A Middlebox-Based Policy Enforcement Architecture for SDN Network Information Service Centers: Postprint

Authors: Li Hailong, Zhang Zhao, Dong Siqi, Hu Lei

Date: 2018-11-29T00:00:00+00:00

Abstract

A middlebox is a device for which network administrators manually configure behavior policies. The advent of Software-Defined Networking (SDN) has diversified the possibilities for middlebox policy enforcement. To enhance the security protection of information service centers, we propose a dynamic middlebox policy enforcement architecture based on SDN that can respond to network events without administrator involvement, along with an interface that facilitates communication between the controller and middleboxes. A middlebox prototype incorporating a firewall and Intrusion Prevention System (IPS) was implemented in a virtual machine to evaluate the policy enforcement architecture and validate the experimental results obtained from the prototype. The results demonstrate that this architecture can dynamically enforce middlebox policies without impacting network performance, enabling normal operation of network applications.

Full Text

Preamble

Middleware-based SDN Network Information Service Center Policy Implementation Framework

Li Hailong, Zhang Zhao, Dong Siqi, Hu Lei
(School of Military Operation Support, Rocket Force University of Engineering, Xi' an 710025, China)

Abstract: A middlebox is a device whose behavior policies are manually configured by network administrators. The advent of Software-Defined Networking (SDN) has diversified the possibilities for middlebox policy implementation. To

improve the security protection of information service centers, this paper proposes a dynamic middlebox policy enforcement architecture based on SDN that can respond to network events without administrator involvement, along with an interface specification to enable communication between the controller and middleboxes. We implemented a middlebox prototype incorporating a firewall and Intrusion Prevention System (IPS) in a virtual machine to evaluate the policy execution framework and verify the experimental results obtained from the prototype.

Keywords: middlebox; SDN; policy enforcement

0 Introduction

The centralized control view in SDN makes security system automation feasible. The security protection system for information service centers ensures reliable operation of information service infrastructure, controlled access to information resources, and secure information exchange between different domains, while defending against external attacks and unauthorized access. This system comprises three components: the information service center LAN security protection system, the information service protection system, and the inter-domain information transfer security control system. The LAN security protection system primarily deploys firewalls, intrusion detection, antivirus, and virtual machine security protection systems, employing autonomous control and trusted computing measures to enhance system security capabilities and ensure reliable operation of information service center infrastructure.

In traditional network configurations, security system deployment relies heavily on manual operations by network administrators, who develop corresponding defense strategies for different network threats. However, these manual security measures consume significant effort while delivering relatively low defense efficiency. SDN technology, with its programmable devices and network-wide visibility, offers new possibilities for addressing these challenges.

1 Related Research

A middlebox is a network element that performs functions beyond simple switching and routing in computer networks [1]. They play critical roles in security enhancement (e.g., intrusion detection/prevention systems and firewalls), performance improvement (e.g., WAN optimizers and proxies), and IP address protection (e.g., Network Address Translation). Middlebox behavior is typically defined by manually and proactively configured policies that require network administrators to predict possible network events. This approach suffers from several drawbacks:

a) The policy configuration process is error-prone due to its complexity and heavy reliance on human intervention;

b) Network operations may not handle events correctly or in a timely manner;

c) Many modern applications are dynamic and establish connections that are difficult to predict [2].

In software-defined networks, the data forwarding and control planes are separated, with one or more central controllers participating in network rule configuration. This separation provides better network device programmability [3]. The centralization of control functions and programmability of network devices creates new possibilities for handling middleboxes and their configuration. Several approaches have been introduced in the literature [4–8]. In [4,5], SDN controllers use southbound interfaces to remove or add policies. Reference [4] designs APIs for joint control, while [5] proposes an architecture that enforces policies by migrating middlebox instances. However, both methods still require administrators to manually configure policies when internal states change, thus failing to eliminate human control from the process.

Reference [6] proposes the FlowTags architecture, which modifies specific internal functions of enhanced middleboxes to improve controller scalability and reduce packet processing delays. Reference [7] suggests dynamically configuring switches to forward traffic through a sequence of middleboxes (e.g., firewall → proxy → NAT). Since changes made by middleboxes may cause packets to be forwarded incorrectly, an SDN controller maps packet information before and after middlebox processing. This mapping is performed solely by the controller without any interaction with the middleboxes themselves. Both [6] and [7] discuss policy setting on individual middleboxes but may interfere with correct policy configuration on other middleboxes, resulting in suboptimal operational convenience.

Reference [8] modifies the SDN controller to obtain middlebox modification content and can notify middleboxes of changes made by other devices, using a specific southbound interface for communication between the controller and middleboxes. However, the architecture described in [8] focuses on ensuring the shortest path between source and destination includes all required middleboxes, without establishing SDN controller-middlebox interaction. In summary, while these works propose methods to address packet modification issues using SDN features, none eliminate the manual policy configuration process in middleboxes —a complex task for administrators. Furthermore, some approaches like [7] and [8] do not consider communication between SDN controllers and middleboxes, reducing their integration capability and limiting the advantages of centralized control.

This paper proposes a middlebox policy enforcement architecture that leverages the programmable interfaces enabled by centralized SDN controllers. This architecture allows middleboxes to handle dynamic network applications and respond to network events without manual configuration. A software agent acts as an intermediary between the SDN controller and middleboxes while using the

OpenFlow protocol to interact with infrastructure devices without requiring any OpenFlow modifications.

2 Policy Enforcement Mechanism

The proposed policy enforcement mechanism alleviates the burden of middlebox configuration in SDN networks by exploiting the advantages of programmable devices and centralized control. It enables dynamic middlebox policy enforcement with minimal changes to conventional network interfaces.

As shown in Figure 1 [Figure 1: see original paper], the proposed architecture uses the centralized view of the SDN controller to dynamically and automatically select optimal policies based on data obtained from middleboxes. An optional software agent ensures interaction between the controller and middleboxes. The agent retrieves existing data from middleboxes and sends it to applications running on the controller for analysis alongside the centralized flow information. For example, the agent can access logs or configuration files stored in the middlebox and forward this data to applications. The agent remains transparent to both middleboxes and controllers, ensuring compatibility with current SDN standards while leaving middlebox functionality unchanged.

Additionally, the controller' s complete network view enables policy optimization. This approach allows middleboxes to communicate with the controller during operation so that the controller can use this information to configure other middleboxes or switches. For instance, an IPS can identify malicious traffic and notify the controller, which can then forward this traffic to another IPS for deeper inspection or configure a switch to drop the traffic. These operations can be automatically controlled by applications running on the controller or sent to administrators for further analysis. While network administrators still play a key role in middlebox configuration, they can create applications to automate this task, applying policies to multiple middleboxes simultaneously rather than configuring each one sequentially. If administrators pre-authorize policy changes, policies can be automatically modified based on network events, making middleboxes more responsive.

The example described and evaluated in this paper involves an application running on the controller that uses a software agent to configure a firewall. This application analyzes packets received by the controller to determine whether the firewall should permit or deny communications with specific characteristics. The prototype uses the open-source firewall (Iptables [9]) and IPS (Snort [10]) under Python, with software agents detailed below introducing each component of the proposed architecture.

2.1 Controller and Applications

The proposed architecture is implemented using a controller based on the Ryu framework. Ryu is an actively maintained open-source framework for SDN

that supports the latest OpenFlow versions and various other protocols, including OpenStack [11]. When the controller receives a packet-in message from a switch, it passes the message to running applications. These messages contain information characterizing network traffic records, which are extracted and used to plan policies configured in middleboxes.

2.2 Communication with Middleboxes

Proprietary middleboxes typically provide application interfaces for external agent interaction and support application development to extend their functionality. However, operational control is restricted by manufacturers, limiting middlebox modifications. Conversely, open-source middleboxes can be modified according to developer needs, though such modifications are complex and costly.

For these reasons, the proposed architecture provides two options for controller-middlebox communication: (a) modifying the middlebox by adding APIs; (b) employing an intermediary agent when middleboxes are proprietary or too complex to modify.

In both options, data is exchanged using JavaScript Object Notation (JSON), which enables compact data transmission for faster transfer speeds. Policies are transmitted in the following format: - Packet header: traffic information processed by the policy (e.g., IP addresses, ports) - State: policy status (e.g., allow or deny actions executed by firewall block/release, or proxy cache status)

An example request to add a policy to a middlebox is as follows:

```
{
  "context": {
    "src_ip": "192.168.1.10",
    "dst_ip": "168.168.1.11",
    "src_port": "54532",
    "dst_port": "4345",
    "action": {"action": "block"}
  }
}
```

The interface also issues responses to controller requests. Successful requests return HTTP code 200, while unsuccessful requests return error codes with messages describing the problem.

When the software agent communicates with the controller via API addition, the added API is integrated directly into the middlebox using the same data transmission format, resulting in relatively small transmission overhead. When using an agent to modify a middlebox is too complex, the cost of modifying and debugging the middlebox itself far exceeds that of adding individual agents, making the agent approach more cost-effective in such scenarios.

Regarding security during API-controller interaction, since the API is added

directly to the middlebox, it communicates with the controller in the control plane while being isolated from other planes. According to SDN characteristics, this isolation provides clear plane separation. Furthermore, SDN employs security measures including authentication accounts before interaction and IPS protection during communication, which can secure the API-controller interaction.

2.3 Software Agent

As mentioned, the software agent can serve as an alternative to middlebox APIs. It is an optional component designed to simplify policy enforcement adoption, particularly when middlebox code is proprietary or too complex to modify. The agent transparently proxies requests between the controller and middleboxes, implementing either pre-existing APIs provided by middlebox vendors or independent APIs based on data stored in middleboxes to retrieve state information.

2.4 Middlebox State and Data Storage

Middleboxes have various types, each with its own way of representing internal state. However, all middleboxes share a common characteristic: they inspect and manipulate network traffic. Therefore, their state is generally represented by operations performed on traffic. This paper adopts an approach similar to [5] that stores traffic modification information within the controller, enabling faster application access compared to external storage [12].

3 Experimental Validation and Results Analysis

3.1 Experimental Setup

To validate the policy enforcement architecture, we implemented a prototype using Mininet 2.2.1 in a virtual machine (VM). For evaluation, we selected two middleboxes to cover common production environments: a firewall (Iptables [13]) and an IPS (Snort [14]), both using generic open-source software. We chose Python to develop software agents that retrieve and set policies in each middlebox.

The application running on the controller receives packet-in messages and searches for predefined information such as IP addresses, TCP/UDP ports, and application-layer protocols. This information is used to create and set appropriate policies in middleboxes. Once the application receives confirmation of successful policy configuration, the controller updates flow rules to allow traffic forwarding through network devices to the middlebox.

Algorithm 1 shows an example application for configuring firewall policies. It permits VoIP communications through the firewall using dynamic ports with Real-time Transport Protocol (RTP) and Session Initiation Protocol (SIP).

Algorithm 1: Algorithm used for policy configuration on a firewall

```
while application is running do
  receive the Packet-in
  if There is no policy with such information then
    for all Protocol header in the packet-in do
      if Package from link layer then
        Extract source and destination MAC addresses
      else if Packet from the network layer then
        Extract source and destination addresses
      else if Packet from transport layer then
        Extract destination port
        Update the flow rule
        if UDP packet then
          Set the flow rule as a UDP flow
        else if RTP or SIP packet then
          Create a policy with information extracted from the packet-in
          Send the policy to the firewall
          if Policy successfully configured then
            Update the flow rule to forward the data to its destination
    Send the flow rule to the switch
  else
    bypass
```

The prototype was evaluated in a set of simulations using two types of middleboxes (IPS and firewall). To verify experimental results, we conducted hypothesis testing. The evaluation aimed to verify whether the proposed mechanism could be implemented without affecting network and application performance. The prototype was expected to dynamically modify firewall policies and receive policies from the IPS without impacting relevant performance metrics. These experiments used VoIP communications running on enhanced Simple Internet Protocol (SIPp).

Experimental Topology and Configuration

The topology used in the experiments is shown in Figure 2 [Figure 2: see original paper]. We employed Asterisk software [25] as the VoIP call center, an Iptables firewall to process all traffic between clients and the center, a Snort IPS to analyze network traffic and identify flows to the VoIP center, a Ryu controller to execute policy enforcement applications, and client hosts to initiate VoIP calls. All experiments used RTP for data transmission and SIP for signaling. The IPS identified VoIP calls based on payload analysis, with two rules for detecting SIP and RTP packets. In the experiments, Snort IPS was configured to search for SIP and RTP patterns in packet payloads using its "content" keyword.

Before experiments began, the firewall was configured to block any connections to destination ports above 1023, with all other ports blocked. This configuration typically causes problems because VoIP communications may use dynamic high ports. The policy enforcement prototype should be able to passively configure the firewall without manual intervention to mitigate this issue.

During experiments, the prototype sent and received policies. However, policies received from the IPS were not used for firewall configuration, which was instead based on packet-in messages received by the controller. Table 1 shows example firewall policies configured by the SDN controller during experiments, while Table 2 shows rules applied by the IPS during implementation. In these tables, "any" represents any address or port, and the "content" column indicates patterns searched by the IPS in packets.

The topology in Figure 2 was emulated on Mininet with client hosts using SIPp to generate 50 simultaneous calls. RTP traffic was generated from preloaded capture files in SIPp. When SIPp execution completed, it produced statistics including total calls, successful calls, response counts, and failed calls. The experiment generated 20,000 calls in total.

Performance Metrics

The evaluation considered two scenarios: dynamic policy (implementing the proposed architecture) and static policy (using static middlebox configurations). The network impact of the proposed architecture was measured by additional latency or response time shown in SIPp output, jitter obtained via Wireshark [15], and packet loss. The performance metrics are defined as:

a) **Latency or Response Time:** The time required for the network to fully respond to application requests. For VoIP calls, this is the delay caused by transmitting packets from source to destination.

b) **Packet Loss:** The number of lost packets relative to the total packets exchanged by the application. Packet loss degrades user experience, particularly for real-time interactive applications.

c) **Jitter:** Variation in packet delivery delay across the network.

3.2 Results Analysis and Comparison

In experiments with both firewall and IPS, all VoIP calls were successful, demonstrating that the proposed policy enforcement architecture successfully manipulated firewall rules as needed. The prototype also successfully obtained IPS policies. No packet loss occurred in any case.

Table 3 shows VoIP call response times for static and dynamic policy scenarios, while Table 4 compares jitter. The differences in call delay and jitter are minimal, indicating that performance with and without the proposed architecture is very similar.

To confirm the hypothesis that prototype performance shows no significant change (i.e., prototype results are statistically equivalent to static policy results at 95% confidence), we performed a Kolmogorov-Smirnov (KS) test on jitter and response time results. The p-value calculated using R software was less

than 0.05, indicating the data does not follow a normal distribution. Therefore, we applied non-parametric tests considering medians rather than means.

We selected the Wilcoxon Signed-Rank Test (WSRT) for paired data, as all data came from the same network. The dynamic policy scenario represented the experimental group, while the static policy scenario served as the control group. The test considered n = 20,000 call samples.

Hypothesis Testing

For response time metrics, the paired WSRT yielded a p-value of $H_0 : \mu_d = 0$ (prototype does not affect performance metrics) versus $H_1 : \mu_d \neq 0$ (prototype affects performance). This led to rejecting H_0, concluding that medians are statistically different. However, the dynamic policy scenario showed shorter response times than static policy. Therefore, a second WSRT was performed to evaluate whether dynamic policy demonstrated better performance:

$H_0 : \mu_d = 0$ (dynamic policy median equals static policy median)
$H_1 : \mu_d < 0$ (dynamic policy median is less than static policy median)

In this test, the p-value was 1, so H_0 was not rejected. Thus, the median response time of the dynamic prototype is statistically lower than that of static policy, meaning the prototype does not negatively impact VoIP call response time.

For jitter metrics, WSRT returned a p-value of $H_0 : \mu_d = 0$ versus $H_1 : \mu_d \neq 0$, leading to rejection of H_0. Following the response time analysis, we tested:

$H_0 : \mu_d = 0$
$H_1 : \mu_d < 0$

The resulting p-value was 1, not rejecting H_0. Therefore, the prototype has no negative impact on jitter. In conclusion, the implemented architecture does not add burden to the original network.

4 Conclusion

This paper proposes a novel architecture for dynamically enforcing middlebox policies in SDN networks. Although middleboxes are essential for network operation, they have traditionally been configured manually and statically by administrators, making them error-prone and inflexible for handling modern applications. In the proposed architecture, SDN is used to automatically configure optimal middlebox policies.

The architecture was implemented as a running prototype in a Mininet emulation environment and evaluated through experiments with two widely-used middleboxes (firewall and IPS) under different network scales to assess scalability. The experiments considered VoIP applications and analyzed relevant performance metrics (response time, packet loss, and jitter). The prototype

successfully enforced middlebox policies automatically and dynamically without affecting network performance in all evaluated scenarios. Therefore, the presented architecture proves to be an effective option for configuring middlebox policies in SDN networks, offering a new approach to leveraging centralized programmability in SDN architectures to make middlebox configuration easier and more efficient.

References

[1] Carpenter B, Brim S. RFC3234, Middleboxes: taxonomy and issues [S].
[2] Stiemerling M, Quittek J, Cadar C. RFC4540, NEC' s simple middlebox configuration (SIMCO) protocol version 3.0 [S]. 2006.
[3] Katti S, Levis P, McKeown N, et al. Software-defined networking [EB/OL]. https://ee.stanford.edu/research/software-defined-networking.
[4] Gember-Jacobson A, Viswanathan R, Prakash C, et al. OpenNF: enabling innovation in network function control [J]. ACM SIGCOMM Computer Communication Review, 2014, 44(4): 163-174.
[5] Gember A, Prabhu P, Ghadiyali Z, et al. Toward software-defined middlebox networking [C]// Proc of ACM Workshop on Hot Topics in Networks. New York: ACM Press, 2012: 7-12.
[6] Fayazbakhsh S K, Chiang L, Sekar V, et al. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags [C]// Proc of USENIX Conference on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2014: 533-546.
[7] Qazi Z A, Tu C C, Chiang L, et al. SIMPLE-fying middlebox policy enforcement using SDN [J]. ACM SIGCOMM Computer Communication Review, 2013, 43(4): 27-38.
[8] Cao Zizhong, Kodialam M, Lakshman T V. Traffic steering in software-defined networks: planning and online routing [J]. ACM SIGCOMM Computer Communication Review, 2014, 44(4): 65-70.
[9] Xuan Leifei, Wu Peifei. The optimization and implementation of iptables rules set on Linux [C]// Proc of International Conference on Information Science and Control Engineering. 2015: 988-991.
[10] Alhomoud A, Munir R, Disso J P, et al. Performance evaluation study of intrusion detection systems [J]. Procedia Computer Science, 2011, 5(9): 173-180.
[11] RYU project team. Ryu SDN framework [EB/OL]. [2018-05-05]. https://osrg.github.io/ryu.
[12] Krishnamurthy A, Chandrabose S P, Gember-Jacobson A. Pratyaastha: an efficient elastic distributed SDN control plane [C]// Proc of Workshop on Hot Topics in Software Defined Networking. New York: ACM Press, 2014: 133-138.
[13] Netfilter Core Team. The netfilter.org "iptables" project [EB/OL]. [2018-05-05]. http://www.netfilter.org/projects/iptables/.
[14] Snort Core Team. Snort: intrusion detection/prevention system [EB/OL]. [2018-05-05]. https://www.snort.org/.

[15] Orebaugh A, Ramirez G, Burke J, et al. Wireshark & Ethereal network protocol analyzer toolkit (Jay Beale' s open source security) [C]// Syngress Publishing, 2006: 523-540.
[16] Atary A, Bremler-Barr A. Efficient round-trip time monitoring in OpenFlow networks [C]// Proc of IEEE International Conference on Computer Communications. Piscataway, NJ: IEEE Press, 2016: 1-9.
[17] Espinet F, Joumblatt D, Rossi D. Framework, models and controlled experiments for network troubleshooting [J]. Computer Networks, 2016, 107: 36-54.
[18] Wang Bing, Zheng Yao, Lou Wenjing, et al. DDoS attack protection in the era of cloud computing and Software-Defined Networking [J]. Computer Networks, 2015, 81(C): 308-319.
[19] Femminella M, Francescangeli R, Giacinti F, et al. Design, implementation, and performance evaluation of an advanced SIP-based call control for VoIP services [C]// Proc of IEEE International Conference on Communications. Piscataway, NJ: IEEE Press, 2009: 1465-1469.
[20] Rafique M Z, Akbar M A, Farooq M. Evaluating DoS Attacks against SIP-Based VoIP Systems [C]// Proc of Global Telecommunications Conference. Piscataway, NJ: IEEE Press, 2010: 1-6.
[21] Chamorro V G, Castillo C N, Lopez-Pires F. An Elastic VoIP Solution Based on OpenStack [C]// Proc of International Conference on Information Systems Engineering. Piscataway, NJ: IEEE Press, 2016: 43-47.
[22] Stanek J, Kencl L. SIPp-DD: SIP DDoS flood-attack simulation tool [C]// Proc of International Conference on Computer Communications and Networks. Piscataway, NJ: IEEE Press, 2011: 1-7.
[23] Yang H Y, Lee K H, Ko S J. Communication quality of voice over TCP used for firewall traversal [C]// Proc of IEEE International Conference on Multimedia and Expo. Piscataway, NJ: IEEE Press, 2008: 29-32.
[24] Zhou Dawen, Huang Benxiong, Mo Yijun. Distributed architecture of VOIP for firewall/NAT traversing [C]// Proc of International Conference on Wireless Communications, Networking and Mobile Computing. Piscataway, NJ: IEEE Press, 2005: 1160-1163.
[25] Imran A, Qadeer M A, Khan M J R. Asterisk VoIP private branch exchange [C]// Proc of International Conference on Multimedia, Signal Processing and Communication Technologies. Piscataway, NJ: IEEE Press, 2009: 217-220.

Note: Figure translations are in progress. See original paper for figures.