and Challenges

A. New Networking Paradigms

Network functions (NFs) are systems that examine and modify packets or flows in a sophisticated fashion. Hardware-based NFs have become fundamental elements in enterprise networks [38] [48]. They are expensive, often vendor proprietary, difficult to manage, and inflexible with respect to location and capacity. These demerits hinder the evolution of network architecture and new service introduction. Network Function Virtualization (NFV) aims to overcome these limitations leveraging virtualization and cloud technologies [6]. NFV shifts NF from hardware appliances to virtualized instances running on standard high volume resource (servers, storage and switches). Built on cloud platforms for resource management, NFV shares resource across services and different customer base. In addition, NFV promises the benefits of low cost, management ease, scalability, openness, convenient service introduction, etc.

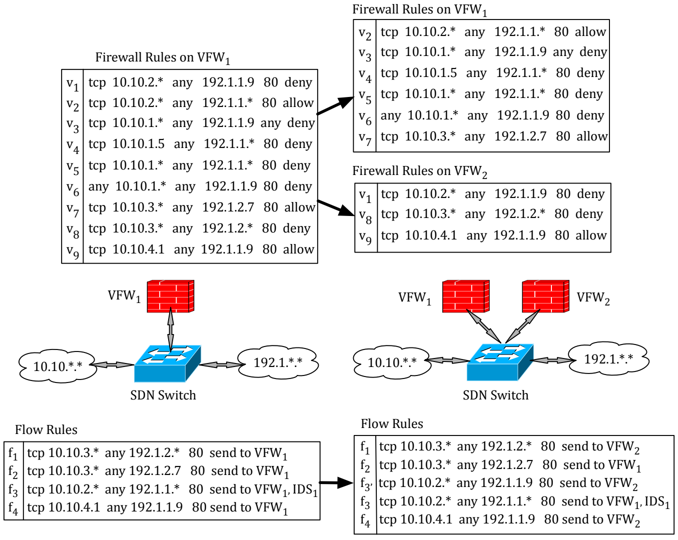
Traditional network devices have the control plane, which makes decision on traffic forwarding, and the data plane, which forwards traffic, tightly coupled. This makes them very difficult to implement network policies and introduce new services or protocols. Also, it is impossible to automatically respond to network faults or load changes [34], [41]. By decoupling forwarding hardware from control decisions, Software-Defined Networking (SDN) centralizes network intelligence in software-based controller, making network devices simple packet forwarding devices. Network devices can be programmed via an open interface, such as OpenFlow [39].

Given their benefits, NFV and SDN have recently attracted significant attentions from both academia and industry. A recent survey indicates that 97% of the major network operators plan to deploy SDN and 93% plan to deploy NFV [46]. Google, Amazon, AT&T, and Intel have announced their deployment plans [12]–[14], [19], [42]. NFV does not rely on SDN to exist, but the programmable feature of SDN greatly facilitate NFV [26] [45].

B. Challenges in Virtual Firewall Elasticity Control

Recent research efforts, notably Split/Merge [44] and OpenNF [28], have laid the groundwork for supporting the elastic scaling of a variety of virtualized network functions.

[1] In this paper, we use the term ‘buffer overflow’ to refer to the situation when buffer space is inadequate to buffer in-flight traffic, causing traffic drop.



Firewall Rules on VFW_1

v_1	tcp	10.10.2.*	any	192.1.1.9	80	deny
v_2	tcp	10.10.2.*	any	192.1.1.*	80	allow
v_3	tcp	10.10.1.*	any	192.1.1.9	any	deny
v_4	tcp	10.10.1.5	any	192.1.1.*	80	deny
v_5	tcp	10.10.1.*	any	192.1.1.*	80	deny
v_6	any	10.10.1.*	any	192.1.1.9	80	deny
v_7	tcp	10.10.3.*	any	192.1.2.7	80	allow
v_8	tcp	10.10.3.*	any	192.1.2.*	80	deny
v_9	tcp	10.10.4.1	any	192.1.1.9	80	allow

Firewall Rules on VFW_1

v_2	tcp	10.10.2.*	any	192.1.1.*	80	allow
v_3	tcp	10.10.1.*	any	192.1.1.9	any	deny
v_4	tcp	10.10.1.5	any	192.1.1.*	80	deny
v_5	tcp	10.10.1.*	any	192.1.1.*	80	deny
v_6	any	10.10.1.*	any	192.1.1.9	80	deny
v_7	tcp	10.10.3.*	any	192.1.2.7	80	allow

Firewall Rules on VFW_2

v_1	tcp	10.10.2.*	any	192.1.1.9	80	deny
v_8	tcp	10.10.3.*	any	192.1.2.*	80	deny
v_9	tcp	10.10.4.1	any	192.1.1.9	80	allow

Flow Rules

f_1	tcp	10.10.3.*	any	192.1.2.*	80	send to VFW_1
f_2	tcp	10.10.3.*	any	192.1.2.7	80	send to VFW_1
f_3	tcp	10.10.2.*	any	192.1.1.*	80	send to VFW_1, IDS_1
f_4	tcp	10.10.4.1	any	192.1.1.9	80	send to VFW_1

Flow Rules

f_1	tcp	10.10.3.*	any	192.1.2.*	80	send to VFW_2
f_2	tcp	10.10.3.*	any	192.1.2.7	80	send to VFW_1
$f_{3'}$	tcp	10.10.2.*	any	192.1.1.9	80	send to VFW_2
f_3	tcp	10.10.2.*	any	192.1.1.*	80	send to VFW_1, IDS_1
f_4	tcp	10.10.4.1	any	192.1.1.9	80	send to VFW_2

(a) Before scaling out.

(b) After scaling out.

Fig. 1: Example of virtual firewall scaling out. Note that a firewall rule r is a 6-tuple of $\{$protocol, source IP, source port, destination IP, destination port, action$\}$. A flow rule in an SDN switch may have more than 40 fields [39]. Here, we only show the fields that are pertinent to our illustration.

Both Split/Merge and OpenNF mainly focus on controlling the *state migration* of virtualized network functions. Especially, OpenNF provides APIs that help migrate given internal states of the source instance to the destination instance when scaling out an overloaded instance. The state migration mechanism provided by OpenNF satisfies two properties: loss-free and order-preserving. Loss-free property guarantees that all in-flight traffic generated during the state migration are buffered and processed. Order-preserving property guarantees that in-flight traffic are processed in the order of their arrival. Even though *state migration* is also a necessary operation in enabling the elastic scaling of virtual firewalls, more critical challenges are needed to be addressed in *virtual firewall* elasticity control.

When a virtual firewall is overloaded, one option for solving the overload is to *copy* the whole set of firewall rules on the overloaded firewall to each new instance and distribute traffic. However, the *copy* option has a major drawback of performance degradation. A firewall processes a packet by sequentially searching its rule space until the first-matching rule is found. Larger the firewall rule size is, larger the search overhead is and hence lower the processing capacity is. Existing research efforts have been devoted to compressing firewall rule set in order to improve firewall performance [35] [36]. We have also conducted experiments to quantify the effect of firewall rule size on virtual firewall performance (see § VIII-B1). Our experimental results show a linear decrease in virtual firewall processing capacity as the firewall rule size increases (see Figure 7). Therefore, we prefer to ***split*** firewall rules to deal with the overload, enabling each new instance to hold only a subset of the firewall rules and thus achieving higher processing capacity. Below we articulate new challenges in virtual firewall scaling.

1) ***Semantic Consistency:*** When splitting a set of firewall rules across multiple virtual firewall instances, care must be taken with respect to firewall rule dependencies so that the semantics of the rules are unchanged after the split. Changes in the semantics may lead to safety violations.

Figure 1 shows an example [2] where a virtual firewall, VFW_1, is scaled out. Firewall rule v_1 and v_2 on VFW_1 before scaling (Figure 1(a)) are dependent. Together they express a security policy dictating that "any host in network 10.10.2.* can access any *HTTP* server in network 192.168.1.*, except server 192.168.1.9." Consider a split example where v_1 and v_8 are migrated to VFW_2, and others stay put (Figure 1 (b)). After split, VFW_1 grants any host in 10.10.2.* access to 192.168.1.9, which should be denied. Thus, safety is violated. A remedy is to rely on the SDN switch to be aware of the dependency of v_1 and v_2 and route traffic from 10.10.2.* to 192.168.1.9 to VFW_2. The remedy, while seems plausible, is impractical for a large set of firewall rules with complicated dependencies. To see why, consider another example if we were to move dependent firewall rules v_3, v_4, v_5, and v_6 on VFW_1 to four different instances. It requires very complicated routing support. For every flow that matches more than one rule, the SDN switch must remember its first matching rule, in order to compensate the reckless split that disrespects the dependencies.

Overlooking rule dependencies may also create safety issues for later mergence. For example, suppose VFW_1 and VFW_2 in Figure 1 (b) are going to be merged back into one. Without care, the mergence may produce a rule order where v_2 precedes v_1, in which case it falsely grants access to *http* server 192.1.1.9, leading to safety violation. In reality, a firewall comprises hundreds of firewall rules with logical intertwinement [29] [40] [49]. And they may undergo a number of splits and mergences. This makes maintaining semantic consistency a challenge in virtual firewall scaling.

2) ***Correct Flow Update:*** When migrating firewall rules, flow rules in SDN switches are updated alongside. Finding the correct flow update is challenging due to a number of reasons.

First, an SDN switch generally comprises thousands of flow rules that are dependent [31]. Modification of a flow rule which has dependency relations with others is error-prone. Consider again the split example in Figure 1. Flow rules f_1 and f_2 are dependent. To account for the migration of firewall rule v_8 from VFW_1 to VFW_2, we search the flow rule space and find the first flow rule f_1 exactly matches v_8. Therefore, we change the action of f_1 to 'send to VFW_2'. Because f_1 overshadows f_2, f_2 will never take effect. In this case, when traffic matching v_7 arrives at the switch, f_1 applies and the traffic is wrongly sent to VFW_2. But v_7 is actually on VFW_1.

Second, a flow rule in an SDN switch may be used by more than one applications. Update of flows to account for firewall rule migration may undesirably change the routing intentions of other applications. For example, f_3 in Figure 1 (a) serves both firewall and IDS. To account for the migration of firewall rule v_1, f_3 needs to be updated. The correct update is to insert f'_3 right before f_3 (Figure 1 (b)). Together they express the correct routing for migrating v_1. But this changes routing intentions of IDS_1, because traffic matching "tcp 10.10.2.* any

[2] For the elucidation purpose, we use stateless firewalls here. The same challenges also apply to *stateful* firewalls.

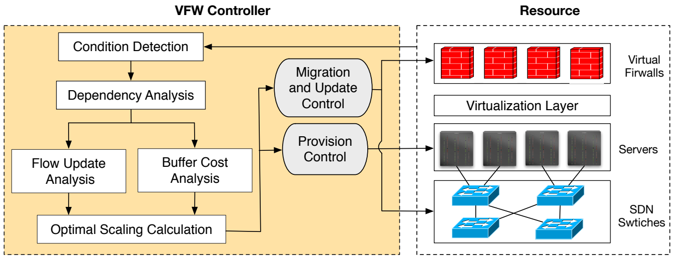


Fig. 2: `VFW Controller` components and workflow.

192.1.1.9" is not sent to IDS$_1$, as intended, instead it is sent to VFW$_2$.

Third, flow rules do not always exactly match firewall rules. This requires split of flow rules or insertion of new flow rules do handle traffic steering. For example, we inserted f_3' for migrating v_1. The split or insertion operation is non-trivial and also increases the number of flow rules in SDN switches, which are limited by the number of flow rules they can support [3].

*3) **Buffer Overflow Avoidance**:* A safe migration must be loss-free, meaning in-flight traffic must be buffered [28]. Existing systems, such as Split/Merge and OpenNF, buffer all in-fight traffic at the central controller, which could lead to serious scalability issue. In particular, they assume *infinite* buffer space at the controller and ignore the potential problem of *buffer overflow*. Therefore, a preferable migration control method should adopt distributed buffering that buffers in-flight traffic at each destination instance, as a way to lower the risk of buffer overflow [27]. In addition, we observe that different firewall rules incur different amounts of in-flight traffic. For example, migrating a firewall rule with hit rate of 0 [4] does not incur any in-flight traffic, while migrating a firewall rule with high hit rate requires large buffer space. To avoid buffer overflow, we need a prior calculation of the amount of in-flight traffic for each firewall rule, and design a firewall rule selection mechanism so that only firewall rules that do not cause buffer overflow are selected for the migration.

*4) **Optimal Scaling**:* Resources for creating virtual firewalls are valuable and limited, hence must be optimally used. When a virtual firewall is overloaded, solving the overload with a minimum number of new virtual firewall instances is preferable. When multiple virtual firewall instances are underloaded, a merge operation should kill as many virtual firewall instances as possible. Also, updating flow rules may entail insertions of new flow rules, which consume limited TCAM space in SDN switches as discussed above. Some firewall rule is matched by only one flow rule, migration of such firewall rule only needs to update one flow rule and doesn't need to add new flow rules. However, since some wildcard firewall rules can be matched by multiple flow rules, the migrations of those firewall rules involve modifying multiple flow rules and possibly creating new flow rules. Therefore, the firewall rules that incur high update costs should be given low priority to be migrated.

III. VFW Controller Overview

The components and workflow of `VFW Controller` are shown in Figure 2. `VFW Controller` monitors each virtual firewall and detects traffic overload and underload conditions. Once a condition is detected, `VFW Controller` first performs *Dependency Analysis*, *Flow Update Analysis*, and *Buffer Cost Analysis*. Those analysis results are utilized by *Optimal Scaling Calculation*. Then, *Provision Control* and *Migration and Update Control* interact with the compute and network resources and execute virtual firewall scaling.

Dependency Analysis (§IV) in `VFW Controller` identifies three dependency relations: (1) the dependency relations of firewall rules on the virtual firewalls; (2) the dependency relations of the flow rules in SDN switches; and (3) the inter-dependency relations between the firewall rules and the flow rules. Understanding the dependency relations of the firewall rules is critical to ensure the semantic consistency after scaling. The dependency relations of the flow rules and the inter-dependency relations serve to find the flow rules to be updated. *Dependency Analysis* divides both firewall rules and flow rules into groups based on their dependency relations. *Flow Update Analysis* (§V) determines the correct update of flow rules and the update cost for each firewall rule group. *Buffer Cost Analysis* (§VI) predicts the amount of in-flight traffic generated by the migration of each firewall rule group. This prediction is necessary to avoid buffer overflow.

Optimal Scaling Calculation (§VII) component considers previous analyses and uses a three-step heuristic approach to determine, in the case of overload, (1) the minimum number of new instances to be created; (2) selective firewall rule groups to be migrated to each new instance; and (3) flow rules to be updated. This approach also achieves minimum update cost and no buffer overflow. In the case of underload, *Optimal Scaling Calculation* component uses an ILP approach to determine (1) which instances are to be killed among all underloaded instances; (2) how to redistribute firewall rule groups; and (3) corresponding flow rule updates. The ILP approach guarantees that the most resources are released after the mergence of instances.

Provision Control creates or deletes instances according the calculation results from *Optimal Scaling Calculation* component. *Migration and Update Control* coordinates the migration of firewall rules and states, and flow rule updates.

IV. Dependency Analysis and Semantic Consistency

Firewall policies used in enterprise networks are known to be complex due to their large rule sizes [5] and complicated rule dependencies [24] [29]. Flow rules in SDN switches are complex as well [31] [32]. In this section, we first analyze the *intra-dependency* relations of firewall rules and flow rules respectively, and the *inter-dependency* relations between firewall rules and flow rules. We then discuss our approach for maintaining semantic consistency in virtual firewall scaling.

[3] These rules are in TCAM and an SDN switch can only support a few thousand rules (e.g., 1.5k TCAM entries in 5406zl switch [23]).

[4] Hit rate of a firewall rule is the rate that rule is matched. A 0 hit rate means there is no flow matching the rule.

[5] A recent study uncovered that the average number of firewall rules in an enterprise network is 793 [22].

A. Dependency Relation

We start by giving the following definitions. Note that the definitions of *direct dependency* and *indirect dependency* apply to both firewall rules and flow rules.

Definition 1 (Packet space). *Packet space of a rule r, denoted as $PS(r)$, is defined as a 5-dimensional hyperspace with dimensions being protocol, source IP, source port, destination IP, destination port.*

For a firewall rule, which is generally represented as a 6-tuple of {protocol, source IP, source port, destination IP, destination port, action} with either 'deny' or 'allow' action, finding its packet space is straightforward and just needs to use the first 5 fields of the firewall rule. For a flow rule, we must consider various `Set-Field` actions, which can rewrite the values of respective header fields in packets [15]. For example, if a flow rule f has "protocol = tcp, source IP = 10.10.3.*, source port = any, destination IP = 192.1.2.7, destination port = any" without a `Set-Field` action, its packet space can be represented with these five fields as $< tcp, 10.10.3.*, any, 192.1.2.7, any >$. However, if a flow rule has `Set-Field` actions, which rewrite some header fields, its packet space must be represented using the rewritten values of fields. For instance, if the above f has a `Set-Field` action, "Set destination IP = 192.1.2.10", its packet space should be represented as $< tcp, 10.10.3.*, any, 192.1.2.10, any >$.

Definition 2 (Direct dependency). *Two rules r_i and r_j in a rule set $\mathbb{R}$ are directly dependent iff $PS(r_i) \cap PS(r_j) \neq \emptyset$, where $PS(r_i)$ is the packet space defined by r_i, and $PS(r_j)$ is the packet space defined by r_j.*

Definition 3 (Indirect dependency). *Two rules r_i and r_j in a rule set $\mathbb{R}$ are indirectly dependent iff $PS(r_i) \cap PS(r_j) = \emptyset$ and there exists a subset $R \subseteq \mathbb{R} \backslash \{r_i, r_j\}$ such that $PS(r_i) \cap PS(R) \neq \emptyset$ and $PS(r_j) \cap PS(R) \neq \emptyset$.*

Note that indirect dependency is associative, that is, if rule r_i and r_j are indirectly dependent, and r_j and r_l are indirectly dependent, then r_i and r_l are indirectly independent.

`VFW Controller` uses Algorithm 1 to partition a rule set R to disjoint groups. Rules inside a group are in the same order as they are in R, and are either directly or indirectly dependent. Across groups, there is no dependency relation. The inputs of Algorithm 1 are the rule set R to be partitioned and a set $\mathbb{G}$ that is used to store rule groups and initially set to be empty. Algorithm 1 sequentially examines each rule $r \in R$ against each existing rule group $G \in \mathbb{G}$. We use a set Γ to store all the rules in R that precede r and have dependency relation with r, and Γ is initially set to be empty. If $\mathbb{G}$ is empty (i.e. r is the first rule in R), then we append r to Γ, add Γ as a new group to $\mathbb{G}$, and move the next rule in R. Otherwise, if there exists a rule $r_i \in G$ that has dependency relation with r, then *all* the rules in G are appended to Γ, and group G is removed from $\mathbb{G}$. After all existing groups in $\mathbb{G}$ have been checked, all the rules in R that precede r and have dependency relation with r are now stored in Γ. We append rule to r to Γ. At last Γ is added as a new group to $\mathbb{G}$. After R has been iterated through, all the rule groups are stored in $\mathbb{G}$. Applying Algorithm 1 to a set of firewall rules on a virtual firewall produces a set of firewall rule groups, which we denote as $\mathbb{V} = \{V_1, ..., V_m\}$ throughout this paper.

Algorithm 1: Partitioning of a rule set to disjoint groups.

```
Input: $R$, a set of ordered rules to be partitioned;
       $\mathbb{G} = \emptyset$, a set, initially set to be empty, to store
       rule groups.
Output: $\mathbb{G}$, the set of rule groups;
foreach $r \in R$ do
    $\Gamma = \emptyset$; /* A set to store preceding rules that has
        dependency relation with $r$ . Initially $\emptyset$.      */
    if $\mathbb{G} = \emptyset$ then
        /* That is to say $r$ is the first rule in $R$    */
        $\Gamma.Append(r)$;
    else
        $PS(r) \longleftarrow PacketSpace(r)$;
        foreach $G \in \mathbb{G}$ do
            foreach $r_i \in G$ do
                $PS(r_i) \longleftarrow PacketSpace(r_i)$;
                if $PS(r) \cap PS(r_i) \neq \emptyset$ then
                    $\Gamma.Append(G)$;/* Sequentially append
                        each rule in group $G$ to $\Gamma$.    */
                    $\mathbb{G}.Remove(G)$;/* Remove group $G$ from
                        $\mathbb{G}$.                          */
                    break;
        /* Now all the rules that precede $r$ and have
            dependency relation with $r$ are stored in $\Gamma$.
            */
        $\Gamma.Append(r)$; /* Append $r$ to $\Gamma$.            */
    $\mathbb{G}.Add(\Gamma)$; /* Add $\Gamma$ as a new group to $\mathbb{G}$.    */
return $\mathbb{G}$;
```

Applying Algorithm 1 to a set of flow rules on an SDN switch produces a set of flow rule groups, which we denote as $\mathbb{F} = \{F_1, ..., F_n\}$ throughout this paper. We identify the relation between a firewall rule group V and a flow rule group F is one of the following:

- Independency *iff* $PS(V) \cap PS(F) = \emptyset$. We denote independency as $V \overset{ind}{\text{—}} F$.
- Congruence *iff* $PS(V) = PS(F)$. We denote congruence as $V \overset{con}{\text{—}} F$.
- Superspace *iff* $PS(V) \supset PS(F)$. We denote superspace as $V \overset{sup}{\text{—}} F$.
- Subspace *iff* $PS(V) \subset PS(F)$. We denote subspace as $V \overset{sub}{\text{—}} F$.
- Intersection *iff* $PS(V) \cap PS(F) \subset PS(V)$ and $PS(V) \cap PS(F) \subset PS(F)$. We denote intersection as $V \overset{int}{\text{—}} F$.

Definition 4 (Inter-dependency). *A firewall rule group V and a flow rule group F are inter-dependent if $PS(V) \cap PS(F) \neq \emptyset$.*

When V and F are inter-dependent, they are related by one of *Congruence*, *Superspace*, *Subspace* and *Intersection* relation.

B. Association and Class

We next introduce association relation between two firewall rule groups, and the concept of firewall rule class. We will show in §V that class-based migration serves to achieve less update cost.

Definition 5 (Direct association). *Two firewall rule groups V_i and V_j in a firewall rule set $\mathbb{V}$ are directly associated iff there*

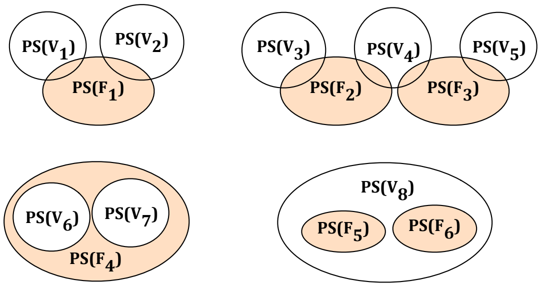


Fig. 3: Example of firewall rule associations and classes.

Algorithm 2: Classification of firewall rule groups to classes.

```
Input: $\mathbb{V} = \{V_1, ..., V_m\}$, a set of firewall rule groups; $\mathbb{F} = \{F_1, ..., F_n\}$,
        a set of flow groups;
        $\mathbb{C} = \emptyset$, a set of firewall rule classes which is initially set
        to be empty;
Output: $\mathbb{C}$;
/* Identify direct associations.                                  */
foreach $F \in \mathbb{F}$ do
    $\bar{V} = \emptyset$; /* A set to store firewall rule groups that
        are directed associated via $F$, initially set to
        be empty;                                                   */
    foreach $V \in \mathbb{V}$ do
        if $V$ and $F$ are inter-dependent then
            $\bar{V}.Append(V)$;/* Each firewall rule in $V$ is
                sequentially appended to $\bar{V}$.                 */
    if $\bar{V} \neq \emptyset$ then
        $\mathbb{C}.Add(\bar{V})$;/* $\bar{V}$ is added to $\mathbb{C}$ as a class.   */
/* Identify indirect associations. Any two classes that
    intersect are united to make a bigger class.                    */
foreach $\bar{V}_i \in \mathbb{C}$ and $\bar{V}_j \in \mathbb{C}$ do
    if $\bar{V}_j \cap \bar{V}_j \neq \emptyset$ then
        $\bar{V}_i.Append(\bar{V}_j)$; /* Each firewall rule in $\bar{V}_j$ is
            sequentially appended to $\bar{V}_i$.                   */
        $\mathbb{C}.Remove(\bar{V}_j)$;
return $\mathbb{C}$;
```

exists a flow rule group F in a flow rule set $\mathbb{F}$ such that V_i and F are inter-dependent, and V_j and F are inter-dependent. We use the notation $V_i \underline{\text{dir-asso}} V_j$ to denote V_i and V_j are directly associated.

Definition 6 (Indirect association). *Two firewall rule groups V_i and V_j in a firewall rule set $\mathbb{V}$ are indirectly associated iff they are not directly associated and there exists $\bar{V} \subseteq \mathbb{V} \backslash \{V_i, V_j\}$, such that V_i and $\bar{V}$ are directly associated, and V_j and $\bar{V}$ are directly associated.*

Definition 7 (Firewall Rule Class). *A firewall rule class is the union of firewall rule groups that are directly or indirectly associated.*

Figure 3 demonstrates association relation and firewall rule class using an example where the firewall rule set is $\mathbb{V} = \{V_1, ..., V_8\}$ and the flow rule set is $\mathbb{F} = \{F_1, ..., F_6\}$. In the example,

- Direct association: $V_1 \underline{\text{dir-asso}} V_2$, $V_3 \underline{\text{dir-asso}} V_4$, $V_4 \underline{\text{dir-asso}} V_5$, $V_6 \underline{\text{dir-asso}} V_7$
- Indirect association: $V_3 \underline{\text{ind-asso}} V_5$

Based on the association relations, the firewall rule set $\mathbb{V} = \{V_1, ..., V_8\}$ is further divided into four classes $\mathbb{V} = \{\bar{V}_1, ..., \bar{V}_4\}$, where class $\bar{V}_1 = V_1 \cup V_2$, $\bar{V}_2 = V_3 \cup V_4 \cup V_5$, $\bar{V}_3 = V_6 \cup V_7$, $\bar{V}_4 = V_8$.

`VFW Controller` uses Algorithm 2 to classify firewall rule groups $\mathbb{V} = \{V_1, ..., V_m\}$ into classes $\mathbb{V} = \{\bar{V}_1, ..., \bar{V}_{\bar{m}}\}$. Firewall rule groups inside a class are either directly or indirectly associated. There is no association across classes. The inputs of Algorithm 2 are $\mathbb{V}$ (the set of firewall groups to be classified), $\mathbb{F}$ (the set of flow rule groups of an SDN switch), and $\mathbb{C}$ (a set to store firewall rule classes). The algorithm has 2 steps. First, it identifies all the direct association relations. For each flow rule group $F \in \mathbb{F}$, the algorithm finds the firewall rule groups that are directly associated via F, unites these firewall rule groups together to make a class, and adds this class to $\mathbb{C}$. Now each class in $\mathbb{C}$ contains firewall rule groups that are directly associated. Take for example the firewall rule set and flow rule set in Figure 3, the first step will generate 6 classes: $\bar{V}_1' = V_1 \cup V_2$, $\bar{V}_2' = V_3 \cup V_4$, $\bar{V}_3' = V_4 \cup V_5$, $\bar{V}_4' = V_6 \cup V_7$, $\bar{V}_5' = V_8$, and $\bar{V}_6' = V_8$. Second, the algorithm unites the classes that intersect. This is to identify the indirect associations. For example as $\bar{V}_1'$ and $\bar{V}_2'$ both have V_4, they will be united. In this way, the indirect association between V_3 and V_5 are identified. Similarly $\bar{V}_5'$ and $\bar{V}_6'$ will be also united. So the second step will generate 4 classes: $\bar{V}_1' = V_1 \cup V_2$, $\bar{V}_2' = V_3 \cup V_4 \cup V_5$, $\bar{V}_4' = V_6 \cup V_7$, $\bar{V}_5' = V_8$.

C. Semantic Consistency

Semantic consistency of firewall rules must be maintained in virtual firewall scaling, otherwise safety violations may occur. Two causes lead to semantic inconsistency (see §II for examples). First, firewall rules that are dependent are split across multiple virtual firewall instances. Second, the order of firewall rules that are dependent is changed after migration. Therefore, we propose the following group-based strategy to maintain semantic consistency in `VFW Controller`:

Group-Based Migration Strategy: *To guarantee semantic consistency, firewall rules in a group are migrated to the same destination virtual firewall instance and are in the same order as they are in the source virtual firewall instance. The destination virtual firewall instance can only start to process traffic matching rules in a group until all the rules and flow states associated with the group are ready on the destination instance.*

We have studied a number of *real-world* firewall policies (see § VIII-B2) and found that rule dependencies are common in firewalls policies. Hence, dependency analysis is necessary in `VFW Controller`. We also found that it is a very rare case in reality that a firewall policy only contains one big group. Therefore, we ignore the discussion of such a special case in this paper.

V. Flow Update Analysis

In this section, we analyze how to update flow rules in SDN switches to provide the required traffic steering support for the migration of a firewall rule group. We identify a set of necessary update operations and define update cost. Strategy to reduce update cost is presented as well.

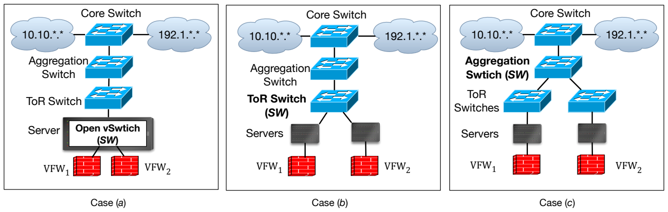


Fig. 4: *SW* in three different cases with respect to a classic 3-tire cloud architecture.

A. Update Operations

Let $V_i \in \mathbb{V}$ be the firewall rule group to be migrated from a source virtual firewall VFW$_1$ to a destination virtual firewall VFW$_2$. The key SDN switch is the last SDN switch, through which all the traffic matching V_i will pass before diverging on their paths to reach VFW$_1$ and VFW$_2$. In what follows, we denote such key SDN switch as *SW*. Figure 4 shows *SW* switches in different cases: (*a*) VFW$_1$ and VFW$_2$ are on the same server; (*b*) VFW$_1$ and VFW$_2$ are in different servers on the same rack; and (*c*) VFW$_1$ and VFW$_2$ are in different racks. In case (*a*), the *SW* is an open vSwitch on the server. In case (*b*), the *SW* is the Top-of-Rack (ToR) switch that connects the two servers. In case (*c*), the *SW* is the Aggregation Switch that connects the two racks. We consider case (*a*) will be the most common, as it introduces the least traffic overload and update efforts. Only when two virtual firewalls cannot be placed on the same sever due to resource inadequacy, they will be placed on separate servers.

To update the SDN switches on the path from *SW* to VFW$_1$, the existing flow rules that route the matching traffic to VFW$_1$ will be deleted. To update the SDN switches on the path from *SW* to VFW$_2$, new flow rules to route the matching traffic to VFW$_2$ will be added. The updates on *SW* is, however, less straightforward, hence we focus on the update of *SW* in this paper. `VFW Controller` tracks *SWs*, leveraging the capability provided by SDN where a central controller maintains a panoramic view of the entire network.

Let $\mathbb{F} = \{F_1, ..., F_n\}$ be the set of flow groups (obtained using the partition algorithm in §IV) on *SW*. To find the updates on $\mathbb{F}$, `VFW Controller` iterates through $\mathbb{F} = \{F_1, ..., F_n\}$ sequentially, and compares the inter-dependency relation between V_i and each $F_j \in \mathbb{F}$ to determine the updates. We identify two types of update operations: CHANGE and INSERT.

- If $V_i \overset{ind}{\longrightarrow} F_j$, no update is required.
- If $V_i \overset{con}{\longrightarrow} F_j$ or $V_i \overset{sup}{\longrightarrow} F_j$, only CHANGE operation is required. For every flow rule $f \in F_j$, if its forwarding actions contain 'send to VFW$_1$', the forwarding action is changed to 'send to VFW$_2$'. Note that the same flow rule may contain forwarding actions for the routing intentions of other applications (see the example flow rule f_3 in Figure 1). Flow update should not change any of those actions.
- If $V_i \overset{sub}{\longrightarrow} F_j$ or $V_i \overset{int}{\longrightarrow} F_j$, both CHANGE and INSERT operations are required. Comparisons between $PS(v)$, the packet space defined by each $v \in V_i$, and $PS(f)$, the packet space defined by each $f \in F_j$, will be performed in order to find the correct update.
  (1) If $PS(v) \cap PS(f) = \emptyset$, f needs no update.
  (2) If $PS(v) \supseteq PS(f)$, then CHANGE operation is performed to change the 'send to VFW$_1$' action of f to 'send to VFW$_2$'.
  (3) If $PS(v) \subset PS(f)$, then INSERT operation is performed. A new flow rule f' is inserted right before f to express that traffic matching v is sent to VFW$_2$. Take the v and f below for example.

v	tcp	10.10.2.*	*	192.1.1.9	80	deny
f	*	10.10.2.*	*	192.1.1.*	80	send to VFW$_1$ and IDS$_1$

  Then f' below is inserted right before f.

f'	tcp	10.10.2.*	*	192.1.1.9	80	send to VFW$_2$ and IDS$_1$

  (4) If $PS(v) \cap PS(f) \subset PS(v)$ and $PS(v) \cap PS(f) \subset PS(f)$, INSERT operation is performed. A new flow rule f' is inserted right before f. Each field of f' is the same as f, except that the protocol, source IP, source port, destination IP, destination port fields of f' are the intersection of the respective fields of v and f, and the forwarding action of f' is 'send to VFW$_2$'. For example, for the following v and f

v	*	10.10.2.*	*	192.1.1.9	80	deny
f	*	10.10.2.1	*	192.1.1.*	80	send to VFW$_1$ and IDS$_1$

  the new flow rule f' is:

f'	*	10.10.2.1	*	192.1.1.9	80	send to VFW$_2$ and IDS$_1$

After all the groups in $\mathbb{F} = \{F_1, ..., F_n\}$ have been examined, `VFW Controller` obtains the update of *SW* to support the migration of V_i. When comparing V_i with each F_j as above, `VFW Controller` also keeps track of the number of new flow rules that are INSERTed. In what follows we denote that number as α_{ij}.

B. Update Cost and Cost Reduction

As presented above, updates of the flow rules on *SW* consist of INSERT and CHANGE operations. Each INSERT operation causes an increase in the number of flow rules in the *SW* switch. The TCAM space for holding flow rules in an SDN switch is valuable and limited, therefore increases in the size of flow rules must be tracked. We define the *update cost* of firewall rule group V_i as follows.

Definition 8 (Update Cost). *The update cost of a firewall rule group is the total number of new flow rules inserted in SW during its update to support the migration of V_i.*

Let γ_i be the update cost of V_i, then we have

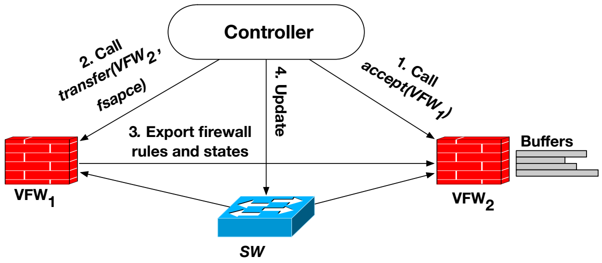

Controller
2. Call transfer(VFW 2, fsapce)
1. Call accept(VFW 1)
4. Update
3. Export firewall rules and states
Buffers
VFW1
VFW2
SW

Fig. 5: Workflow of migration control.

$$\gamma_i = \sum_{j=1}^{n} \alpha_{ij}$$

Given a large number of virtual firewalls in a virtualized environment and the fact that constant firewall rule migrations are expected, the total update cost must be reduced. There are two ways to reduce the total update cost:

- Select firewall rule groups that cause smaller update cost for migration.
- Analyze the relationships between firewall rule class and flow rule group. The comparison we have performed in §V-A is pairwise between a firewall rule group V_i and a flow group F_j.

We observed that new flow rules are inserted only when $V_i \underline{\,sub\,} F_j$ or $V_i \underline{\,int\,} F_j$. If there is a class $\bar{V}$ that consists of V_i and other firewall rule groups and $\bar{V} \underline{\,sup\,} F_j$, then no new flow rules will be inserted. To see why, consider a simple case as follows. Two firewall rule groups V_1 and V_2 are to be migrated. $F_j \in \mathbb{F}$ is one flow rule group in *SW* and we have $V_1 \underline{\,int\,} F_j$ and $V_2 \underline{\,int\,} F_j$, and $V_1 \cup V_2 \underline{\,sup\,} F_j$. Both INSERT and CHANGE operations are required for the migration of V_1 and V_2. Let α_{1j} be the number of new flow rules that will be inserted to F_j in order to support the migration of V_1, and α_{2j} be the number of new flow rules needed in order to support the migration of V_2. Thus the total number of new flow rules to be inserted to F_j is $\alpha_{1j} + \alpha_{2j}$. Let $\bar{V} = V_1 \cup V_2$, because $\bar{V} \underline{\,sup\,} F_j$, then no INSERT operation is required to support the migration of $\bar{V}$, hence the total number of new flow rules to be inserted is 0. Based on such observation, we propose the following class-based migration strategy in `VFW Controller` to reduce the total update cost:

Class-Based Migration Strategy: *If the update cost of migrating a class* $\bar{V} = \cup_{i=1}^{m'} V_i$ *is smaller than the sum of the update cost of each* V_i *that constitutes* $\bar{V}$*, then* $\bar{V}$ *is migrated as a big group to the same destination virtual firewall to reduce update cost.*

VI. Buffer Cost Analysis

Migration of a firewall rule group V_i from a source virtual firewall VFW$_1$ to a destination virtual firewall VFW$_2$ consists of the following key procedures: moving V_i and corresponding flow states on VFW$_1$ to VFW$_2$, installing of V_i and the states on VFW$_2$, and updating the flow rules in the involved SDN switches to reroute traffic matching V_i to VFW$_2$, and deleting V_i and the states on VFW$_1$.

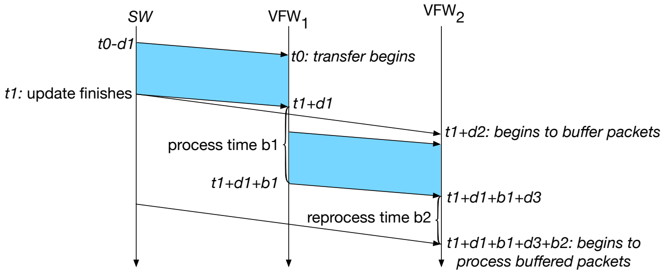


Fig. 6: Packet paths during the migration.

During the migration, in-flight traffic will be generated and need to be buffered until the installation completes. Then, in-flight traffic will be flushed to VFW$_2$ for processing. Prior knowledge on the amount of in-flight traffic is crucial to avoid buffer overflow. We observed that the migration of each firewall rule (group) incurs different amount of in-flight traffic. Based on the prior knowledge, we can select firewall rule groups to migrate in order to avoid overflowing the buffers of the destination virtual firewalls.

We define the buffer cost of a firewall rule group as follows:

Definition 9 (Buffer Cost). *The buffer cost of a firewall rule group is the amount of in-flight traffic that are generated during the migration of the group.*

We next present the migration control mechanism used in `VFW Controller`, followed by how to calculate buffer cost. The challenge is to model the migration control and quantify the time period from the start of buffering to the release of the buffered traffic. Factors affecting the time period include delays introduced by end-host (OS, compute resource assigned to a virtual firewall, etc.), which depend on cloud infrastructure.

As shown in Figure 5, `VFW Controller` adopts distributed buffering mechanism derived from [27] to buffer in-flight traffic at each destination instance. First, the central controller invokes API function *accept(VFW$_1$)* on VFW$_2$ to inform on the upcoming export of firewall rules and flow states from VFW$_1$. Second, the controller invokes another API function *transfer(VFW$_2$, fspace)* to instruct VFW$_1$ to transfer the firewall rules and the flow states defined in *fspace* to VFW$_2$. Then VFW$_1$ transfers the specified firewall rules and states to VFW$_2$. After receiving them, VFW$_2$ installs them immediately. The controller waits for the installation to be completed and issues updates to *SW* switch. Before the update finishes, *SW* keeps sending traffic to VFW$_1$, and VFW$_1$ continues processing these traffic. If any traffic, which VFW$_1$ processes *after* the transfer of the firewall rules and flow states, causes the state of a flow to evolve, then the traffic is sent by VFW$_1$ to VFW$_2$. VFW$_2$ reprocesses this traffic to bring its state up to date. After the updates on *SW* complete, VFW$_2$ receives traffic directly from *SW*. The traffic directly from *SW* must be buffered at VFW$_2$. VFW$_2$ can only start to process the buffered traffic until it has received and processed all the traffic from VFW$_1$. This is to ensure that VFW$_2$ has the most updated states before process the buffered traffic.

A. *Buffer Cost Calculation*

Figure 6 shows the packet paths during the migration of V_i. In Figure 6,

- $t0$: is the time that VFW$_1$ starts to transfer the firewall rules and flow states specified in *fspace*;
- $t1$: is the time that *SW* finishes the update;
- $d1$: is the transmission delay between *SW* and VFW$_1$;
- $d2$: is the transmission delay between *SW* and VFW$_2$;
- $d3$: the transmission delay between VFW$_1$ and VFW$_2$;
- $b1$: is the average time that VFW$_1$ spends processing a packet; and
- $b2$: is the average time that VFW$_2$ spends processing a packet.

At ***SW***, the traffic matching V_i that arrives before $t1$ is sent to VFW$_1$, and after $t1$ it is sent to VFW$_2$. Let Γ be a set comprising the matching traffic that arrives during $(t0-d1, t1)$. Γ is shown as the blue part in Figure 6. Traffic in Γ is sent to VFW$_1$.

At **VFW**$_1$, the firewall rules and flow states defined in *fspace* are sent to VFW$_2$. Traffic in Γ starts to arrive after $t0$, and the last packet in Γ arrives before $t1 + d1$. VFW$_1$ processes all the traffic in Γ. VFW$_1$ finishes processing Γ before $t1+d1+b1$. If any traffic in Γ whose processing causes the state of a flow to evolve, VFW$_1$ will send the traffic to VFW$_2$ for reprocessing in order to keep state consistency. For generality, we assume all the traffic in Γ are sent to VFW$_2$ for reprocessing. This assumption will lead to an estimate of buffer cost that is larger than the actual cost in some cases. Given that we want to avoid buffer overflow, over-estimates are preferable.

At **VFW**$_2$, traffic directly sent from *SW* starts to arrive after $t1+d2$. However, the traffic directly from *SW* must be buffered until VFW$_2$ has received and reprocessed all the traffic in Γ. The last packet in Γ arrives at VFW$_2$ before $t1+d1+b1+d3$ and it is processed before $t1 + d1 + b1 + d3 + b2$. Therefore, traffic that is directly sent from *SW* and arrives at VFW$_2$ during $(t1 + d2, t1 + d1 + b1 + d3 + b2)$ is buffered.

Suppose there are k_i flows matching V_i and the rate (bits per second) of flow j is λ_j. Then we estimate the buffer cost of V_i as

$$\begin{aligned}\beta_i &= (\sum_{j=1}^{k_i} \lambda_j) \times \{(t1 + d1 + b1 + d3 + b2) - (t1 + d2)\} \\ &= (\sum_{j=1}^{k_i} \lambda_j) \times (d1 + b1 + d3 + b2 - d2)\end{aligned} \tag{1}$$

VFW Controller can leverage the capability of SDN to conveniently obtain λ_j, $d1$, $d2$ and $d3$. *SW* periodically reports the statistics of flows to VFW Controller, which are used to obtain λ_j. $d1$, $d2$ and $d3$ are transmission delays, which can be measured in practice and collected by VFW Controller. $b1$ and $b2$ depend on the resource assigned to VFW$_1$ and VFW$_2$, and can be obtained through experiments (see Figure 7(b)). For simplicity, VFW Controller assigns the same resource to each virtual firewall, hence $b1 = b2$.

VII. Optimal Scaling

In this section, we present an three-step heuristic approach VFW Controller uses to solve for optimal scaling-out and the ILP approach for optimal scaling-in. We start with modeling a virtual firewall.

A. *Virtual Firewall Modeling*

The processing capacity c of a virtual firewall depends on the resource granted and the number of loaded firewall rules. For easier management, each virtual firewall created by VFW Controller is granted the same resource. Therefore, we model c as a function of S, i.e. $c(S)$ where S is the number of loaded firewall rules. As S increases, c decreases. Because when processing a packet, a firewall search through its rule space until the first matching rule is found. Then the packet is processed according to the first matching rule. Larger S means larger the average search overhead, hence smaller c. Our experiments in §VIII-B1 validate such a relationship between c and S.

The runtime throughput of a virtual firewall, η, is a function of Λ, the aggregated rate of the incoming traffic flows.

$$\eta(\Lambda) = \begin{cases} \Lambda & \text{if } \Lambda \leq c \\ c & \text{if } \Lambda > c \end{cases}$$

When $\Lambda > c$, packet loss may be expected at a virtual firewall, because the rate of traffic coming to the virtual firewall exceeds the rate of traffic leaving. Service Level Agreement (SLA) on performance generally requires the processing capacity of a virtual firewall to meet a predefined value Φ, that is, $c(S) \geq \Phi$. Since c decreases as S increases, Φ essentially places an upper bound on S. The upper bound is $c^{-1}(\Phi)$. Therefore, $c(S) \geq \Phi$ is equivalent to $S \leq c^{-1}(\Phi)$.

B. *Optimal Scaling-Out*

Accurate overload detection is not the focus of this paper. VFW Controller simply deems a virtual firewall overloaded if one of the following conditions occurs: (i) $\eta > 0.9c$; (ii) $c(S) < \Phi$ (or equivalently $S < c^{-1}(\Phi)$). Condition (i) states that a virtual firewall is using above 90% of its processing capacity. 10% remnant is intentional to avoid packet loss. Condition (ii) states that the processing capacity of a virtual firewall fails to meet the SLA agreement.

To solve an overload, new virtual firewall instances, each of which has a limited buffer space of B, are created, selective firewall rule groups and the associated flow states on the overloaded virtual firewall are migrated to the new instances. The migration of each firewall rule group causes both a buffer cost and a update cost. VFW Controller solves the overload using the minimum number of instances while incurring the least total update cost and no buffer overflow.

After a virtual firewall is detected overloaded, VFW Controller runs *Dependency Analysis*, *Flow Update Analysis*, *Buffer Cost Calculation*, and has the following parameters:

- A set of m firewall rule groups $\mathbb{V} = \{V_1, V_2, ..., V_m\}$ with group V_i having a rule size of s_i, an update cost of γ_i, a buffer cost of β_i, and k_i matching traffic flows with the flow rates of λ_{i1}, λ_{i2}, ..., λ_{ik_i}.

- A processing capacity of $c(\sum_{i=1}^{m} s_i)$, and a runtime throughput of η.
- An overload condition that $c(\sum_{i=1}^{m} s_i) < \Phi$ or $\eta > 0.9c(\sum_{i=1}^{m} s_i)$.

Then, `VFW Controller` adopts a three-step heuristic approach to work out the minimum number of new instances to create and the firewall rule group distribution.

Step 1: `VFW Controller` estimates the minimum number of new instances to create. Let n be the number.

$$n = \begin{cases} \lfloor \frac{\sum_{i=1}^{m} \sum_{j=1}^{k_i} \lambda_{ij} - 0.9c(\sum_{i=1}^{m} s_i)}{\Phi} \rfloor & \text{if } \eta > 0.9c(\sum_{i=1}^{m} s_i) \\ \lfloor \frac{\sum_{i=1}^{m} s_i - c^{-1}(\Phi)}{c^{-1}(\Phi)} \rfloor & \text{if } S < c^{-1}(\Phi) \end{cases}$$

In the case when $\eta > 0.9c(\sum_{i=1}^{m} s_i)$, the amount of traffic that should be shunted away from the overloaded virtual firewall is $\sum_{i=1}^{m} \sum_{j=1}^{k_i} \lambda_{ij} - \eta$, which will be undertaken by the new instances, each of which must guarantee a processing capacity of Φ as dictated by the SLA. In the case when $S < c^{-1}(\Phi)$, the number of firewall rules that must be migrated is $\sum_{i=1}^{m} s_i - c^{-1}(\Phi)$, which will be split among the new instances, each of which is constrained by an upper bound $c^{-1}(\Phi)$. The floor operation in the estimate above implies that the estimate may be smaller than required.

Step 2: `VFW Controller` applies ILP to solve for firewall rule distribution. Let $\mathbf{x} = \{x_{11}, ..., x_{mn}\}$ be a set of unknown variables, where $x_{ij} \in \{0, 1\}$ is an indicator of migrating firewall rule group V_i to new instance j. The ILP formulation to solve $\mathbf{x}$ is below:

$$\min \quad \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} \gamma_i$$

s.t.

(1) $x_{ij} \in \{0, 1\}$ for $1 \le i \le m, 1 \le j \le n$

(2) $\sum_{j=1}^{n} x_{ij} \le 1$ for all $1 \le i \le m$

(3) $\sum_{i=1}^{m} x_{ij} \beta_i \le B$ for all $1 \le j \le n$ // Prevent buffer overflow on each new instance.

(4) $\sum_{i=1}^{m} x_{ij} s_i \le c^{-1}(\Phi)$ for all $1 \le j \le n$ // Each new instance must satisfy the SLA.

(5) $\sum_{i=1}^{m} x_{ij} (\sum_{l=1}^{k_i} \lambda_{il}) \le 0.9c(\sum_{i=1}^{m} s_i x_{ij})$ for all $1 \le j \le n$ // Each new instance must not be overloaded.

(6) $(\sum_{i=1}^{m} \sum_{j=1}^{k_i} \lambda_{ij} - \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{l=1}^{k_i} x_{ij} \lambda_{i,l}) \le 0.9c(\sum_{i=1}^{m} s_i - \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} s_i)$ // After scale out, the old firewall is not overloaded. This constraint is used when overload condition (i) occurs.

or

(6') $(\sum_{i=1}^{m} s_i - \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} s_i) \le c^{-1}(\Phi)$ //After scaling out, the old virtual firewall must satisfy the SLA. This constraint is used when overload condition (ii) occurs.

Solving the above ILP formulation, we obtain $\mathbf{x}$. If $x_{ij} = 1$ then firewall rule group V_i is to be migrated to new instance j. If $x_{ij} = 0$ for all j, then firewall rule group V_i stays on the old virtual firewall. If the above ILP formulation has no solution, which implies that the estimate of n at Step 1 is smaller than required, then go to Step 3.

Step 3: Increase n by one and perform Step 2 again until a solution is found.

C. Optimal Scaling-In

`VFW Controller` deems a virtual firewall underloaded, if its runtime throughput is 50% smaller than its processing capacity. When more than one virtual firewall are underloaded, `VFW Controller` merges them. `VFW Controller` applies the following ILP formulation to determine the maximum number of virtual firewalls to be killed while meeting the following constraints:

- Each remaining virtual firewall must not be overloaded after the mergence.
- Each remaining virtual firewall must not have its buffer space overflowed.
- Each remaining virtual firewall must satisfy the SLA.
- The total update cost in the mergence is bounded by a predefined value $\boldsymbol{\gamma}$.

Suppose there are n underloaded virtual firewalls. underloaded virtual firewall j has:

- A set of m^j firewall rule groups $\mathbb{V}^{\mathbb{j}} = \{V_1^j, V_2^j, ..., V_{m^j}^j\}$ with group V_i^j having a rule size of s_i^j, an update cost of γ_i^j, a buffer cost of β_i^j, and k_i^j matching traffic flows with the flow rates of λ_{i1}^j, λ_{i2}^j, ..., $\lambda_{ik_i^j}^j$.
- A processing capacity of $c(\sum_{i=1}^{m^j} s_i^j)$, and a runtime throughput of η^j.

Let $\mathbf{x} = \{x_{11}, ..., x_{nn}\}$ be a set of unknown variables, where $x_{qj} \in \{0, 1\}$ is an indicator of merging underloaded virtual firewall j onto q. The ILP formulation to solve $\mathbf{x}$ is presented below:

$$\max \sum_{j=1}^{n} \sum_{q=1}^{n} x_{qj}$$

s.t.

(1) $x_{qj} \in \{0, 1\}$ for all $1 \le q \le n, 1 \le j \le n$

(2) $x_{qq} = 0$ for all $1 \le q \le n$

(3) $\sum_{q=1}^{n} x_{qj} \le 1$ for all $1 \le j \le n$

(4) $\sum_{i=1}^{m^q} s_i^q + \sum_{j=1}^{n} \sum_{i=1}^{m^j} x_{qj} s_i^j \le c^{-1}(\Phi)$ for all $1 \le q \le n$

(5) $\sum_{i=1}^{m^q} \sum_{l=1}^{k_i^q} \lambda_{il}^q + \sum_{j=1}^{n} \sum_{i=1}^{m^j} \sum_{l=1}^{k_i^j} \lambda_{il}^j x_{qj} \le 0.9c(\sum_{i=1}^{m^q} s_i^q + \sum_{j=1}^{n} \sum_{i=1}^{m^j} x_{qj} s_i^j)$ for all $1 \le q \le n$

(6) $\sum_{j=1}^{n} \sum_{i=1}^{m^j} x_{qj} \beta_i^j \le B$ for all $1 \le q \le n$

(7) $\sum_{q=1}^{n} \sum_{j=1}^{n} \sum_{i=1}^{m^j} x_{qj} \gamma_i^j \le \boldsymbol{\gamma}$

Solving the above ILP formulation, we obtain all x_{qj}. If $x_{qj} = 1$, then virtual firewall j is merged onto q. If $\sum_{q=1}^{n} x_{qj} = 0$, then virtual firewall j remains. The maximum number of virtual firewall to be killed is $\sum_{j=1}^{n} \sum_{q=1}^{n} x_{qj}$.

In reality, there may be cases where conditions of scaling-out or scaling-in occur frequently thus requiring frequent scaling-out or scaling-in. To avoid thrashing, a potential solution is to set a threshold of frequency, above which the scaling process is not allowed. The suggested threshold depends on the scaling performance of `VFW Controller`, which can be measured in practice.

VIII. Implementation and Evaluation

A. Implementation

We have implemented a prototype of `VFW Controller` on top of ClickOS [38]. ClickOS is a Xen-based NFV platform optimized for fast provision of virtualized network functions in large scale. ClickOS creates small VMs (each less than 12MB) and boots one instance within 30 milliseconds. We have enhanced ClickOS to provide stateful virtual firewalls using Click modular router software [33]. Click provides rich networking processing elements, which can be leveraged to construct a wide range of virtual middleboxes. We have used three elements, *IPfilter*, *IPClassifier* and *IPFragmenter*, provided by Click to implement firewall packet processing function. We have also developed new Click elements for firewall rule management, buffer management, and interfaces for migrating rules and states. In particular, we have developed a programming interface on top of XL [10] [6], which `VFW Controller` uses to exert migration controls on individual virtual firewalls. Such a control automation is essential to fully take advantage of virtual firewall benefits [6]. Our VFW implementation provides useful insights to implement and control other virtualized network functions as well.

Key functions of `VFW Controller` have been realized as individual modules. In particular, we have implemented a *Dependency Analysis* module based on Header Space Library (Hassel) [4], which is a tool for static header space analysis, a *Flow Update Analysis* module to find the correct flow updates and calculate update costs, a *Buffer Cost Analysis* module to calculate buffer costs, and an *Optimal Scaling Calculation* module that realizes the approaches for optimal scaling by calling a Matlab ILP solver. `VFW Controller` also includes a Floodlight module that implements Floodlight SDN controller functions [8]. `VFW Controller` uses Floodlight REST APIs to communicate flow updates to Floodlight module, which programs SDN switches through an OpenFlow interface. The same channel is also used by SDN switches to send network traffic statistics back to `VFW Controller`.

To improve the performance of `VFW Controller`, our implementation uses both online and prior processing. All the analyses, including dependency analysis, flow update analysis, update cost calculation, and buffer cost calculation are carried out prior. `VFW Controller` maintains a copy of firewall rules for each virtual firewall and the flow rules in its database for prior analyses. Results from the analyses are stored and retrieved whenever scaling is to be performed. Overload/underload detection, optimal scaling calculation, virtual firewall creation/deletion, migrations of firewall rules and flow states, and flow updates are carried out online.

B. Evaluation

`VFW Controller` achieves *safe*, *efficient* and *optimal* virtual firewall scaling. We evaluate `VFW Controller` with the following goals:

- Demonstration of the relationship between virtual firewall performance and the rule size (Figure 7). This justifies `VFW Controller`'s choice of rule *split* over rule *copy*.
- Study of the rule dependency relations in firewall policies (Table I). This justifies the necessity of dependency analysis in `VFW Controller`.
- Demonstration of `VFW Controller`'s capability to quickly scale (Figure 8).
- Quantifying the impact of firewall rule migration on virtual firewall throughput (Figure 9).
- Evaluating the performance of `VFW Controller`'s optimal scaling calculation (Figure 10).

Setup and Methodology: Our experiments were conducted using CloudLab [3], an open cloud platform providing various resources (server, storage, and network) for experimenters to run cloud software stacks such as OpenStack and CloudStack. In our experiments, we deployed a client machine that generated traffic, a server machine that received traffic, and a firewall machine created by `VFW Controller` to process the traffic between the client and the server. The client generated synthetic workloads using scapy [7], a powerful interactive packet manipulation program.

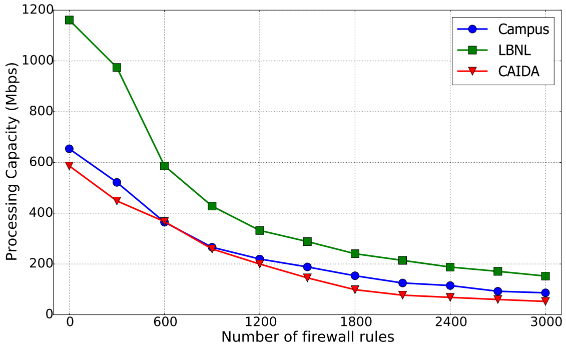
1) Performance of Virtual Firewalls: In this experiment, we examined the relationship between the processing capacity, c, of a virtual firewall and its firewall rule size S. We used three traffic datasets captured from real-world networks: 1) the CAIDA UCSD anonymized Internet trace [16] is a representative of Internet traffic; 2) the LBNL/ICSI enterprise trace [5] is a typical traffic collected from an enterprise network; and 3) the Campus network trace that was collected from our campus network gateway. Against each of the dataset, we conducted experiments to study how the firewall rule size affects the performance of a virtual firewall. In each experiment, we let S increase from 1 to 3000, which we considered as a maximum number of rules in a single VFW. Then we measured the processing capability for each S value and repeated each measurement 100 times to calculate the average processing capacity, as shown in Figure 7(a). The average processing capacity linearly decreases as S increases over all datasets. The LBNL enterprise traffic contains packets with greater length of payload, thus bares a more significant impact on performance. We applied polynomial curve fitting linear regress on the CAIDA trace, which was captured by April 2016 and represents the most up-to-date characteristic of today's Internet traffic, and obtained the function $c(S)$ as

$$c(S) = -0.0043S + 6.2785 \qquad (2)$$

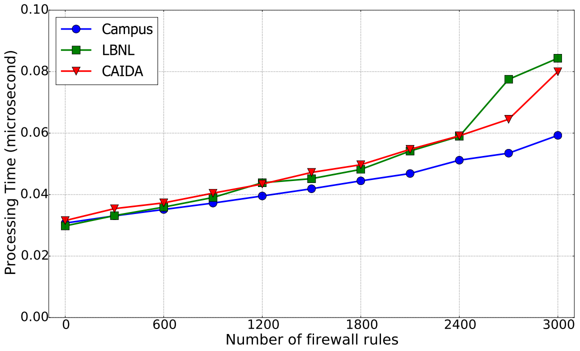
This function fits the corresponding CAIDA curve in Figure 7(a) with $R^2 = 0.9864$. R^2 is a measure of goodness of fit with a value of 1 denoting a perfect fit. $c(S)$ will be used by the evaluation of optimal scaling calculation (§ VIII-B5).

[6] XL is a toolstack that provides the capability to provision guest VM in Xen.

[7] http://www.secdev.org/projects/scapy/.



(a) Processing capacity.



(b) Processing time per packet.

Fig. 7: Relationship between virtual firewall performance and rule size.

In the experiment, we also recorded the average time a virtual firewall instance spent to process a packet for each of the real-world dataset, as shown in Figure 7(b). As S increases, the average processing time shows an linear increase, which eventually incurs the latency of the passed packets. These results of this experiment can be used for buffer cost calculation (§ VI-A).

2) Rule Dependencies in Firewall Policies: In this experiment, we studied 8 *real-world* firewall policies from different resources. Most of them are from campus networks and some are from major ISPs. We partitioned each policy to disjoint groups using the partition algorithm given in Section IV. The experimental results are listed in Table I. The first two columns of the table show the policies we used and their rule numbers. The third column gives the number of groups identified in each policy, and the fourth column shows the number of the firewall rules in the largest group of each policy. This experiment demonstrates that rule dependencies are common in *real-world* firewall policies. Therefore, the dependency analysis in VFW Controller is necessary. We also noticed from our study that the largest group we encountered contains only 18 firewall rules (from Policy H in Table I).

3) Elasticity of VFW Controller: In this experiment, we demonstrated VFW Controller's ability to elastically scale out an overloaded virtual firewall. We designed three scenarios. In scenario (1), a single virtual firewall was created and configured with 400 firewall rules to process the traffic between the client and the server. In scenario (2), two single virtual firewalls were created to work in parallel process the same traffic. The 400 firewall rules in scenario (1) were split and installed in those two virtual firewalls. In both scenarios, the virtual firewalls worked in standalone mode, which meant they were not connected to VFW Controller and no scaling would be performed. In scenario (3), a single virtual firewall with the same configuration as that in scenario (1) was created to work in standalone mode at first, then switched to connected mode, which meant the virtual firewall was connected to VFW Controller and was scaled out. We compared the runtime throughput of the virtual firewalls in the three scenarios as shown in Figure 8.

TABLE I: Rule dependencies in real-world firewall policies.

Policy	Rule (#)	Group (#)	Largest Group Member (#)
A	12	2	3
B	18	3	5
C	25	3	6
D	52	7	7
E	83	9	7
F	132	10	9
G	354	10	12
H	926	13	18

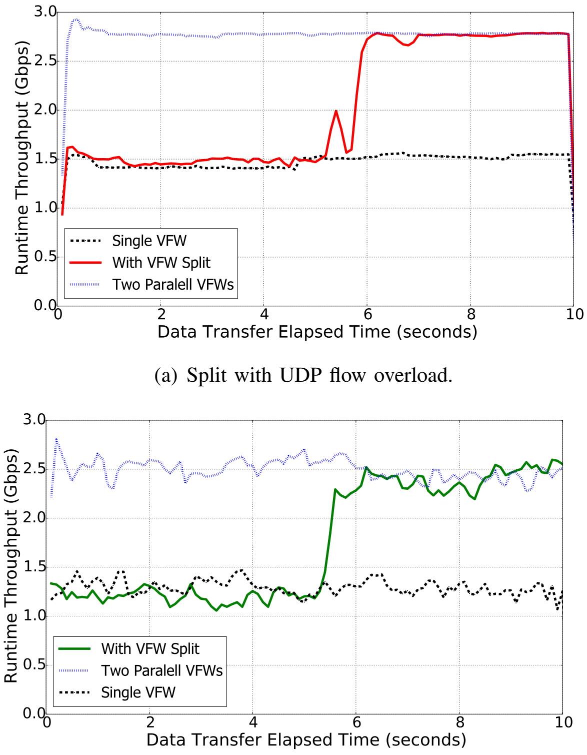
Figure 8(a) shows the runtime throughput of the three scenarios when the client generated 4 UDP flows with an aggregated traffic rate of about 2.8 Gbps. In scenario (1), the single virtual firewall achieved a maximum throughput (i.e. processing capacity) of about 1.5 Gbps. Significant packet loss was experienced. In scenario (2), the two virtual firewall were able to handle the incoming traffic, producing a runtime throughput of about 2.8 Gbps. No packet loss occurred. In scenario (3), we intentionally let a single virtual firewall to work from time $t = 0s$ to $t = 5s$. Packet loss occurred during this time period. Then we connected this virtual firewall to VFW Controller and it was scaled into two virtual firewall instances. The scaling-out took a short period of time (¡ 1 second), after which we observed the aggregated runtime throughput increased to around 2.8 Gbps.

We also evaluated VFW Controller against TCP traffic. Figure 8(b) shows the runtime throughput of the three scenarios when the client established 2 TCP connections with the server. We observed a boost of runtime throughput between $t = 5s$ and $t = 6s$.

In summary, the above results demonstrated that VFW Controller can quickly scaled out an overloaded virtual firewall and solved the overload condition.

4) Impact of Migration: During the migration, the in-flight traffic are buffered until the migration completes, therefore the runtime throughput of a virtual firewall is expected to degrade during the migration. According to equation (1) in §VI, if more traffic flows are associated with the migrated firewall rules, more in-flight traffic will be buffered. Therefore larger the degradation will be and longer the degradation will last. In this experiment, we quantified both the duration and magnitude of throughput degradation. We set up two scenarios to test UDP and TCP flows, respectively.

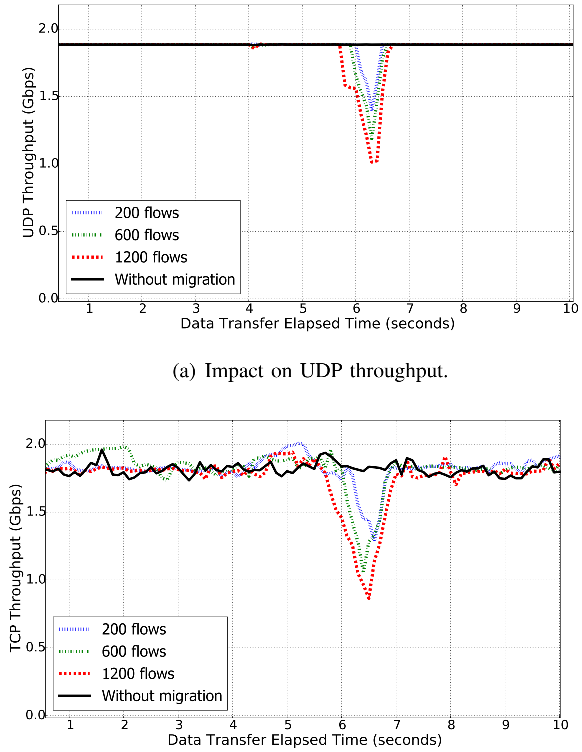
In scenario (1), the client kept sending UDP traffic destined to the server. The UDP packets were routed through a virtual firewall created by VFW Controller. In Figure 9(a), the



(a) Split with UDP flow overload.

(b) Split with TCP flow overload.

Fig. 8: `VFW Controller` for VFW elasticity.



(a) Impact on UDP throughput.

(b) Impact on TCP throughput.

Fig. 9: Performance overhead of migration.

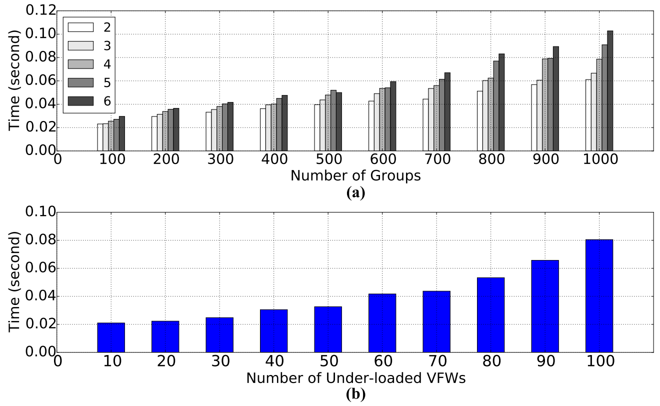


Fig. 10: Performance of provision plan calculation.

solid black line, as a base line, shows the throughput of the virtual firewall when no migration occurred. The dotted lines in Figure 9(a) show the runtime throughput of the virtual firewall when migrating 200, 600, 1200 firewall rules, respectively. The migrations were scheduled at $t = 6s$. We fabricated firewall rules so that each firewall rule has one matching flow. The duration and magnitude of throughput degradation increased as the number of migrated firewall rules (or the number of UDP flows) increased. However, the degradation lasted for a very short period of time (≈0.75s with 1200 flows) and the throughput bounced back very quickly. In scenario (2), the client sent TCP traffic. Figure 9(b) shows that the degradation of throughput of the virtual firewall when processing TCP traffic. As the number of firewall rules (or TCP flows) increases, the degradation lasted longer and was bigger. The degradation in the case of TCP traffic lasted slightly longer than that of UDP, because TCP depends on congestion avoidance mechanisms to control its traffic rate, which means TCP flows take more time to recover from a throughput degradation than UDP flows. For both UDP and TCP flows, the throughput began to bounce back in less than 0.1 second after it reaching the lowest point (see Figures 9(a) and (b)).

To sum up, the duration and magnitude of throughput degradation increase as the number of migrated firewall rules (or the number of flows) increase. And the degradation shows an bigger impact on TCP flows than UDP flows.

5) Performance of Optimal Scaling Calculation: We introduced a three-step heuristic approach in `VFW Controller` to calculate an optimal solution for scaling out an overloaded virtual firewall. The performance of this approach depends almost fully on the ILP formulation and solving used in the second step. Therefore, we mainly tested the performance of our ILP formulation and solving. The ILP performance is influenced by (i) m, the number of firewall rule groups on the overloaded virtual firewall; and (ii) n, the number of new virtual firewall instances to be created. In our experiment, we measured the time to find the optimal solution for varied m and n values. We let n range from 2 to 6, and m range from 100 up to 1000. Figure 10(a) shows our experiment results. The time to find an optimal solution increases as m or n increases, however, the time is very short. Even for $m = 1000$, our approach needs less than 0.11 second.

Figure 10(b) depicts the time that our approach consumed to find an optimal solution for scaling in when the number of underloaded virtual firewall instances is changed. Our approach is very efficient. Even when 100 virtual firewall instances are detected to be underloaded, our approach takes less than 0.08 second to find an optimal scaling-in solution.

IX. Related Work

Several recent research efforts, such as Pico Replication [43], Split/Merge [44] and OpenNF [28], have been devoted to designing control systems to address scaling issues of virtualized network functions. Pico Replication provides APIs that NFs can use to create, access, and modify internal states. Split/Merge achieves load-balanced elasticity of virtual middleboxes, via splitting internal states of NFs among virtual middlebox replicas and re-routing flows. However, both Pico Replication and Split/Merge cannot achieve loss-free during NF state migration. OpenNF provides fine-grained control over the move of internal NF states from one NF instance to another, and enables loss-free and order-preserving state migration. While those control systems attempted to accommodate a wide range of virtualized network functions, *semantic consistency* is specific to virtual firewalls and remains unsolved by those systems. Besides, neither do existing control systems address the challenge of *correct flow update*, nor do they support *optimal scaling*. In addition, most of existing control systems, such as OpenNF, use centralized buffers at the controller to buffer in-flight traffic. Such a solution has significant drawbacks, since it consumes valuable bandwidth between the controller and virtualized functions, and lacks an effective mechanism to avoid *buffer overflow*. In comparison with those exiting systems, `VFW Controller` is specifically designed as a controller to address challenges in virtual firewall scaling with respect to *semantic consistency*, *correct flow update*, *buffer overflow avoidance*, and *optimal scaling*.

NFV and SDN techniques have been recently used to solve the inflexibility and inelasticity limitations of hardware-based DDoS defense appliances. Particularly, Fayaz et al. [25] proposed Bohatei, a flexible and elastic virtual DDoS defense system, for effectively defending DDoS attacks. In contrast, we address the unique challenges presented by hardware-based firewalls, leveraging the features provided by NFV and SDN as well.

Hu et al. [30] proposed FlowGuard, a framework for building SDN firewalls, to protect OpenFlow-based networks. FlowGuard-based firewalls are actually SDN applications running on top of SDN controllers. Challenges in designing SDN firewall applications were addressed in FlowGuard. However, the work presented in this paper investigates solutions for the safe and efficient management of a new type of firewalls, *virtual* firewalls, in the context of both NFV and SDN.

Zhang et al. [50] demonstrated that careless policy updates may result in security violations. They presented safe and efficient policy update algorithms for firewall policy updates. However, the proposed algorithms are only able to deal with policy updates on a *single* firewall, while the rule migration mechanism introduced in `VFW Controller` coordinates firewall rules across *multiple* firewalls.

X. Conclusion and Future Work

The hardware-based firewall is limited by its inflexible nature with regard to fixed capacity and fixed deployment location. We observed two emerging networking paradigms, NFV and SDN, offer the opportunity to address the limitations by implementing firewall functions as virtual firewalls. In this paper, we have proposed `VFW Controller`, a virtual firewall controller, which enables *safe*, *efficient* and *optimal* virtual firewall scaling. To demonstrate the feasibility of our approach, we have implemented the core components of `VFW Controller` on top of ClickOS. In the future, we plan to implement `VFW Controller` in other popular open-source NFV platforms, such as OPNFV [7] and CORD [1]. We also plan to generalize the buffer cost analysis and optimal scaling approaches introduced in this paper and integrate them with other exiting NF control systems, such OpenNF. Besides, we will investigate solutions to address specific challenges in the elastic scaling of other network *security* functions. For example, data and control dependencies [21] need to be addressed in building *virtual* intrusion detection systems.

Acknowledgment

This work was partially supported by grants from National Science Foundation (NSF-ACI-1642143, NSF-ACI-1642031, NSF-IIS-1527421, and NSF-CNS-1537924).

References

[1] Central Office Re-architected as a Datacenter (CORD). http://opencord.org/.

[2] Check Point virtual appliance for AWS. https://aws.amazon.com/marketplace/pp/B00CWNBJOY.

[3] CloudLab. http://www.cloudlab.us/.

[4] Header Space Library (Hassel). http://stanford.edu/~kazemian/hassel.tar.gz.

[5] LBNL/ICSI Enterprise Tracing Project. http://www.icir.org/enterprise-tracing/.

[6] Network Function Virtualisation - Introductory White Paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf.

[7] OPENFV. https://www.opnfv.org/.

[8] Project Floodlight. http://www.projectfloodlight.org/projects/.

[9] Scalable Firewall Services with Virtual Systems. https://www.paloaltonetworks.com/products/features/virtual-systems.html.

[10] Xen Toolstack. http://wiki.xen.org/wiki/XL.

[11] Overview of virtual firewalls on VBLOCK$^{\text{TM}}$ infrastructure platforms. http://www.vce.com/asset/documents/virtual-firewall-whitepaper.pdf, 2012.

[12] AT&T Vision Alignment Challenge Technology Survey. http://www.att.com/Common/about_us/pdf/AT&TDomain2.0VisionWhitePaper.pdf, 2013.

[13] How SDN enabled innovations will impact AT&T's plans to transform it's infrastructure. https://www.youtube.com/watch?v=tLshR-BkIas, 2014.

[14] http://www.prweb.com/releases/opennetsummit/20150217/prweb12592239.htm, 2015.

[15] OpenFlow Switch Specification Version 1.5.1. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf, 2015.

[16] The CAIDA UCSD Anonymized Internet Traces 2016-0406. http://www.caida.org/data/passive/passive_2016_dataset.xml, 2016.

[17] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[18] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. In *Proceeding of SIGCOMM Workshop on Research on Enterprise Networking*, Barcelona, Spain, 2009.

[19] P. Bernier. Google implements NFV, SDN as part of its Andromeda Effort. http://www.nfvzone.com/topics/nfv/articles/376827-google-implements-nfv-sdn-as-part-its-+andromeda.htm, 2014.

[20] P. Busschbach. Network functions virtualization: challenges and solutions. http://www.tmcnet.com/tmc/whitepapers/documents/whitepapers/2013/9377-network-functions-virtualization-challenges-+solutions.pdf, 2013.

[21] L. D. Carli, R. Sommer, and S. Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceeding of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, Scottsdale, AZ, 2014.

[22] M. Chapel. Firewall rules are meant to be managed, not broken. http://www.biztechmagazine.com/article/2012/08/firewall-rule-management-key-network-security, 2012.

[23] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[24] Q. Duan and E. S. Al-Shaer. Traffic-aware dynamic firewall policy management: techniques and applications. *IEEE Communications Magazine*, 51:73–79, 2013.

[25] S. Fayaz, Y. Tobioka, and V. Sekar. Bohatei: flexible and elastic DDoS defense. In *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[26] O. N. Foundation. OpenFlow-enabled SDN and network functions virtualisation. https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-sdn-nvf-solution.pdf, 2014.

[27] A. Gember-Jacobson and A. Akella. Improving the safety, scalability, and efficiency of network function state transfers. In *ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2015.

[28] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 163–174, 2014.

[29] H. Hu, G.-J. Ahn, and K. Kulkarni. Detecting and Resolving Firewall Policy Anomalies. *IEEE Transactions on Dependable & Secure Computing*, 9(3):318–331, 2012.

[30] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao. FlowGuard: building robust firewalls for software-defined networks. In *Proceedings of the 3rd ACM SIGCOMM workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, pages 97–102. ACM, 2014.

[31] P. Kazemian, G. Varghese, and N. Mckeown. Header Space Analysis: Static Checking For Networks. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[32] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pages 15–27, 2013.

[33] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[34] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: a comprehensive survey. *Proceedings of the IEEE*, 13:14–76, 2015.

[35] A. X. Liu and M. G. Gouda. Complete Redundancy Removal for Packet Classifiers in TCAMs. *IEEE Transactions on Parallel & Distributed Systems*, 21(4):424–437, 2010.

[36] A. X. Liu, E. Torng, and C. R. Meiners. Firewall Compressor: An Algorithm for Minimizing Firewall Policies. In *Proceedings of the 27th Conference on Computer Communications (INFOCOM'08)*, pages 176–180, 2008.

[37] R. Maddipudi. vCloud networking and security 5.1 app firewall. http://blogs.vmware.com/vsphere/2013/04/vcloud-networking-and-security-5-1-+app-firewall-part-1.html, 2013.

[38] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*, pages 459–473, 2014.

[39] N. Mckeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2010.

[40] G. Misherghi, L. Yuan, Z. Su, C.-N. Chuah, and H. Chen. A general framework for benchmarking firewall optimization techniques. *IEEE Transactions on Network & Service Management*, 5(4):227–238, 2008.

[41] A. Nunes, M. Mendonca, X. Nguyen, and K. Obraczka. A survey of software-defined networking: past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[42] R. Ozdag. The day Amazon ended the hardware-defined network. http://www.cyaninc.com/blog/2013/11/26/the-day-amazon-ended-the-hardware-defined-network#.VbehsxNViko, 2013.

[43] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A high availability framework for middleboxes. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 1. ACM, 2013.

[44] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pages 227–240, 2013.

[45] S. K. N. Rao. SDN and its Use-Cases-NV and NFV. http://www.nectechnologies.in/en_TI/pdf/NTI_whitepaper_SDN_NFV.pdf, 2014.

[46] I. Scales. Survey indicates NFV/SDN deployments in 2015. http://www.telecomtv.com/articles/sdn/survey-indicates-nfv-sdn-deployments-in-2015-11062/, 2015.

[47] 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. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[48] J. Sherry and S. Ratnasamy. A survey of enterprise middlebox deployment. In *Technical Report UCB/EECS-2012-24*. EECS Department, University of California, Berkeley, 2012.

[49] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–pp. IEEE, 2006.

[50] C. C. Zhang, M. Winslett, and C. A. Gunter. On the safety and efficiency of firewall policy deployment. In *2007 IEEE Symposium on Security and Privacy (S&P'07)*, pages 33–50. IEEE, 2007.