A Software-Defined Firewall Bypass for Congestion Offloading

Florian Heimgaertner, Mark Schmidt, David Morgenstern, and Michael Menth
Chair of Communication Networks, University of Tuebingen, Tuebingen, Germany
Email: {florian.heimgaertner,mark-thomas.schmidt,menth}@uni-tuebingen.de, david@morgenstern.net

***Abstract*—With increasing network bandwidths, stateful firewalls are likely to become communication bottlenecks in networks. To mitigate this problem, we propose to bypass selected traffic around firewalls using software-defined networking (SDN). We discuss various approaches and elaborate the following concept. A controller samples outgoing packets at the firewall using sFlow to detect congestion. In case of congestion, flows already admitted by the firewall are identified and offloaded at an appropriate rate by installing flow-specific bypass rules on an OpenFlow-capable switch. We suggest two different algorithms to select appropriate flows and provide a proof-of-concept implementation in a network testbed using the Ryu controller framework. Experimental results illustrate the system behavior at different load levels with and without offloading. We provide an analytical system model to predict the offloading performance for other system parameters than experimentally evaluated and validate the model with our experimental results. A parameter study suggests that the offloaded traffic rate may be a multiple of the firewall's capacity if the switch supports sufficient flow rules or is able to match for TCP flags.**

I. Introduction

Access ports with 1 Gb/s are becoming more and more common in business and campus Ethernet networks. To transport the traffic generated by the endpoints, backbones and uplink connections are upgraded to 10 Gb/s or higher bandwidths. This upgrade comes with the need for new network hardware. While Ethernet switches with 1 Gb/s and 10 Gb/s interfaces are affordable, firewalls handling 10 Gb/s and more are expensive. To benefit from increased network bandwidths without replacing existing firewall hardware, we propose to bypass some of the traffic around the firewall using software-defined networking (SDN). This makes sense because firewalls can handle many flows but are limited in transmission speed while commodity OpenFlow switches support only a limited number of flow rules in hardware, but operate at high bandwidth. The approach also reduces the security level but only to a moderate extent because most checks are performed by firewalls during the setup of a flow and the bypasses are used only for individual flows that have already been permitted by the firewall.

The contribution of this work is a discussion of techniques and prerequisites for firewall bypassing. We further concentrate on an approach where a controller samples outgoing packets at the firewall to detect congestion. In case of congestion, the controller identifies appropriate flows and offloads them by installing flow-specific rules on a switch to bypass the firewall. We suggest algorithms to detect the congestion on the basis of sampled rates and to determine appropriate offloading rates such that available flow rules are effectively leveraged. To provide a proof-of-concept, we implement the proposed mechanism using OpenFlow and sFlow in a networking testbed. Experimental results illustrate the system behavior at different load levels with and without offloading. The offloading performance in terms of offloadable traffic is limited in our experimental setup but depends on hardware capabilities and traffic patterns that may change in the future. We provide an analytical system model to predict the offloading performance for different system parameters and validate the model with our experimental results. A parameter study suggests that the offloaded traffic rate may be a multiple of the firewall's capacity if the switch supports sufficient flow rules or is able to match for TCP flags. The suggested mechanism distinguishes from other work as it can be integrated in legacy networks that are not fully SDN-enabled. Another salient feature of this approach is that signaling with the firewall is not needed, i.e., the firewall is handled as a black box which makes the approach independent of a specific firewall product.

This work is structured as follows. Section II provides an overview of the technolgy used. Section III reviews related work. In Section IV, we discuss various concepts for firewall bypassing and Section V explains the algorithms used for the approach explained above. Section VI reports a proof-of-concept implementation and illustrates the system behavior through experimental results. Section VII presents a theoretical performance analysis including numerical results and Section VIII concludes this work.

II. Technological Background

SDN separates the *data plane* and the *control plane* by shifting intelligence from distributed forwarding nodes, i.e., routers or switches, to a logically centralized controller [1]. While non-SDN switches populate forwarding tables by learning addresses, an SDN controller installs a set of forwarding rules on SDN switches either on initialization or during runtime. Flow rules may be associated with a timer so that they are automatically removed from the forwarding table of the switch when they have not been used for the specified time. The communication channel between SDN controllers and SDN switches is called *southbound interface*. The Open Networking Foundation (ONF) has standardized the OpenFlow [2] protocol for that purpose. Our controller implementation is based on the Ryu SDN framework [3] which supports all OpenFlow versions from 1.0 up to 1.5 and sFlow.

sFlow [4] is a vendor-independent technology for monitoring network traffic. Measurement data are sent by sFlow *agents* in sFlow datagrams to sFlow *collectors*. The agent is part of the switch while the collector can be part of a monitoring system or, in our case, an SDN controller. We use the *packet flow sampling* method of sFlow version 5 and the *raw packet header* datagram format.

III. Related Work

The authors of [5] propose traffic-aware flow offloading (TFO) to offload heavy hitters from routers to specialized forwarding hardware. A similar approach is CacheFlow [6] which uses a high-speed SDN switch with limited hardware flow tables like a cache for a slower software SDN switch supporting a larger number of flow rules. This way, less frequently used rules are offloaded to software switches. Both approaches are aggregate-based, e.g., TFO works on the level of BGP prefixes, which is not acceptable for our purpose as flows should be permitted individually if they are accepted by the firewall.

The authors of [7] use SDN to implement a reactive, stateful firewall. SDN switches are configured to enforce security policies using forwarding rules. The concept was implemented with Ryu and tested using Mininet and Open vSwitch (OVS). A similar concept [8] was implemented using the POX controller and OVS. However, these approaches are limited by controller performance and might not work with hardware switches.

Network function virtualization (NFV) enables replacing middlebox hardware like firewalls by software components on virtual machines (VMs). VNGuard [9] is a framework for efficient provisioning and management of NFV firewalls. The framework allows for dynamic placement of NFV firewalls inside a network via reconfiguration of the network topology using SDN.

The authors of [10] present a combination of an NFV firewall and an SDN-based firewall. A pure SDN firewall as described in [7] monitors the state of a connection using an SDN controller. Therefore, handshake packets are transmitted to the controller causing additional delay because the control channel is relatively slow. An NFV firewall is limited in throughput by the fact that all processing is done in software. The combined SDN/NFV firewall presented in [10] initially forwards traffic via an NFV firewall which tracks the state of all accepted connections. For very large connections, the SDN-based strategy takes over without additional delay as the handshake was carried out via the NFV firewall.

NFShunt [11] is an extension of the Science DMZ [12] concept. The authors propose a bypass for a Linux-based software firewall using SDN. NFShunt extends the firewall rule set so that the bypassing rules can be specified in the firewall configuration.

All the strategies described above require a network infrastructure which is completely based on SDN-capable switches, but this is not the case in many current installations. Therefore, their applicability is limited. We aim at relieving firewalls from congestion in existing networks while only requiring minor changes to the network infrastructure. While our work appears similar to the one presented in [10], a major difference is that we consider the firewall as a black box.

A patent held by Google [13] suggests to bypass every flow that has been accepted by the firewall using SDN, i.e., the firewall serves mainly as decision engine. However, this proposal ignores that switches have only a limited number of flow rules and that traffic offloading in the absence of congestion just reduces the security level. Our approach is similar in the sense that it uses the firewall as decision engine, but it strives to bypass largest flows only in the presence of congestion for two reasons. First, this approach makes best use of available flow rules on the switch. Second, it compromises the security level only to avoid service degradation. For this purpose we propose robust and effective measurement and management algorithms.

IV. Firewall Bypass

In this section we clarify some terminology, explain general firewall usage, and propose static and dynamic firewall bypassing for selected traffic.

A. Terminology and Usage

A *flow* describes a directed connection between two endpoints identified by the 5-tuple *(src.-addr., dst.-addr., protocol, src.-port, dst.-port)*. For connection-oriented transport protocols like TCP, a connection comprises a pair of flows, each flow representing one direction. A firewall is a network middlebox which applies rules and policies to decide whether certain packets are permitted to pass. Stateless firewalls just apply static filters to individual packets. In contrast, stateful firewalls track connections and apply rules to packets in the context of the connection they belong to.

Firewalls are commonly used to shield a layer-3 network which is denoted as *inside* while the outside of the network is denoted as *outside*. The objective is to protect the inside against malicious traffic from outside and to prevent undesired actions from inside to the outside. Normally, inside and outside are connected by a router.

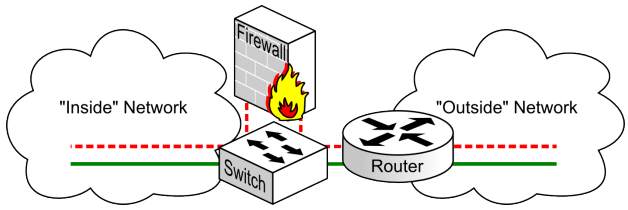
Two different operation modes exist for firewalls, *transparent* mode and *routed* mode. In transparent mode, a firewall acts like a layer-2 bridge, i.e., it is invisible to layer-3 devices. In routed mode, a firewall acts like a layer-3 router.

Firewall bypassing means that selected traffic is diverted around the firewall to reduce its forwarding load, i.e., the traffic is offloaded. It requires a network element on the path that conditionally steers traffic around the firewall depending on flow descriptors. We use the term *bypass* to describe a bidirectional path between the inside and outside networks that does not traverse the firewall. For the action of installing a bypass for a pair of flows we use the term *offloading*. The bypass is implemented using a switch connected to the inside network, the firewall, and the outside network. In transparent mode, the switch is connected to the outside network via a router. As in routed mode the bypass does not only skip

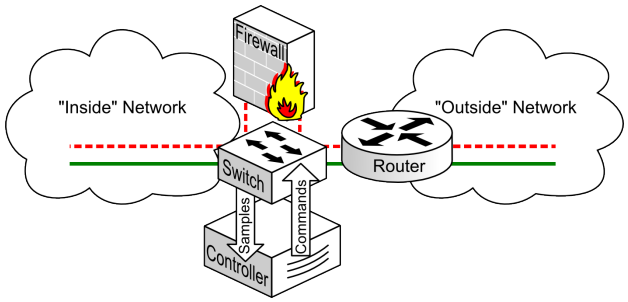
the filtering part of the firewall, but also the router part, the switch needs to implement parts of the router functionality, such as rewriting MAC addresses. This makes the bypass implementation more complicated. Therefore, we consider a transparent firewall scenario in this paper.

B. Static Firewall Bypass

Static firewall bypassing means that some ranges of flow descriptors are permitted a priori and are diverted around the firewall. These ranges can be denoted as a whitelist and configured in the access control lists of a managed switch. The whitelisted traffic may directly be forwarded from inside to outside and vice-versa while other traffic is first forwarded to the firewall. This is illustrated in Figure 1(a). The solid green line represents a trusted flow that can bypass the firewall by whitelist entry while the flow represented by the dashed red line goes through the firewall. Whitelisting is effective if a large fraction of the traffic can be considered trusted a priori.



(a) A switch may bypass traffic defined by a static whitelist.



(b) A controller may learn via sFlow about congestion and offloadable flows on the firewall and dynamically install bypassing rules on an SDN switch.

Fig. 1: Bypass options for transparent firewalls.

C. Dynamic Firewall Bypass

Dynamic firewall bypassing means that individual flows may be offloaded after being permitted by the firewall. This is acceptable as the most important checks are performed by a firewall during the setup phase of a connection, afterwards a connection is rather unlikely to be blocked by the firewall. Therefore, we define a flow as offloadable if at least one packet has passed the firewall after the setup phase.

A typical scenario is depicted in Figure 1(b). An SDN-capable switch steers traffic through a transparent firewall before leaving or entering the inside network. An SDN controller installs per-flow bypassing rules on the switch so that selected flows permitted by the firewall are forwarded directly from inside to outside and vice-versa. A challenge is that OpenFlow-capable switches support only a moderate number of flow rules. Therefore, bypassing should be applied only during periods of congestion and to largest possible flows. To detect congestion and to learn about offloadable flows, we use sFlow [4] to sample outgoing packets on specific ports of the switch and export their headers to the SDN controller. When the controller detects congestion and has identified an offloadable flow, it can install forwarding entries at the switch to bypass that flow. As flow rules are uni-directional, two different flow rules are needed for inbound and outbound traffic of a TCP connection.

A simple idea to deal with the shortage of flow rules on OpenFlow-capable switches is to remove the bypass for a specific flow when the flow becomes silent and to relocate it to the firewall such that future packets of that flow are again checked by the firewall. However, this does generally not work in practice. If the state of an offloaded flow has timed out on the firewall, the firewall drops packets after relocation. If the state has not yet timed out, the firewall still drops the packets for out-of-window TCP sequence numbers. Therefore, relocation of previously offloaded flows back to the firewall is not feasible.

V. Algorithms for SDN-Based Firewall Bypassing

In this section we explain algorithms for SDN-based firewall bypassing. The SDN controller learns about the load on the firewall by traffic sampling using sFlow. Based on this information, the controller decides when and which specific rule should be installed on the switch to bypass a certain flow. We offer a random and an intelligent strategy to select flows for offloading.

A. Load Measurement

We use sFlow to export the headers of every n_s^{th} packet leaving the switch on a specific port to the controller. The arrival time and size of the sampled packets serve as input for rate measurement using TDRM-UTEMA for time-dependent rate measurement [14], yielding an estimate of the current traffic rate on the firewall. It may be expressed as sampled packet rate r_F^p (packets/s) or as sampled byte rate r_F^b (bytes/s). We set TDRM-UTEMA's memory to $M_F = 5\,\mathrm{s}$ which essentially determines the time scale of the measurement process. The memory is chosen in the order of delay perceived for a packet delivery during congestion including retransmissions.

B. Offloading Algorithm

The limited number of flow rules on the switch prohibits the offloading of an arbitrary number of flows. The resulting challenge is to decide when a certain flow should be offloaded. We first determine the number of installable bypasses and the time after which they may be reused. Then, we explain under which conditions offloading is activated and derive an appropriate offloading rate.

1) Offloading Requirements: Offloading of a TCP connection requires two flow rules on the switch: one to bypass outbound traffic coming from inside and one to bypass inbound traffic coming from outside. Therefore, the number of available rules on the switch n_r allows for $n_{by} = \frac{n_r}{2}$ bypassed connections.

2) Reuse Time of Flow Rules: We consider the time $t_{reuse} = t_{rct} + t_{out} + t_{aoh}$ after which a flow rule is again available on the controller for offloading another flow. Thereby t_{rct} denotes the remaining completion time of the flow, i.e., the time from its offloading start until its last packet. If the flow rule was unused for t_{out} time, it is deinstalled from the switch. That time is a configuration parameter. Finally, t_{aoh} accounts for additional overhead that occurs, e.g., because flow rules are deinstalled in batches.

3) Detection of (Imminent) Overload: We consider the firewall as highly loaded when the utilization of the forwarding capacity C_F^b (bytes/s) exceeds a configured high-load threshold $T_H = 0.8$. Furthermore, we use $r_F^b > T_H \cdot C_F^b$ as precondition for offloading. The condition is checked whenever a new sample is received. A time-averaged fraction of high-load situations is tracked by a moving average and denoted as H. The UTEMA method [14] is used for that purpose with a memory of $M_H = 2 \cdot t_{reuse}$ because the intent is to smooth offloading over t_{reuse} time. Details of this smoothing are explained in the next paragraph.

4) Target Offloading Rate: The scarcity of potential bypasses limits the maximum average offloading rate to $r_{off}^{reuse} = \frac{n_{by}}{t_{reuse}}$ over the reuse interval. Taking into account that traffic is offloaded only for a fraction H of the time, the sustainable offloading rate is $r_{off}^{sus} = \frac{n_{by}}{H \cdot t_{reuse}}$. This rate can be very large when imminent congestion is rarely observed so that available bypasses can be quickly consumed. To avoid too fast exhaustion, we require that the remaining bypasses n_{by}^{rem} suffice for a short smoothing time t_{ssm}, which leads to a smoothed offloading rate $r_{ssm} = \frac{n_{by}^{rem}}{t_{ssm}}$. Combining both conditions leads to a target offloading rate of

$$r_{off} = min\left(r_{off}^{sus}, r_{off}^{ssm}\right) = min\left(\frac{n_{by}}{H \cdot t_{reuse}}, \frac{n_{by}^{rem}}{t_{ssm}}\right). \quad (1)$$

C. Random Offloading (ROff)

Through packet sampling with sFlow, the controller knows about flows traversing the firewall. To avoid offloading inactive flows, the controller offloads a flow only when it receives a sampled packet from it. Moreover, a TCP flow is offloaded only if it is fully established, i.e., if the SYN flag of the sampled packet is not set.

In the absence of (imminent) congestion, flows are generally not offloaded. In the presence of (imminent) congestion, the flow of a sampled packet is offloaded with a probability p_{off} that helps to meet the target offloading rate r_{off}. This target offloading probability is computed by $p_{off} = \frac{r_{off}}{r_F^b}$.

As all sampled packets are treated equally, we call this strategy random offloading (ROff). Nevertheless, long flows are sampled with a higher probability than small flows, causing higher offloading probabilities for larger flows, which is beneficial for offloading effectiveness.

D. Intelligent Offloading (IOff)

To improve the offloading effectiveness, we propose an alternative offloading algorithm that selects longest possible flows. We call it intelligent offloading (IOff). It pursues the rationale that flows for which many sampled packets have been received tend to be large and run longer than other flows.

With every sample arrival, the controller records flows from which it received sampled packets in a flow list. A counter c_f indicates the number of sampled packets received for a flow f. A flow is removed from the list if it is offloaded or if no further sampled packet has been received from that flow for longer than $t_{list}^{max} = 2\,\text{s}$. In addition, the maximum counter value c_{max} is determined with every sampled packet and a moving average c_{max}^{avg} over consecutive values of c_{max} is tracked using UTEMA [14] with a memory of $M_{cntr} = 0.5\,\text{s}$. The memory is chosen to cover multiple packet samples on the one hand and to quickly forget about past flows on the other hand. Experiments have shown that results are rather insensitive to the choice of M_{cntr}.

The following actions are performed only in the presence of (imminent) congestion. The offloading probability p_{off} is computed and an accumulated offloading probability p_{off}^{acc} is incremented by that value. The accumulated offloading probability p_{off}^{acc} is zero at system start. If p_{off}^{acc} is positive and the counter of the sampled flow is the largest in the flow list, the flow is offloaded. Otherwise, the sampled flow is offloaded with a modified offloading probability $p_{off}^{mod} = p_{off}^{acc} \cdot \min\left(1, \left(\frac{c_f}{c_{max}^{avg}}\right)^k\right)$ with $k = 3$. We tested other values of k without significant difference. When a flow is offloaded, the accumulated offloading probability is decremented by 1.

VI. Proof-of-Concept Implementation and Experimental Results

In this section, we demonstrate the feasibility of the proposed approach with a proof-of-concept (PoC) implementation. We report encountered challenges and provide experimental results.

A. Proof-of-Concept Implementation

For our PoC implementation we use the networking testbed illustrated in Figure 2. It comprises an OpenFlow-capable switch, a transparent firewall, a traffic source, a traffic sink, and an OpenFlow controller.

An inside host is interconnected via a switch and a router with an outside network. The default configuration of the switch diverts all traffic leaving or entering the inside network through the firewall before forwarding it to the router or the inside host. A controller host is connected to the switch and controls the forwarding tables of the switch using the OpenFlow protocol.

KVM [15] and libvirt [16] are used for virtualization. The VMs for the controller, the inside host, and the outside host each have a dedicated physical network interface card (NIC) using PCI pass-through. The router VM uses two physical NICs. Each VM is assigned 1 GB RAM and two CPU cores. We use an HP ProCurve 5412zl with v2 modules as OpenFlow-capable switch and a Cisco ASA 5550 as a stateful firewall.

The physical machine and the VMs use the Ubuntu [17] Linux distribution as operating system. The controller runs an

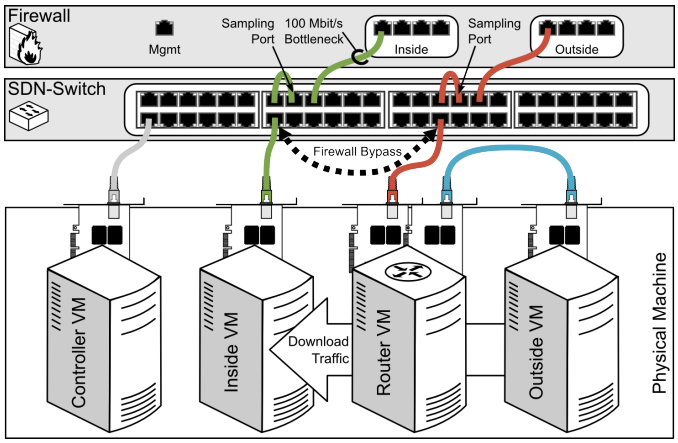


Fig. 2: The PoC implementation uses a commodity PC, an OpenFlow-capable switch, and a hardware firewall. An inside and outside VM are connected through a router with an OpenFlow switch and a transparent firewall in between that may be bypassed through a controller application.

own application based on Ryu [3] written in Python. It implements both random and intelligent offloading as described in Section V and is connected to the controller port of the switch. The inside host runs an HTTP client and the outside host runs an instance of the nginx [18] web server. The HTTP client is also written in Python. It requests files from the web server according to a Poisson process using appropriate parameters. In the experiments, downloads are performed in parallel if they cannot be fully served before the client requests new files from the server.

B. Experience of Technical Limitations

During experimentation, we faced several technical limitations of the PoC setup. Many of them are due to the usage of available hardware or our testbed setup, some others are of general nature and exclude more advanced bypass solutions.

Through experimentation we learned that at most 2042 layer-4 flow rules can be installed in hardware tables on the switch limiting the number of bypasses to 1021. However, some OpenFlow-capable switches can support up to 20000 flow rules [19]. This is still a rather low value and due to the hybrid switch design. Software-based OpenFlow switches can support significantly more flow rules with the drawback of lower forwarding performance.

According to the manual, n_s for sFlow sampling may be as small as 50. However, we conducted preliminary tests that showed significant inaccuracies on our testbed. At a port speed of 100 Mb/s, sFlow can adhere to its configured sampling rate only for $n_s \geq 147$. To add a safety margin, we chose $n_s = 200$ for the experiments.

We need to sample packets leaving the firewall, but sFlow can only sample outgoing packets on a switch port. Therefore, in our PoC implementation traffic from the firewall is fed to another switch port for sampling purposes before being forwarded to its actual destination (see Figure 2).

A firewall keeps per-flow states. They are removed either at flow termination, which is detected through TCP flags (FIN, RST), or through timeouts that are usually set to 3600 s. OpenFlow can also match for TCP flags. However, this feature is not available prior to Version 1.5, and, therefore, not supported by most switches. We use Version 1.3 in our experiments. Therefore, offloading rules on the switch should be configured with a timeout value of $t_{out} = 3600\,\text{s}$ to avoid that flows on the switch are blocked before they would be blocked on the firewall. As a result, the major fraction of rules installed on the switch belong to terminated flows and wait for a timeout. This is a rather inefficient usage of scarce forwarding resources. We set t_{out} to a smaller value of 300 s in our experiments to allow short runs of 60 minutes.

Some firewalls, e.g., Cisco ASA, implement source port randomization. If this feature is enabled, the firewall replaces the TCP source port for outgoing packets of a flow and performs the reverse operation for the TCP destination port of incoming packets. If a flow is offloaded from the firewall, the modification is no longer performed. Then, endpoints cannot match the packets of the established flow and drop them. Therefore, source port randomization must be disabled to facilitate offloading. A similar feature exists for initial TCP sequence numbers, which must also be disabled.

C. Parametrization of Experiments

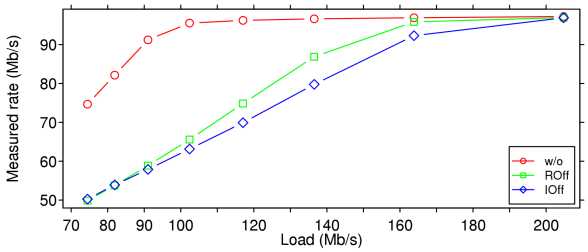
For experimentation, we apply the parameters in Table I. We limit the throughput of the firewall to $C_F^b = 100\,\text{Mb/s}$ by connecting its inside port via a 100 Mb/s link to the switch. For load measurement, sFlow is configured to sample every $n_s^{th} = 200$ packet forwarded from the firewall to the inside VM. The traffic rate on the firewall is computed with TDRM-UTEMA and a memory of $M_F = 1\,\text{s}$.

TABLE I: Parametrization of the PoC implementation for experimentation.

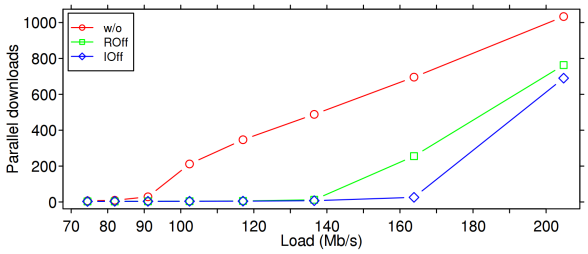
$C_F^b = 100\,\text{Mb/s}$	$n_s = 200$	$M_F = 1\,\text{s}$	$t_{ssm} = 10\,\text{s}$
$n_r = 2000$	$n_{by} = \frac{n_r}{2}$	$t_{out} = 300\,\text{s}$	$t_{reuse} = t_{out}$
$T_H = 0.8$	$M_H = 2 \cdot t_{out}$	$M_{cntr} = 0.5\,\text{s}$	$k = 3$
$E[B] = 1000\,\text{kB}$	$c_{var}[B] = 3$	λ variable	$B_{min} = 10\,\text{kB}$

We use $n_r = 2000$ flow rules so that at most $n_{by} = 1000$ TCP connections can be offloaded. We work with an inactivity timeout of $t_{out} = 300\,\text{s}$ for installed rules on the switch. This is the dominating component of the reuse time so that we set the reuse time to the same value. We define the high-load threshold to be $T_H = 0.8$ and use a memory of twice the reuse time to compute the time-averaged fraction of high-load with UTEMA. For intelligent offloading, the maximum counter is tracked using UTEMA and a memory of $M_{cntr} = 0.5\,\text{s}$, and the exponent $k = 3$ is used for the computation of modified offloading probabilities. Additional experiments showed that the choice of M_{cntr} and k has only little influence on performance results.

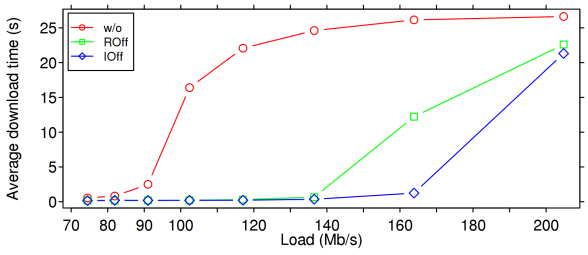
We model file sizes according to a random variable $B = B_{min} + B_{H_2}$ which is composed of a constant component B_{min} and a random component B_{H_2} that follows a hyper-



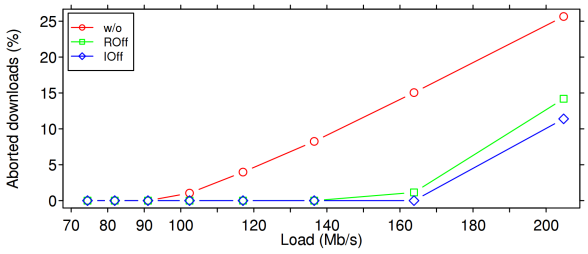
(a) Measured IP traffic rate on the firewall.



(b) Number of parallel downloads.



(c) Average download time per flow.



(d) Fraction of aborted downloads.

Fig. 3: Impact of load and offloading strategy on system performance.

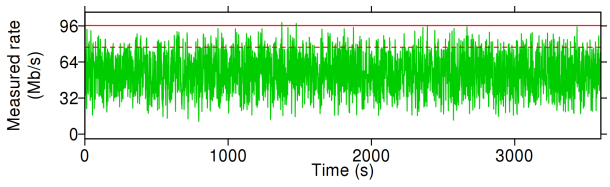
exponential distribution. This yields the following complementary distribution function:

$$P(B < x) = \begin{cases} 1 & x \leq B_{min} \\ p_0 \cdot e^{-\mu_0 \cdot (x - B_{min})} + & x > B_{min} \\ p_1 \cdot e^{-\mu_1 \cdot (x - B_{min})}. & \end{cases} \quad (2)$$

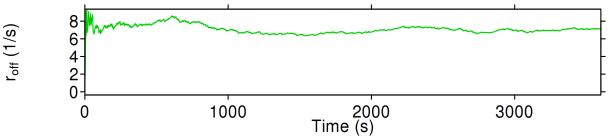
We choose an expectation of $E[B] = 1000\,\text{kB}$, a minimum file size $B_{min} = 10\,\text{kB}$, and a coefficient of variation of $c_{var}(B) = 3$ to model a few very large flows and many small ones because flow lengths usually exhibit high variance [20]. The largest flows have about 100 MB. We further determine the other parameters $p_{0/1}$ and $\mu_{0/1}$ such that $\frac{p_0}{p_1} = \frac{\mu_0}{\mu_1}$ is met. The same sets of files and also the same order is used for download in any experiment (different parameters) if not mentioned differently. Different seeds are used for several runs per experiment.

D. Experimental Results

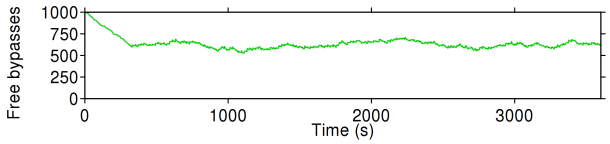
The offloading algorithms are designed to bypass flows only in case of (imminent) overload. Therefore, the observed performance results depend on the traffic load. We define the load by $\rho = \lambda \cdot E[B]$ whereby λ is the request rate of the HTTP client. To control the load, we vary the flow arrival rate value between $\frac{1}{110\,\text{ms}}$ and $\frac{1}{40\,\text{ms}}$. We first visualize the effect of congestion without offloading. We analyze the operation of offloading algorithms for imminent and moderate congestion. Then, we demonstrate the effect of random and intelligent offloading depending on the load.



(a) Measured IP traffic rate on the firewall.



(b) Target offloading rate.

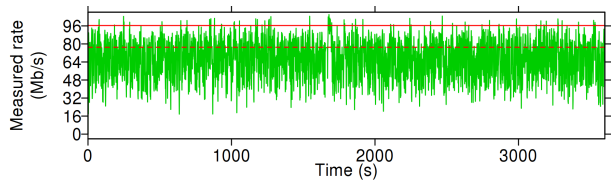


(c) Available bypasses.

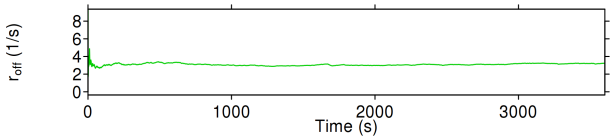
Fig. 4: Illustration of system behavior for imminent congestion ($\rho = \frac{1000\,\text{kB}}{100\,\text{ms}} = 81.92\,\text{Mb/s}$).

1) The Effect of Congestion: We conducted experiments over 60 min each. We performed 10 runs for every experiment with different seed and we provide average values in Figures 3 and 6. In the following, we refer to load levels of around 80 Mb/s, 100 Mb, and 135 Mb/s as imminent, moderate, and heavy congestion. The lines for "w/o offloading" in Figures 3(a)–3(d) illustrate the effect of congestion depending on different load. Figure 3(a) shows that the traffic rate on the firewall increases with load but reaches its limit at around 100 Mb/s. Figure 3(b) shows that the number of parallel downloads increases about linearly starting from a load of 90 Mb/s and reaches several hundreds in case of heavy congestion. The increased number of parallel downloads reduces download rates which prolongs download times (see Figure 3(c)). Some flows even face socket timeouts due to excessive delay (see Figure 3(d)). To give examples: imminent, moderate, and heavy congestion lead to download times of 0.8 s, 16.4 s, and 24.6 s and to 0%, 1.1%, and 8.5% aborted downloads.

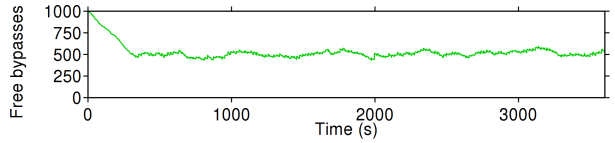
The average download time also contains the download time of aborted flows up to their abortion. Therefore, the request rate λ, the average download time $\overline{D}$, and the average number of parallel downloads $\overline{X}$ follow Little's law: $\lambda \cdot \overline{D} = \overline{X}$.



(a) Measured IP traffic rate on the firewall.



(b) Target offloading rate.



(c) Available bypasses.

Fig. 5: Illustration of system behavior for moderate congestion ($\rho = \frac{1000\,\text{kB}}{80\,\text{ms}}$=102.4 Mb/s).

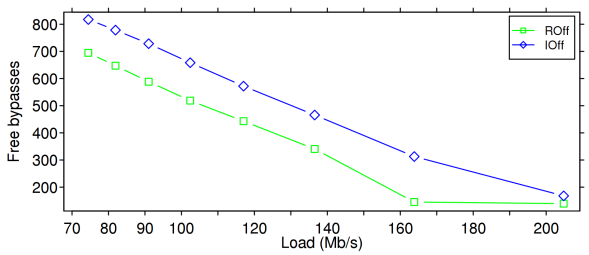
2) Illustration of Random Offloading: We illustrate the operation of random offloading during a single run, first with imminent and then with moderate congestion. Intelligent offloading produces similar results (not shown).

Figure 4(a) shows that with imminent congestion, the measured rate r_F^b on the firewall is mostly below the dashed high-load threshold T_H (converted to a rate). As a result, the offloading rates in Figure 4(b) are rather large so that multiple flows can be quickly offloaded in case of congestion. The system mostly disposes of around 625 free bypasses (see Figure 4(c)).

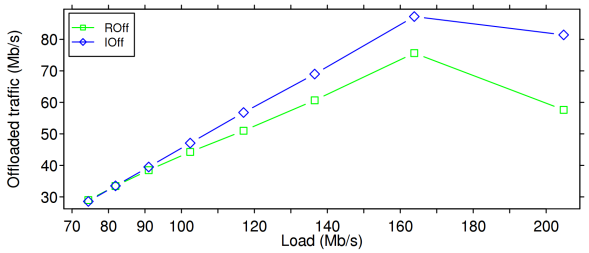
With moderate congestion, the measured rate r_F^b on the firewall in Figure 5(a) is mostly above the dashed high-load threshold T_H. Therefore, the offloading rate is clearly lower than with imminent congestion (see Figure 5(b)). Mostly around 500 bypasses are unused (see Figure 5(c)).

3) Effects of Offloading: Figure 3(a) shows that the measured rate on the firewall increases with the load, but more slowly with offloading whereby the difference between random and intelligent offloading (ROff, IOff) is small. The number of parallel connections increases significantly w/o offloading, with offloading the increase happens only at significantly higher load (see Figure 3(b)). The same holds for download times and aborted downloads in Figures 3(c) and 3(d).

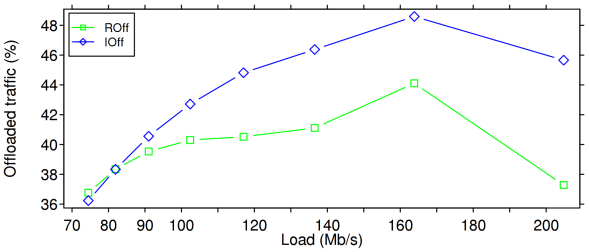
We now concentrate on the difference between random and intelligent offloading. As shown in Figure 6(a), random offloading has mostly fewer free bypasses available than intelligent offloading. Nevertheless, intelligent offloading bypasses more traffic because larger flows are offloaded (see Figure 6(b)). The offloaded traffic rate increases with offered load, but Figure 6(c) shows that the percentage of offloaded traffic is bound by about 44% for random offloading and by about 48% for intelligent offloading.



(a) Available bypasses.



(b) Offloaded traffic rate.



(c) Percentage of offloaded traffic.

Fig. 6: Impact of random and intelligent offloading.

VII. Analytical Performance Evaluation

In this section we model firewall bypassing by analytical means and use results from this model to provide performance predictions about systems with more flow rules, higher firewall capacities, different traffic models, and timeouts.

A. Model

The offloading mechanism can bypass n_{by} connections within the reuse time of the flow rules t_{reuse}. We further assume that the controller randomly samples the traffic carried on the firewall and uses the samples for offloading decisions. We approximate the traffic volume carried by the firewall within t_{reuse} time by $t_{reuse} \cdot \lambda \cdot E[B]$. As any packet is sampled with equal probability, arriving traffic of a flow is sampled and then offloaded with a rate of $\alpha = \frac{n_{by}}{t_{reuse} \cdot \lambda \cdot E[B]}$. The particular about that rate is that it relates to traffic volume instead of time. Thus, the downloaded traffic volume A until a flow is offloaded is exponentially distributed according to $P(A \leq x) = 1 - e^{-\alpha \cdot x}$. With $P(B < x)$ being the complementary distribution function of the file size, we can compute the probability that a flow is longer than x (bytes) and not yet offloaded by

$$P(B_{off}^{not} > x) = P(B > x) \cdot P(A > x). \quad (3)$$

The expectation of that value can be calculated by

$$E[B_{off}^{not}] = \int_0^\infty P(B_{off}^{not} > x)dx. \quad (4)$$

Combining Equations (2) – (4) yields

$$\begin{aligned} E[B_{off}^{not}] &= \int_0^\infty P(B > x) \cdot e^{-\alpha x} dx \\ &= \int_0^{B_{min}} 1 \cdot e^{-\alpha \cdot x} dx \\ &+ \int_{B_{min}}^\infty (p_0 \cdot e^{-\mu_0 \cdot (x - B_{min})} \\ &+ p_1 \cdot e^{-\mu_1 \cdot (x - B_{min})}) \cdot e^{-\alpha x} dx \\ &= \frac{1 - e^{-\alpha \cdot B_{min}}}{\alpha} + \frac{p_0 \cdot e^{-\alpha \cdot B_{min}}}{\alpha + \mu_0} + \frac{p_1 \cdot e^{-\alpha \cdot B_{min}}}{\alpha + \mu_1} \quad (5) \end{aligned}$$

We use the fraction of offloadable traffic $\omega = 1 - \frac{E[B_{off}^{not}]}{E[B]}$ as measure for offloading effectiveness.

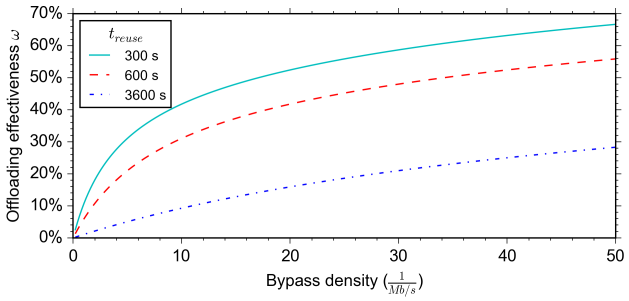
B. Performance Results

To simplify the analysis, we define the bypass density $d_{by} = \frac{n_{by}}{\lambda \cdot E[B]}$ given as a multiple of $\frac{1}{Mb/s}$. It is the number of available bypasses divided by the offered traffic rate which reflects the load. In the following, we validate the proposed analysis and investigate the offloading effectiveness under various conditions.

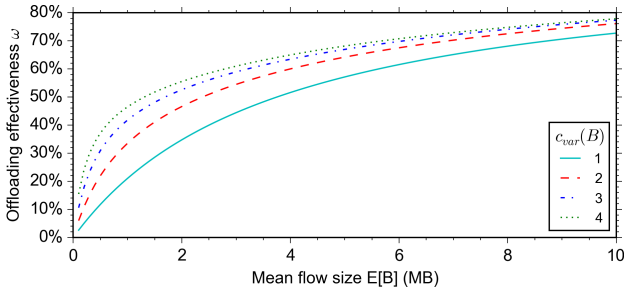
1) Validation of the Analytical Model: For the parameters of our experiment, $d_{by} = \frac{1000}{100\,\text{Mb/s}} = 10\frac{1}{\text{Mb/s}}$, $t_{reuse} = 300\,\text{s}$, $E[B] = 1\,\text{Mbyte}$, and $c_{var}(B) = 3$ we achieve an offloading effectiveness of 41.75% which is very close to the experimental results of about 41% in Figure 6(c).

2) Impact of System Parameters: Figure 7(a) illustrates that the offloading effectiveness increases with increasing bypass density d_{by} and decreases with increasing reuse time $t_{reuse} \in \{300, 600, 3600\}\,\text{s}$. When $t_{reuse} = 600\,\text{s}$ is used, a bypass density of $d_{by} = 35$ connections per Mb/s is needed to offload 50% of the traffic. That means, 14000 (140000) flow rules are needed to offload 50% from a 200 Mb/s (2 Gb/s) traffic aggregate so that it can be served by a firewall with a capacity of 100 Mb/s (1 Gb/s). Using a larger reuse time $t_{reuse} = 3600\,\text{s}$ (standard value for timeouts on firewalls) significantly reduces the offloading effectiveness. If flow rules can be removed faster by matching TCP flags, a lower t_{reuse} is obtained in spite of a large timeout value which leads to increased offloading effectiveness.

3) Impact of the Traffic Model: We now keep the bypass density constant at $d_{by} = 10\frac{1}{\text{Mb/s}}$ and vary the mean $E[B]$ and coefficient of variation $c_{var}(B)$ of the flow size. Figure 7(b) shows that offloading effectiveness clearly increases with increasing flow size $E[B]$. This is because offloaded flows have more remaining data to transmit when flows are larger. The effectiveness also increases with increasing flow size variability because this increases the size of long flows which have a larger likelihood to be offloaded.



(a) Impact of bypass density d_{by} and reuse time t_{reuse} for $E[B] = 1\,\text{MB}$ and $c_{var} = 3$.



(b) Impact of mean $E[B]$ and coefficient of variation $c_{var}(B)$ of the flow size for a bypass density $d_{by} = 10\frac{1}{\text{Mb/s}}$ and $t_{reuse} = 300\,\text{s}$.

Fig. 7: Analytic results for offloading effectiveness (percentage of offloadable traffic).

VIII. Conclusion

We have proposed static and dynamic firewall bypassing using an OpenFlow-capable switch. We presented detailed algorithms for dynamic firewall bypassing as well as a prototype implementation. The OpenFlow controller learns about congestion on the firewall using sFlow and installs appropriate bypassing rules on the OpenFlow-capable switch in case of (imminent) congestion. As flow rules are limited, we developed algorithms to make efficient use of them over time taking the congestion level into account. Experimental results demonstrated the feasibility of the suggested approach and illustrated the operation and performance of the mechanisms. We proposed an intelligent offloading strategy which is more efficient than random offloading. We derived an analytic performance evaluation model and validated its accuracy with experimental data. Analytic results showed that a large percentage of the overall traffic can be offloaded from the firewall. However, this requires a large number of flow rules and/or short reuse times of flow rules. The performance also benefits from a large mean and a large variance of flow sizes.

Acknowledgment

The authors thank Robert Finze, Jochen Kraemer, and Mario Hock for valuable input and stimulating discussions, and the university datacenters at Tuebingen (ZDV) and Karlsruhe (SCC) for providing testbed hardware.

This work was supported by the bwNET100G+ project which is funded by the Ministry of Science, Research and the Arts Baden-Württemberg (MWK). The authors alone are responsible for the content of this paper.

REFERENCES

[1] W. Braun and M. Menth, "Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices," *Future Internet*, vol. 6, no. 2, 2014.

[2] Open Networking Foundation, "OpenFlow Switch Specification," 2012.

[3] Ryu SDN Framework Community, "Ryu," http://osrg.github.io/ryu/.

[4] P. Phaal and M. Lavine, "sFlow Specification Version 5," 2004.

[5] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging Zipf's Law for Traffic Offloading," *ACM SIGCOMM Computer Communications Review*, vol. 42, no. 1, 2012.

[6] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite CacheFlow in Software-defined Networks," in *ACM Workshop on Hot Topics in Software Defined Networks (HotSDN)*, Aug. 2014.

[7] S. Zerkane, D. Espes, P. L. Parc, and F. Cuppens, "Software Defined Networking Reactive Stateful Firewall," in *IFIP International Information Security and Privacy Conference (SEC)*, 2016.

[8] J. Collings and J. Liu, "An OpenFlow-Based Prototype of SDN-Oriented Stateful Hardware Firewalls," in *IEEE International Conference on Network Protocols (ICNP)*, Oct. 2014.

[9] J. Deng, H. Hu, H. Li, Z. P, K. Wang, G. Ahn, J. Bi, and Y. Park, "VNGuard: An NFV/SDN Combination Framework for Provisioning and Managing Virtual Firewalls," in *IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN)*, 2015.

[10] C. Lorenz, D. Hock, J. Scherer, R. Durner, W. Kellerer, S. Gebert, N. Gray, T. Zinner, and P. Tran-Gia, "An SDN/NFV-Enabled Enterprise Network Architecture Offering Fine-Grained Security Policy Enforcement," *IEEE Communications Magazine*, vol. 55, no. 3, 2017.

[11] S. Miteff and S. Hazelhurst, "NFShunt: a Linux Firewall with OpenFlow-Enabled Hardware Bypass," in *IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN)*, 2015.

[12] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The Science DMZ: A Network Design Pattern for Data-intensive Science," in *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[13] A. Pani, "Scalable Stateful Firewall Design in Openflow Based Networks," Jul. 2014, US Patent 8,789,135.

[14] M. Menth and F. Hauser, "On Moving Averages, Histograms and Time-Dependent Rates for Online Measurement," in *ACM/SPEC International Conference on Performance Engineering (ICPE)*, Apr. 2017.

[15] A. Kivity *et al.*, "kvm: the Linux Virtual Machine Monitor," in *Linux Symposium*, 2007.

[16] Red Hat Inc., "libvirt: The Virtualization API," http://libvirt.org, 2012.

[17] Canonical Ltd., "Ubuntu 16.04 LTS (Xenial Xerus)," http://releases.ubuntu.com/16.04/, Apr. 2016.

[18] I. Sysoev *et al.*, "nginx 1.10.0," http://nginx.org/en/, Apr. 2016.

[19] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: Scalable Ethernet for Data Centers," in *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[20] M. E. Crovella and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, 1997.