Software Defined Layer 4.5 Customization for Agile Network Operation

Daniel Łukaszewski and Geoffrey G. Xie
Department of Computer Science
Naval Postgraduate School
{dflukasz, xie}@nps.edu

Abstract—Protocol customizations primarily come in two forms: those driven by public extensions to open standard protocols; and dialecting and performance tuning driven by an enterprise network’s private security and performance needs. Current deployment of protocol customizations is mostly ad hoc, through manual configuration or script programs that are highly specialized to each customization. This method lacks the agility necessary to support the relatively high tempo of private customizations. Also, it is common for today’s protocol customization efforts to experience middlebox interference.

We propose a systematic framework of network-wide orchestration and continuous management of protocol customization to enable agile operation for enterprise and data-center networks. By introducing a logically centralized orchestrator along with a Layer 4.5 fine-grained device customization solution, our framework will allow operators to configure, deploy, and monitor customized flows from a single vantage point, providing timely detection of rogue devices as well as real-time coordination of middlebox traversal. Results from prototyping and experimentation confirm utility of our framework while incurring modest processing overhead, at the levels of 3% and 0.5% for sample customized flows and non-customized flows, respectively. Furthermore, we present two major system refinements: (i) generalizing the design of receiving modules to support customization of encrypted flows, and (ii) adding logic system wide to support seamless rotation of customization modules for live flows. Finally, we discuss specific agile network operation use cases enabled by our solution and outline future work.

Index Terms—protocol customization, layer 4.5, agile network operation, software defined control

I. INTRODUCTION

Traditionally, protocol customization is about extending existing protocols with additional features, primarily for performance and security reasons. We observe that two recent trends have significantly expanded the use of protocol customization, particularly above the transport layer. First, protocol dialecting advocates restricting features of application protocols (such as HTTP) [1], [2] and/or intentionally varying application message formats and messaging patterns [3] within an enterprise network to add a layer of defense against external threats. Second, users of 5G and other emerging technologies such as edge computing should expect sustained performance despite frequent hand-offs (even if between different edge network providers). It is highly desirable that each time a new connection is made, systems on both ends should be able to agree upon and optimize the performance of a common application specific protocol on the fly [4]–[6]. Additionally, edge servers may activate direct hand-off of application flows to backend data centers at user devices and signal for special processing via provider specific packet tagging, in the same manner as packet tagging is used for expedient routing with MPLS [7].

In both use cases above, the customization needs are unique to individual networks and, more importantly, operators actively employ protocol customization as a method to strengthen security and/or enhance performance. The timescale of intervals for such active customization is likely measured by days or even hours, and it should continue to decrease as more use cases arise. We refer to the ability to perform per network customization at such high tempo as agile network operation. We argue that because of its inherent management overhead and limited coordination with middlebox operation, the current ad hoc deployment of protocol customization, i.e., through manual configuration or scripts that are highly specialized to each customization, severely impedes the agility of enterprise networks and data-centers, which are large in size and must uphold stringent security and performance requirements at all time [8]. Therefore, in this paper we explore an approach based on network-wide orchestration, by leveraging the growing adoption of software defined networking (SDN) in enterprise and data-center networks. As we will demonstrate, introducing a Network-wide Customization Orchestrator (NCO) — as illustrated on the left side of Figure 1 — allows operators to deploy and continuously monitor protocol customization on all devices from a single vantage point and furthermore, provide real-time coordination of middlebox traversal to address well known interference problems [9]–[11].

A straightforward method of using an SDN controller to support protocol customization is to virtualize all devices in

Fig. 1. Proposed architecture for centralized control of protocol customization in a network. A Network-wide Customization Orchestrator (NCO) deploys customization modules as Layer 4.5 kernel extensions on devices and continues to monitor their operation and coordinate middlebox traversal.
the network and deploy completely new VMs to targeted devices from the controller when a new customization requirement arises. However, this method may introduce significant downtime during the migration of VMs. Therefore, in this paper we explore a design that supports dynamic “hot” insertion of software modules to devices on the fly, without rebooting them. Moreover, we focus on supporting application layer protocol customization as an initial step. Customizations at the application layer will likely be more frequent than at lower layers for enterprise networks and data centers and as such, they would benefit the most from the agility that network-wide orchestration can provide via SDN style automation and flow level control. In Section VII, we will discuss ways to extend our design to support customization of protocols at transport or lower layers.

Modern operating systems provide a rich set of mechanisms [12]–[14] to upgrade software of a device at virtually all layers without rebooting. Since we focus on application layer customization, we take application transparency, i.e., requiring no changes to existing application software, to be a primary design goal. Meeting this goal necessitates that we tap into and modify application messages outside applications, while the messages traverse the device’s protocol stack below the application layer. Additionally, multiple different applications (Chrome, Firefox, wget, curl, etc.) invoke the same application protocol (HTTP). Recent work towards network-application integration [15]–[17] suggests a need to differentiate application processes when performing customization. Therefore, to allow customization granularity on a per-application process level even in cases where targeted processes are yet active, while avoiding transport protocol modification, we have chosen to tap into application messages when they arrive at socket buffers, right before they are passed down to the transport protocol on the sender end, and right after the transport layer finishes processing on the receiver end. Conceptually — as illustrated on the right side of Figure 1 — our customization taps constitute a shim layer between the application and transport layers, which we call “Layer 4.5.”

Our approach of network-wide orchestration together with Layer 4.5 data manipulation provides a general solution for agile network operation above the transport layer. Operators can choose between two levels of granularity to specialize application payloads of TCP and UDP flows between each pair of end devices: (i) per flow enhancement (e.g., implementation of per packet erasure coding to increase reliability [18]); or (ii) per application enhancement (e.g., dialecting of all or a subset of HTTP/HTTPS or DNS traffic to strengthen security [3]).

In the rest of the paper, we design and evaluate a prototype system of network-wide customization orchestration that consists of a centralized orchestrator and distributed agents that manage Layer 4.5 customization modules for each device. Our main contributions are as follows:

1) We propose an orchestration architecture that not only automates deployment of customization modules to devices but also provides a platform for continuous management features such as liveness monitoring, rogue module detection, and middlebox traversal.

2) We conceptualize Layer 4.5 modules to perform application-transparent, fine-grained process-level flow customization. We realize this through tapping standard socket functions, such as send and receive.

3) We prototype and conduct a preliminary evaluation of the proposed orchestration architecture and Layer 4.5 device customization.

4) We generalize the Layer 4.5 architecture to allow customization of encrypted and unencrypted application flows. The Layer 4.5 device customization overhead is re-evaluated under the revised architecture.

5) We define a process to rotate customizations on active network flows without disrupting application communications in support of agile network operations.

Specifically, the paper’s organization is as follows. Section II discusses recent protocol customization efforts. Details of our design are presented in Section III. In Section IV we describe an initial prototype of the design and evaluate the overhead of its major components. Based on lessons learned from prototyping, Section V generalizes the Layer 4.5 design to allow customization of encrypted flows. Section VI discusses active flow customization and agile network operation use cases our framework enables. Finally, Sections VII and VIII wrap up the paper with a discussion of the limitations of Layer 4.5 customization and future extensions to the framework.

II. RELATED WORKS

We review related work on five fronts. First, we discuss our previously published conference results and the extensions provided in this paper. Second, we discuss two efforts that investigated orchestration of protocol customizations. Third, we describe related device stack customization methods and efforts that aim to support generalized application transparent customization. Fourth, we discuss customization efforts targeting specific applications. Finally, we consider current methods customizations use to address middlebox interference.

NetSoft Publication: This paper is an extended and revised version of a NetSoft 2022 conference report [19]. In this work we (i) improve upon the Layer 4.5 prototype to achieve reduced customization processing and distribution overhead, (ii) generalize the Layer 4.5 design to apply to all applications and highlight the ability to customize both encrypted and unencrypted flows, and (iii) focus on agile network customization to include emphasizing additional use cases.

Orchestration of Customizations: The L3AF project [20] and the protocol plugin work [21] provide support for protocol customization via a distribution channel. L3AF aims to support kernel functions as a service via a central repository and leverages the eBPF programmability of the kernel to target the eXpress Data Path (XDP) and traffic controller (tc) layers of the stack. The plugin work targets application protocol customization leveraging instrumented protocols to allow dynamically replacing device functionality via plugins negotiated and distributed over a control channel. Both of these projects provide the capability to distribute customizations on the network, but neither provides for the continuous management of deployed customizations that our orchestration framework
is designed to support. As the following sections show, this continuous management enables novel solutions for security and middlebox traversal, as well as the ability to orchestrate customization handovers on active flows in support of agile network customization.

**Application Transparent Customization:** The use of SDN and a programmable data plane gave rise to the concept of In-Band Network Telemetry (INT) [22] and Network Function Virtualization (NFV) [23]. In [24], the authors develop a framework, known as CHIMA, for deploying service function chains leveraging INT to monitor the deployment and perform updates as necessary to meet the desired network goals. CHIMA accomplishes this by leveraging an ONOS SDN controller for distribution of the service functions to network switches supporting P4 programmability and INT, while utilizing MPLS for network routing decisions. The continuous monitoring of function deployment matches that of our customization architecture and solidifies the importance of such monitoring. Our research differs from CHIMA in that we focus on customization of end-devices and the applications they are running instead of within the network. Layer 4.5 could be used in-conjunction with in-network customization, especially if the network devices can leverage the Layer 4.5 host customizations to make more informed decisions.

Two recent works performing host-based customization are the Virtual Transport Layer (VTL) [25] and the specialized application filtering work of [26] that utilize eBPF to perform application customization without modifications to the application. First, VTL customizes application flows utilizing TCP sockets by dynamically mapping the sockets to different protocols, such as UDP, to improve network performance. Second, the authors of [26] analyze the performance of using specialized socket filters based on the applications owning the socket. Both works provide customization of applications and are complementary to our efforts. However, these works do not investigate the alteration of application data and the processing/management requirements of such customizations.

**Proxy Based Customization:** First, microservice architectures utilize on-device proxies such as sidecar proxies that run alongside application containers. In [27], the authors describe using the service mesh layer present between microservice applications and the transport layer to achieve customizations by utilizing sidecar proxies as a vantage point for network customization. In [28], however, the authors implement event-based sidecar proxy behavior by utilizing eBPF capabilities in order to reduce the sidecar proxy processing overhead. Layer 4.5 differs from these works by supporting all current applications utilizing the kernel network stack and is not limited to containerized applications utilizing services provided by application companion proxies. We do believe, however, that these sidecar proxy solutions could be expanded to fit into the proposed Layer 4.5 architecture and perform Layer 4.5 customization behavior.

Second, in-network application proxies can also be used to perform protocol customization for a specific application. For example, in [3] we utilize a pair of application proxies to customize the OpenFlow protocol used in the SDN control channel. The use of application proxies for protocol customization requires directing traffic to the specific proxy, which means the traffic traverses the network stack multiple times resulting in additional processing overhead. Furthermore, application proxies are typically used for a single application or application type and modifications to the customizations they provide require changes to the proxy device. The Layer 4.5 customization architecture provides generalized support for application customization on the end device while also providing continuous management of the customization in use, which enables customizations that can be deployed and remain transparent to the application.

**Middlebox Support:** It is well known that the current protocol extension methodology suffers from middlebox interference [9]–[11]. To specifically address interference from middleboxes conducting deep packet inspection, some protocols leverage application encryption [29], [30]. Layer 4.5 customization could also leverage encrypted application traffic to bypass middlebox interference by either applying the customization to the encrypted traffic or by utilizing embedded encryption keys to encrypt the customization and application data.

In SDN environments, middleboxes can be virtualized and supported with OpenBox [31], which leverages the common processing conducted by multiple packet inspection services to reduce redundant processing. Layer 4.5 may be able to integrate with OpenBox by treating protocol customization as a necessary middlebox processing step. The challenge comes with translating the Layer 4.5 customization specification into the click modules used in the OpenBox architecture.

### III. Design of Layer 4.5 Customization Framework

In this section we present the architectural design of the Layer 4.5 customization framework. As illustrated in Figure 1, the framework consists of a Network-wide Customization Orchestrator (NCO) responsible for the management and distribution of per-device customization modules via a customization control channel and customized devices incorporating Layer 4.5 into the TCP/IP stack. It should be noted that the NCO is a logical component that can be simply a software process running on a designated device, such as an SDN controller.

Given our emphasis on network-wide orchestration, we begin by discussing the NCO components necessary to provide customization distribution and subsequent continuous management. Next we introduce the Device Customization Agent (DCA) to support customization automation on each device. Finally, we expand on how our design supports per-network additional security and middlebox traversal requirements.

#### A. Network-Wide Orchestration

Protocol customizations under the Layer 4.5 model may be temporary and rotate often, which traditionally presents a deployment burden to network operators. To ease this burden, we include the NCO, depicted in Figure 2, as a necessary component in the Layer 4.5 architecture. The NCO has a set of distribution functions, a set of continuous management functions, and an internal Customization Information Base (CIB) to support these functions. Additionally, the NCO utilizes an
encrypted control channel to communicate with customized devices, which could be established using NETCONF or OpenFlow protected with TLS.

![Diagram](image-url)

Fig. 2. Layer 4.5 NCO consisting of “distribution” and “continuous management” functions, a CIB for tracking deployed customization modules, and an encrypted control channel to customized devices.

1) Distribution Functions: The NCO distribution functions provide centralized control and deconfliction of the network customizations in use. These functions include the ability to construct, deploy, and revoke customization modules.

Construct function: This function is responsible for building the per-device customization module to include embedding the core Table I parameters and creating a record of them in the CIB. These core parameters are designed to support a standard interface for the NCO to authenticate a deployed module, track module usage in support of continuous management functions, and ensure module attachment to the correct application flows.

TABLE I
PARAMETERS EMBEDDED PER-MODULE

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_id</td>
<td>Per-device unique module ID</td>
</tr>
<tr>
<td>active_ts</td>
<td>Timestamp of latest customization performed</td>
</tr>
<tr>
<td>init_key</td>
<td>Initial key for security functions (shared with NCO)</td>
</tr>
<tr>
<td>tap_socket</td>
<td>5-tuple ID of socket to tap (application label in place of dynamically-generated source port at client)</td>
</tr>
</tbody>
</table>

Each customization module is linked to a device or set of devices via the mod_id parameter. Customized devices use the mod_id when communicating with the NCO, thus a per-device unique ID is necessary to correctly identify the module in use. The module’s active_ts and init_key are used by the continuous management functions and are discussed in Sections III-A2 and III-C, respectively. Finally, to provide for fine-grained application customization, each module is built to match a tap_socket consisting of the standard connection 5-tuple parameters. As the NCO cannot predict the sender socket’s source port (corresponding destination port on the receiver) that is dynamically generated at run time, an application label (e.g., Chrome, curl) is used in its place. For instance, not all applications will perform a socket bind call, setting the source IP address and port, prior to establishing a connection or sending traffic. Of note, the tap_socket customization parameters can also utilize wildcard values for unknown parameters or to generalize the customization to match multiple different flows. In Section III-B we discuss how the application label is tied to a process on a tapped socket.

Deploy function: This function supports transport of customization modules, in binary format, to devices on the network. After a customization module is built, it is marked for deployment in the CIB to the device along with a deployment time. The NCO delivers the customization module over the established control channel and awaits confirmation that the module was installed. Upon confirmation, the per-module intervals from Table II are set and the CIB is updated to reflect the module’s deployed status and window values. The ability for the NCO to set the intervals is intended to provide flexibility for an operator to adjust module reporting requirements to match current network policy. These established windows are used by the continuous management functions and trigger the events shown in Algorithm 1.

TABLE II
MONITORING AND SECURITY INTERVALS PER-MODULE

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>state_req_window</td>
<td>Period between state report requests</td>
</tr>
<tr>
<td>sec_check_window</td>
<td>Period between security checks</td>
</tr>
</tbody>
</table>

Revoke function: To support the removal of outdated or misbehaving customization modules from a customized device, the NCO uses a revoke function. When a module is marked for revocation in the CIB, the NCO issues a revoke command to the appropriate device and awaits confirmation that the module has been unregistered from Layer 4.5 and removed from the host.

2) Continuous Management Functions: The NCO platform is set apart from other customization distribution platforms through the continuous management functions. In this design, we focus on three event driven functions: customization monitoring, security, and middlebox support. Algorithm 1 highlights the use of these functions.

Monitor function: This function allows for retrieving module use statistics across the network to aid in forensics analysis. When the state_req_window expires, a monitoring event (line 2) is triggered and a state report is requested from the device. Each state report consists of the last active_ts and any other network defined statistics recorded in the module. Within the module, the active_ts parameter is updated when a customization is invoked by Layer 4.5 during a socket send or receive call and does not merely track that the module is applied to an open socket. This timestamp is used by the NCO to determine if a module is considered active on the network, which allows the NCO to correlate active modules across the network to find any mismatches or irregularities. For instance, each active module can be cross checked to the device sending or receiving the customized traffic to determine if an unauthorized customization module is in use.
Algorithm 1 NCO: Continuous Management Logic

1: while True do
2:     Monitoring Event: /end of a state_req_window
3:         Perform device state request
4:         Update active_ts and other state info in CIB
5:     Security Event: /end of a sec_check_window
6:         Perform security check of module(s)
7:         if module(s) failed check:
8:             then Revoke failed module(s) on device
9:             Generate alert(s)
10:         Update CIB
11:     Middlebox Event: /flow query from a middlebox
12:     Perform CIB customization lookup
13:     if CIB lookup fails:
14:         then reject flow
15:         else Perform query processing
16:     Update CIB
17: end while

Security function: The security function of the NCO is used to provide a mechanism for adding per-network module security requirements to match a given threat model. At this stage of our design, we consider a threat model consisting of an attacker who can monitor all network communications, but assume the attacker has no means to directly compromise the NCO or customized devices. We acknowledge this model does not fit all private network requirements, but take this as an initial demonstration of how our NCO framework can enhance network security.

When a customization module is deployed, the per-module sec_check_window parameter is established and written to the CIB. At the end of each security window, a security event (line 5) is triggered and the NCO performs the desired security check with the deployed module. If the check fails, the default response is to immediately revoke the module and generate an alert. Otherwise, the CIB is updated to reflect the response from the module. We discuss a specific use case of this function further in Section III-C.

Middlebox function: We do not enforce Layer 4.5 customization capability on each middlebox. However, we do assume that each middlebox in the network can be expanded as necessary to establish a control channel with the NCO in an effort to minimize interference to customized flows. When a middlebox receives a customized packet it is unable to process locally, the middlebox requests processing assistance by sending a copy of the flow to the NCO for customization processing, triggering a middlebox event (line 11). The NCO first attempts to identify the customization in use by matching the values of the tap_socket stored in the CIB. Note that once a customization module is applied to an open socket, the unknown parameters of tap_socket have been set and are reported to the NCO via the periodic state reports. In the event a customization module determination fails, the flow is rejected and an unknown customization alert is triggered on the NCO. If the module is identified, then the NCO performs the required customization processing and a non-customized packet is returned to the middlebox. We discuss supporting local middlebox customization processing with pre-installed customization inverse modules in Section III-D.

B. Automation of Customization of Devices

Each device with Layer 4.5 capability supports automatic installation and removal of customization modules directed by the NCO. Figure 3 illustrates the device customization architecture to include the Device Customization Agent (DCA) and the customization modules for insertion into Layer 4.5.

![Layer 4.5 device architecture](image)

Fig. 3. Layer 4.5 device architecture. The DCA control channel with the NCO is used to receive and install customization modules (orange circles), which are invoked through the socket-transport tap.

1) Device Customization Agent: The DCA serves two main functions on the customized device. First, it is responsible for the management of all customizations installed on the device. Second, to support remote customization management, the DCA provides a set of handler functions and establishes an encrypted control channel with the NCO for invoking each function. The DCA handlers are used to install and revoke customization modules, relay commands to module embedded security and monitoring functions, and to report the state of all installed customizations.

When the DCA starts, it establishes a control channel with the NCO and sends an initial report containing device specific data required by the NCO for device identification and module construction. After initial check-in, the DCA awaits further commands from the NCO and invokes the appropriate handler in response. The report handler constructs a device-level report listing all registered and not previously reported revoked modules along with any device specific information that may be required by the NCO. The install handler accepts a customization module from the NCO, loads it onto the device, and registers it for Layer 4.5 customization. Conversely, the revoke handler will unregister a customization and delete it from the device so that it can no longer be used. The relay handler is used to extend the NCO/DCA control channel and enable the delivery of NCO commands to specific modules. This functionality is key to supporting security goals and is further discussed in Section III-C.

2) Customization Modules: Layer 4.5 customization modules are the basic building blocks for realizing protocol customization requirements. Each module includes standard functions to separate the processing of ingress and egress messages at the sender and receiver, respectively. Conceptually, each module will receive a buffer containing application data as...
a standard input, perform the desired customization, and then output the customized buffer for delivery to the transport layer or the application depending on direction of traffic.

Layer 4.5 customization modules are attached to a socket based on the modules tap_socket parameters. When Layer 4.5 identifies a new socket, a customization lookup process occurs. During this lookup process, the tap_socket application label is used to match the customization to the process owning the socket and assign values to any wildcard tap_socket parameters. The updated tap_socket parameters can then be reported to the NCO to aid the middlebox support function. Note that at this stage Layer 4.5 only allows for a single customization module to be applied to a matching socket. This design choice enables a more predictable customization behavior at the cost of necessary deconfliction and management of deployed customizations, which occurs on the NCO.

Table III presents the Layer 4.5 module API to conduct the required customization actions. The API can be split into two areas of responsibility. First, as seen in Figure 3, the cust_send() and cust_recv() functions are invoked to perform the necessary customization by the Layer 4.5 tap when a corresponding socket call is conducted. Therefore, development of the cust_send() and cust_recv() functions are the responsibility of the customization developer. The functions’ input and output arguments are standardized to provide the application data buffer or the transport layer data buffer as input for customization and require a buffer to be returned for processing by the transport or application layer. For reference, the customization module cust_send() and cust_recv() functions developed for the overhead evaluation presented in Section IV-B consist of approximately 100 combined lines of C code.

The state_report() and sec_respond() functions, however, are defined for each network and applied to all customization modules by the NCO during module construction. These API functions are invoked by the NCO using the DCA relay handler in support of NCO continuous management. First, the state_report() function is responsible for reporting module parameters required by the NCO’s monitoring function, such as the last active timestamp. Second, the sec_respond() function is used to perform the NCO directed security check using the embedded init_key.

The standard API, as shown in Table III, will support software reuse. We hypothesize that a relatively small number of design patterns will meet most of the operational needs for Layer 4.5 flow customization, and furthermore, different design patterns can leverage the same set of helper functions for application payload manipulation. The details of extending the Layer 4.5 customization module API are beyond the scope of this work.

C. Strengthening of Security

Customization module security functions are defined based on per-network requirements. For instance, a network could enforce that each customization module is digitally signed by the NCO and verification is performed during the module loading process [32]. Each module deployed in the network is embedded with a function for invoking the desired module security check, as seen in Figure 4 and Table III, and an init_key that can be adapted using ratcheting [33] techniques or key derivation functions [34] similar to what is done in our previous protocol dialecting work [3].

One particular security function would be to utilize a challenge-response authentication protocol [35] between the NCO and each module. To conduct this challenge, the NCO retrieves the current key for the module from the CIB to encrypt a randomly generated challenge message. Using the established encrypted control channel, the NCO sends the message to the module via the DCA relay function. When the sec_respond() function is called by the DCA, the module will decrypt the message, append a module specific response, and then encrypt the message prior to relaying back to the NCO. The NCO verifies the response and either revokes the module due to a failed response or updates the CIB accordingly. This security function use case is prototyped in Section IV-C.

D. Support for Middlebox Traversal

To provide on-device middlebox customization processing, the NCO can install device specific inverse customization modules as part of the customization module deployment process. A customization inverse module is responsible for transforming a customized message so that the middlebox can perform normal processing. This inverse customization logic differs from the cust_recv() module function by not requiring all logic necessary to interpret the customized portion of the message. As an example, consider a customization module that inserts a new field at the beginning of each application message header, which results in incorrect processing by a middlebox
performing deep packet inspection. An inverse customization module would only be responsible for removing this extra field prior to application header processing and may not necessarily incorporate the logic to correctly interpret the field.

Each network middlebox may require a different type of inverse customization module. If the middlebox is Layer 4.5 customizable and performs the required processing above Layer 4.5, such as an application proxy, then the inverse customization can be the normal customization module with the necessary `cust_recv()` function. Since we do not require each middlebox to be Layer 4.5 customizable, the NCO supports delivery of middlebox specific inverse customization modules that can be added to the middlebox rule set or plug into the middlebox processing pipeline. Section IV-D contains a demonstration of a middlebox inverse function used during deep packet inspection.

### IV. Prototyping and Performance Evaluation

In this section, our goal is to implement a prototype of the Layer 4.5 architecture and test the overhead of customization distribution and insertion into the TCP/IP stack. The experiments were performed under a testbed consisting of two Ubuntu 5.11 VMs running on an 8-Core Intel Core i9 MacBook Pro with 64GB of RAM. Each VM was allocated 2 CPUs, 8GB RAM, and a paravirtualized network adapter. The VMs were connected using an internal network configuration with a 1000Mbps capacity, no traffic loss, and no additional network traffic in order to fully test the overhead without interference of network congestion. The code and testing scripts are made available open-source on Github.

In this initial prototype, we did not choose to implement the Layer 4.5 customization modules and kernel layer DCA portions with eBPF due to the restrictions eBPF places on program capabilities and data access, which depend on where the program is being executed in the Linux kernel [36]. First, all eBPF programs must pass through the eBPF verifier during program compilation. The eBPF verifier is responsible for checking programs to ensure they will finish, which restricts potential capabilities of the Layer 4.5 customization modules. Second, the eBPF memory access that Layer 4.5 desires in order to modify application data is known as direct packet access. At the time of development, this access is only granted to eBPF programs operating at either the traffic controller or XDP layer. For instance, an eBPF program targeting the send path and executing at the traffic controller can modify the application data contained within the packet. This packet data is likely only a subset of the application data being sent, may require IP and transport header modification to match the data customization, and could result in unwanted behavior such as IP fragmentation if the customization applied additional data to the message. Therefore, the initial prototype of Layer 4.5 chose to utilize Linux kernel modules. We believe that it would be possible to utilize eBPF instead, but this may restrict customization module capabilities and require extending eBPF direct packet access functionality within the Linux kernel.

#### A. NCO Module Distribution Overhead

Within the NCO distribution functions, the deployment and revocation of customization modules require coordination with each host, while the construction of a customization module can be performed in advance. Additionally, revoking a module is a simpler NCO task than deploying a module since the corresponding DCA only needs the module ID (i.e., `mod_id`) to identify and revoke the module instead of the module binary which is required during deployment. Thus, in this section we focus on the overhead of deploying a previously built customization module using the control channel established by the NCO and a user-space DCA component. For reference, the NCO was written in 900 Python lines of code (LOC), the user-space DCA component was 350 Python LOC, and the NCO per-host deployment logic to distribute customization modules on the network is provided by Algorithm 2.

<table>
<thead>
<tr>
<th>Algorithm 2 NCO: Per-Host Deployment Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: while True do</td>
</tr>
<tr>
<td>2: Retrieve deployable modules from CIB</td>
</tr>
<tr>
<td>3: for Module in deploy list do</td>
</tr>
<tr>
<td>4: Deploy module binary to host DCA</td>
</tr>
<tr>
<td>5: if Success then</td>
</tr>
<tr>
<td>6: Transition module to CIB deployed table</td>
</tr>
<tr>
<td>7: else</td>
</tr>
<tr>
<td>8: Transition module to CIB error table</td>
</tr>
<tr>
<td>9: end if</td>
</tr>
<tr>
<td>10: end for</td>
</tr>
<tr>
<td>11: end while</td>
</tr>
</tbody>
</table>

The goal of this experiment is not the customization module itself, but the capability of the NCO to deliver a new customization module and update the CIB to reflect the deployment. To understand the limitations of the NCO, we vary the number of devices on the network for each test. Note that we simplify the experiment by emulating multiple devices using a new socket on the client to represent a new device.

![NCO Module Deployment Time](attachment:image.png)

Fig. 5. Measured latency of distributing a new module to 10, 50, 100, 175, and 250 devices, respectively. Green values show mean deployment time.

Figure 5 shows the deployment time results of 15 rounds of distributing a single 600KB customization module to each host.
device. To determine the distribution time, we added a new customization module to the CIB and then measured the time it took the NCO to distribute the module to each end host (lines 2-10). As expected, the deployment time increases as the number of devices on the network increases. Furthermore, the increase is approximately linear to the number of devices. Since the module being delivered is very small compared to the available bandwidth, the majority of the deployment time on the NCO is contributed by the CIB database queries to identify the module for deployment (line 2) and the updates necessary to reflect such deployment (lines 6 and 8). The DCA also contributes to the overhead seen in Figure 5 since it must accept and load the customization module prior to reporting success or failure to the NCO. This overhead is minimal since loading a kernel module in the Linux kernel is possible in a fraction of a second.

To address network scalability, we envision that for a large network with 1000s of devices, operators should utilize multiple NCO instances as supported by SDN platforms such as ONOS [37]. The adaptation of the NCO to an SDN application running on an ONOS controller is left as future work.

### B. Device Layer 4.5 Processing Overhead

The Layer 4.5 kernel-space DCA component was developed using a Linux kernel module, consisting of approximately 2,000 LOC. Figure 6 illustrates the flow path of the Layer 4.5 socket-transport tap and customization from Figure 3.

![Layer 4.5 socket-transport send and receive customization process.](image)

As seen in Figure 6, Layer 4.5 tapping will inevitably add some overhead to all flows in order to isolate the customized flows. When a socket is first detected, a new customization socket is created and a lookup occurs to match the socket to a registered customization. If a customization is found, then the `cust_send()` and `cust_recv()` functions are stored in the customization socket for future use. When a subsequent customization socket lookup is performed on an already processed socket, which is necessary for each send and receive call, we utilize the applicable stored customization function (if any) to process the socket call. To minimize memory overhead and prevent unnecessary copy operations for non-customized sockets, socket receive calls utilize the previously allocated application buffer to receive incoming messages. If the message is not customized, then it is returned to the application. Otherwise, the message is processed by the customization module and the updated message is stored in the application’s buffer.

By utilizing the application’s buffer to store incoming messages, we expect the non-customized flows to experience minimal overhead, while customized flows will be subjected to additional overhead dependent on the attached customization logic. For this reason, we developed a generic tagging customization to insert 32-byte tags into messages of a targeted application at set byte positions (e.g., every 1K bytes). The tag insertion involves a minimum of two expensive in-kernel memory copy operations, which are likely required by most customization modules. Note that this customization also increases the amount of data to be transferred.

1) TCP Application Overhead: The first flow we target is a bulk file transfer, represented by a 3GB Ubuntu image, using HTTP over TCP. When the Layer 4.5 customized server accepts an incoming connection, the customization lookup process identifies the socket corresponding to a registered customization module. The assigned customization module is designed to track the bytes sent from the server application, inserting a 32-byte tag every 1K bytes in a best-effort strategy to ensure the tag is present in each packet sent to the Layer 4.5 client. The corresponding client, using the `curl` application, is assigned a complementary reversal customization during the TCP connect phase and will remove the 32-byte tags prior to delivery to the application. Figure 7 illustrates the Linux baseline performance, the overhead of Layer 4.5 socket taps, and finally the overhead of Layer 4.5 taps with the customization applied. Each experiment was repeated 15 times and the file hash was verified on the client and server after each transfer completed. The module contained 50 LOC for the `cust_send()` function, 80 LOC for the `cust_recv()` function, and a total of 300 LOC.

From the boxplot, we see that the Layer 4.5 socket tap resulted in negligible overhead. However, when the aggressive tagging customization is applied to the socket, the 3GB of data is tagged every 1000 bytes, which results in approximately three million tag insert (server) and delete (client) events and an additional 96MB of transmitted data. Each of these tag events resulted in at least two in-kernel memory copy operations, but only a one second and modest 3% mean increase to file transfer time.

2) UDP Application Overhead: Next we applied the tagging customization to DNS requests made using the `dig` application to a local Layer 4.5 customized DNS server using the `dnsmasq` application. The DNS server was configured without a cache buffer to force a simplified internal lookup that
responds to all requests with the same IP address in an effort to eliminate the unpredictable overhead of internet based DNS queries with a remote server. For this use case we customized all dig generated DNS requests to the local DNS server and applied the 32-byte tag to the beginning of each request. We chose to insert the tag at the front of the request to force its removal before a legitimate request could be processed. The updated tagging module was reduced to 10 LOC for the cast_send() function, 15 LOC for the cast_recv() function, and a total of 150 LOC.

Figure 8 shows the performance overhead seen when making 1,000 different DNS requests to the server, repeated over 15 trials. The Layer 4.5 tapping mean overhead of approximately 0.35% was more significant than seen in the previous file transfer experiment because each DNS request was performed using a new socket and as a result experienced the full customization socket creation and lookup process from Figure 6. When each DNS message is tagged, we see a 1.6% mean increase over the baseline, which was less significant than the previous experiment due to the decreased number of tags (i.e., 1,000 vs. 3 million) and required memory copy operations. The customized batch DNS request/response time increased by less than 0.1 seconds, which is likely to be unnoticeable by the end host applications or users.

C. Prototyping of Challenge-Response Security Check

We implemented the customization security functionality from Figure 4 to enable the NCO/Module challenge-response protocol (80 LOC) described in Section III-C under the threat assumption that each device is secure and the NCO/DCA control channel is protected with TLS. Algorithm 3 provides the NCO challenge-response security logic for challenging a deployed customization module.

**Algorithm 3 NCO: Challenge-Response Security Logic**

1. if Module security window expired then
2. Retrieve module specific init_key
3. Generate and encrypt random challenge
4. Send challenge to DCA relay handler
5. Receive and validate response
6. if Success then
7. Update security window
8. else
9. Revoke customization module
10. Update CIB with failure status
11. end if
12. end if

When the customization module sec_check_window expires, the NCO issues a security challenge to the module. Our prototype implementation performs this action by first randomly generating a 256-bit module specific init_key, writing it into the module during the construction phase, and then storing it in the CIB. When the NCO challenges a module, this key is retrieved from the CIB (line 2) and used to encrypt a randomly generated 8-byte challenge using AES encryption (line 3). This challenge and corresponding nonce is transmitted to the DCA, which in turn uses the relay handler to call the modules sec_respond() function with the NCO’s challenge as an argument (line 4). The customization module first decrypts the challenge and then appends the embedded server generated mod_id to the end of the message. The message is again encrypted by the module with a new nonce and relayed back to the NCO for validation (lines 5-11). The NCO decrypts the message and if a failure is detected, the DCA revoke function is called to remove the module. To validate the implementation, we configured the NCO with a five second sec_check_window and observed 20 rounds of challenges, logging each challenge on the NCO and device to ensure proper encryption/decryption was performed.

D. Demonstration of NCO-Assisted Middlebox Traversal

Consider a scenario in which a Layer 4.5 capable host is exhibiting unusual network behavior, particularly through DNS queries. To investigate if non-standard applications are conducting DNS queries, the NCO operator deploys a new customization module that applies an application ID tag\(^3\) to each DNS query conducted using the dig application, which is not normal client behavior. This tag is inserted to the front of the corresponding DNS query, which is known to result in proper logging and subsequent analysis.

\(^3\)This tag can be encrypted in a way similar to the challenge-response process, or utilizing other mechanisms to mitigate forgeries.
in processing errors at the network deep packet inspection middlebox. To address this issue, the NCO operator also deploys a customization inverse module to the middlebox to identify the presence of the application ID tag during packet inspection and generate an alert.

To visualize the use of this pre-installed customization inverse module on a middlebox conducting deep packet inspection, we developed a Wireshark dissector (40 LOC) that interprets the application ID tag and displays it appropriately. When the NCO distributes the new DNS customization module to the target host, the corresponding dissector is also distributed to the middlebox. To accomplish the installation of the dissector, we expanded the user-space component of the DCA to accept a command to install the inverse module (i.e., dissector script) in the plugin directory of Wireshark.

![Wireshark capture with Layer 4.5 inverse customization module](image)

Fig. 9. Wireshark capture with Layer 4.5 inverse customization module (i.e., dissector) applied to identify the embedded application ID prior to DNS packet contents.

Figure 9 shows the identification of a DNS request using `dig` among multiple "standard" DNS requests. When `dig` was used, the inverse module identified the customization, filled in the Application ID column to alert the operator, and then allowed DNS processing for the remainder of the packet.

V. ENCRYPTED FLOW CUSTOMIZATION

The Layer 4.5 evaluation of Section IV focused on the customization of unencrypted application protocols, but many protocols use encryption to not only add security to the protocol but also to avoid middlebox interference. If Layer 4.5 is used to customize application layer encrypted traffic, then we must keep in mind that all customizations are applied while the data is in an encrypted state. However, protocol customization does not always need to have access to the plain text application data to be useful. In this section, we first highlight a receiver-side processing complication when Layer 4.5 customizes encrypted traffic. We then generalize the Layer 4.5 architecture to allow customization of all flows and evaluate the overhead changes.

A. Customization Complications

Consider a scenario where a network operator deploys a customization to insert an application tag to all internal network traffic in an effort to create labeled data for the network traffic classification machine learning model. After deploying the customization, the operator receives multiple complaints that some applications are no longer working. The operator determines that some applications, in particular encrypted applications, are unable to process incoming customized traffic and terminate the connection after the error occurs. To understand why this error is occurring on encrypted flows, consider the example TLS flow of Figure 10 illustrating the use of a web server customization module tasked with appending a 32-byte tag to every 1000 bytes of transmitted data. This customization tag is applied to the server’s TLS payload and results in a subsequent client decryption error.

![Example of client TLS processing failure when customized data is present within the TLS payload](image)

In the example TLS flow, the customized server sends a 1500-byte application payload to the TLS client (1). Since the TLS application socket is being customized, Layer 4.5 intercepts the message and passes it to the attached customization module, which inserts a 32-byte tag at byte position 1000 because the payload is over 1000 bytes. Layer 4.5 then delivers the customized message to the transport layer for transmission to the client (2). When the customized client receives the 1532 bytes, the `curl` application first requests 5 bytes to determine the associated TLS encrypted payload length (3). Layer 4.5 will intercept the 5-byte message from the transport layer (4) and pass to the client customization module. Since the customization tag is not present, the client customization module updates the position tracker and passes the bytes to the application (5). The `curl` application then requests the remaining 1495 encrypted bytes from the transport layer (6), which leaves 32 bytes of encrypted data in the transport buffer. Before the requested bytes are returned to the `curl` application, the customization module removes the customized bytes, which results in 1463 bytes being delivered to the application (8). At this point the application did not receive all the expected bytes, but instead of requesting the additional missing bytes `curl` attempts to decrypt the 1463 bytes and a decryption error is triggered, resulting in the TLS connection being terminated. This same behavior was experienced in a parallel effort to evaluate network detection of data exfiltration that leveraged Layer 4.5 as the insertion point [38].

B. Generalized Layer 4.5 Design

First, note that the errors experienced when processing customized encrypted flows were only caused by receive
side processing. Additionally, the processing errors occurred because the customization module did receive enough data from the transport layer to fully process the customization prior to delivery to the application. Therefore, we will focus on generalizing the Layer 4.5 DCA/customization module interface design by adding a customization module buffering capability to enable processing both encrypted and unencrypted customized flows. Figure 11 illustrates the new customization receive path flow to illustrate the changes made to support a more generalized design.

![Layer 4.5 generalized socket-transport receive customization process](image)

As seen in Figure 11, instead of performing the layer 4 receive call and filling the application buffer with the potentially customized data, we first perform a PEEK operation to determine when data is present at layer 4 without transferring the data into the application buffer. After we know data is available, we determine if the socket is being customized. If the socket is not customized, then we simply perform the layer 4 receive call and fill the application buffer with the requested data. If the socket is being customized, however, Layer 4.5 will now utilize an intermediary buffer to hold the potentially customized data to allow processing by the customization modules before delivery to the applications. The size of the intermediary buffer will differ for each customization module and will now be specified for the DCA to allocate when attaching the module to the customized application socket. Since the customization buffer may be larger than the application’s buffer, the customization module is required to buffer data as necessary until the application is ready to process it. Additionally, the customization module buffer may hold processed data (possibly all remaining data) for future application receive calls. Therefore, we also perform an early customization check prior to the transport PEEK operation to allow the transfer of data from the customization buffer instead of from the transport layer.

The use of a separate customization buffer results in more complicated customization modules and DCA receive message processing. First, the DCA must now determine if a customization module is attached to the socket and if the customization module has buffered data ready for the application prior to making the application requested receive message call to layer 4. If the socket does not have an attached module, then this check will fail and may result in an additional customization lookup after receiving the message from the transport layer. Second, the customization modules must now perform buffer management to ensure proper customization processing and delivering of the correct amount of data to the application to avoid processing errors.

C. Device Layer 4.5 Processing Overhead

The generalized Layer 4.5 design introduces processing overhead in the form of additional memory operations and customization logic complexity. Therefore, we re-evaluate the performance of Layer 4.5 customization from Section IV-B. We first evaluate the ability to customize an encrypted bulk file transfer and the overhead of the customization. We then evaluate the overhead change for the DNS batch processing customization.

1) TCP Encrypted Application Overhead: We applied the tagging customization from Section IV-B to bulk file transfers using HTTPS over TCP. The main goals of this experiment are to determine if the new buffering capability enables transparently customizing encrypted flows and if there is a significant processing overhead impact of protocol customization to encrypted traffic flows.

First, to determine the customization receive buffer size that should be assigned to the customization module, we reviewed the curl client behavior from the initial experiments of Section IV-B. The client side curl application was observed to conduct a receive message request with maximum of 102400 bytes. Thus, to allow processing more data than previously allowed, the customization module was configured with a customization receive buffer size of 102400 + 3200 bytes. We chose to add 3200 bytes above the maximum buffer since there would be at least 100 32-byte customization tags present if the requested buffer was completely filled. Therefore, we could safely transfer the 3200 additional bytes from layer 4, remove all customization bytes, and then return all remaining bytes to the application without having to buffer data for a future receive call. Figure 12 illustrates the resulting overhead evaluation.

From the boxplot, we see that the overhead experienced after adding the buffering capability did not differ from the initial unencrypted TCP customization results. Note, there was no significant increase in the amount of data processed during each receive message call since the curl application provided a large receive buffer and providing a larger buffer did not
result in processing significantly more data each call. The key takeaway from this experiment is the ability to customize encrypted flows without resulting in unacceptable overhead or unexpected application termination.

2) UDP Application Overhead: We repeat the DNS batch experiment using an updated customization module with a client side `dig` application and server side `dnsmasq` application. Note that the customization module was updated to include buffering logic, but did not actively buffer data. Again, to ensure we could measure the overhead experienced, we decided to conduct batch DNS requests consisting of 1000 different requests to the server, repeated over 15 trials. Figure 13 illustrates the updated overhead evaluation.

From the resulting boxplots, we observed a more significant Layer 4.5 tapping mean overhead that can be attributed to multiple customization lookup calls performed by the `dig` client. Recall from Figure 11 that we first determine if a customization module is attached and buffering data for the application prior to peeking into the transport buffer and continuing to perform the requested receive message call. Since we are using UDP sockets in this experiment, each client DNS request is a new socket and there will not be any customization modules attached until the application data is received, which results in a lookup failure. Performing the customization lookup request twice in the receive message path increased the Layer 4.5 tap overhead, but not significantly.

When each DNS message is tagged, we see a less significant mean increase compared to Figure 8. This minimal increase to the overhead indicates that allocating the additional receive buffer did not result in unacceptable delays. It should be noted that under this customization experiment, `dnsmasq` maintained the same socket for all receive delays and, thus, only allocated the new receive buffer a single time. If we instead customized the `dig` client receive path, this would have resulted in one thousand receive buffer allocation and free operations, which would likely increase the batch customization overhead.

VI. AGILE NETWORK CUSTOMIZATION

Network administrators that implement protocol customization may desire the capability to change the customization in use on the network often. However, rotating from one protocol customization to the next could easily result in customization synchronization challenges and disruptions to network services being customized. To accommodate agile network customization, we further extend the Layer 4.5 architecture to support customization rotation and synchronization without disrupting active network communications [39]. We begin this section by describing the new Layer 4.5 features and follow up with multiple agile network customization use cases.

A. Customization Rotation and Synchronization Design

Recall from Section III-B that Layer 4.5 will only allow a single customization module to be attached to a socket in order to prevent customization management complications. To support customization rotation on an active socket, we relax this condition to allow multiple modules to attach to the socket, but still enforce that only one customization module can actively customize the flow. To accomplish this, the Layer 4.5 DCA will first attach all matching customization modules to a matching socket and divert application flows to attached modules in the order in which they are attached. When a customization module receives application data it must now decide if the flow should be customized or passed on to the next customization module. After a module customizes a flow, the DCA will no longer divert the flow to other modules for further customization.

To allow the NCO to determine the order of attached customizations, we now include a priority assignment for each module. Therefore, when the DCA matches customization modules to an active socket, the modules will be attached according to their priority assignments. Furthermore, the NCO can update the priority of a deployed module, which instructs the DCA to reorder the module if it is currently attached to an active socket. These two new features can now be used to rotate from one customization module to the next on an active socket without causing application flow processing errors. Algorithm 4 provides the logic for the NCO to conduct the customization rotation. Note that for the purposes of rotation, the host sending customized data is considered to be the client and the host processing the customized data is the server.
Network traffic classification is a focused subset of the more general network traffic analysis that aims to classify traffic based on the application or type of application generating the traffic [40]. To automate network traffic classification, machine learning techniques are growing in popularity [40]. Supervised machine learning models require labeled data to help train and validate the models, but creating this labeled data can be challenging because each network may have unique traffic patterns due to the personnel generating traffic and possibly using different applications.

Furthermore, utilizing the expanded Layer 4.5 buffering capability, customization modules have the option to provide early ability, customization modules have the option to provide early access controls for the flow at the socket layer. Each application implementing flow level access controls at a host include iptables, firewalls, or network interface specific access control lists. These methods lack agility because they mostly enforce static rules that must be installed ahead of time.

In contrast, the NCO can monitor a flow and dynamically install access controls for the flow at the socket layer. Each application running on the host is tied to a specific user account, which may be the root user. In order to resolve a local user ID to a network ID, the NCO can be expanded to query the local identity manager (e.g., an LDAP server). Using this information, the NCO can embed specific user ID’s into customization modules that restrict specific applications. Furthermore, utilizing the expanded Layer 4.5 buffering capability, customization modules have the option to provide early responses to the application without sending any of its data down to the transport layer. To accomplish this, we modify the normal customization send path shown in Figure 6.

For simplicity, consider a customization module that is deployed to prevent the root user account from generating web traffic using the Firefox application. When the cust_send() function receives the Firefox web traffic request as part of the customization process, instead of customizing the request and passing it on to the transport layer, the cust_send() function removes the data and fills the customization receive buffer with a pre-generated network response. After receiving confirmation...
that the send call completed, the application performs a receive message call, which is also being customized by the same module. As shown in Figure 11, the early customization buffer receive call will succeed and the response will be provided to the application.

There are several benefits to performing flow access controls at the socket layer. First, the access controls can be applied to multiple user/application combinations and removed/updated by the NCO as the flow behavior or network policy changes. Second, the unauthorized traffic never reaches the transport layer, which reduces network and server load. Last, the application can be provided a customized response to indicate why the traffic was subjected to access controls.

4) Moving Target Network Defense: Moving Target Network Defense (MTND) aims to increase an attacker’s workload by making the traffic behavior of the entire network unpredictable [42]. For instance, network surveillance may be the first step performed by an attacker prior to launching an attack. Part of the surveillance could be the collection of network traffic to learn what applications are being used and how the network is configured. We believe that Layer 4.5 customization rotation is one method that can be used to change traffic behavior of selected hosts frequently and result in degraded network surveillance capabilities as automated tools for analyzing traffic may not be able to properly parse the customized traffic. Additionally, Layer 4.5 customization modules could be classified by intent and deployed by the NCO to conduct Intent Based Networking (IBN). For instance, a network operator could increase the threat level of the network in response to a potential attack. This intent could be matched to a subset of pre-built customization modules, which are then automatically deployed on the network to modify the network behavior.

We believe Layer 4.5 has the potential to assist network operators in achieving a moving target network through the use of agile network customization. However, future research remains to determine optimal customization rotation rates, schema, and effectiveness.

VII. DISCUSSION

In this section we discuss some limitations pertaining to Layer 4.5 device customization and possible extensions to the orchestration and continuous management framework.

A. Limitations of Layer 4.5 Customization of Devices

The first limitation we discuss is that all customization actions are event driven by a socket send or receive call, which means customizations will not be triggered by actions at transport layer (i.e., layer 4) and below, such as receipt of TCP acknowledgements. If these triggers are necessary, then a lower layer solution should be used, perhaps in conjunction with a Layer 4.5 customization module. This limitation also means Layer 4.5 customizations need to be designed to fit the unique message processing logic of tapped applications. For instance, one application could send one IP packet length of data at a time to the socket, while a different application sends a 65KB buffer to the socket and relies on the lower layers to segment the buffer into chunks that will fit into IP packets.

Next, unexpected application behavior influences the customization module development complexity. During our prototyping and testing, we experienced the receiving end of some applications perform multiple requests to retrieve a single application message by initially requesting the first few bytes of the incoming message prior to requesting the remaining message body. For instance, when dnsmasq uses TCP for a DNS request, the application first requests one byte of data from layer 4 to determine the byte length of the accompanying DNS request. This type of behavior will require the use of customization modules with the new buffering capability from Section V to allow retrieving and processing the entire customized request before delivery to the application.

With regards to reliable customization rotation, the customization module needs a method to identify whether it should be applied to the flow or to allow another customization attached to the same socket to be applied. This can be accomplished by using a customization signature, such as a series of bits at a known location or based on the size of the data. While it is still possible to rotate customization modules without this signature detection, doing so may cause unwanted application behavior such as connection termination.

Finally, application layer encryption will limit the types of protocol customizations to some degree. For instance, when application data is encrypted prior to the socket receiving the buffer, then any module aiming to modify the application data will not have proper access. As shown in Section V, a module can insert data into the encrypted messages as long as the customization is removed prior to decryption.

B. Extensions to Continuous Management Functions

Expanding Middlebox Support: The NCO has the potential to aid middleboxes in the understanding of incoming traffic and the possible detection of malicious traffic. There are previous efforts to classify network traffic using machine learning techniques [43] or by tagging traffic flows in the network [44]. Layer 4.5 customization modules have the ability, as demonstrated in Section IV-D, to add application specific information to messages without application knowledge, which can be used to supplement machine learning and flow identification techniques. This additional information could be interpreted by middleboxes, such as intrusion detection/prevention systems, to potentially identify malicious behavior, such as control channel establishment with outside devices.

Another potential method to support middlebox devices would be to tap raw sockets in addition to the TCP/UDP sockets currently tapped by Layer 4.5. If the middlebox is using a raw socket to process traffic, then we could utilize a Layer 4.5 inverse module on the socket to allow the middlebox to evaluate the uncustomized message. Note that inverse customization modules on raw sockets may need to store the customized portion of the incoming messages in order to re-customize the message after middlebox processing. There may also be some security concerns with this method since the inverse module could theoretically be used to alter a message that would normally be flagged by the middlebox.
Supporting Lower Layer Customization: One approach to extend this work to support customization at or below the transport layer would be to incorporate the Layer 4.5 module API into eBPF programs performing lower layer customizations. This would allow the NCO to manage these customizations, enhance their security, and coordinate their traversal of middleboxes.

In addition, one could develop Layer 4.5 customization modules to monitor and inform performance of lower layer customization. For example, such modules could track application throughput for socket connections that are known to have lower layer customizations applied. If negative performance impacts are detected, the Layer 4.5 customization could disable the lower layer customization by setting certain socket options or alerting the DCA to take action.

Raising NCO Abstraction: We believe the NCO not only can run as a control application on an SDN controller, but also could serve as a baseline itself for other control applications by exposing a high level standard API to developers to enhance the monitoring, security, and middlebox traversal capabilities. This flexibility is important because enterprise and data center networks tend to have unique, network specific security and performance requirements.

Beyond increasing NCO’s programmability, it is also worth conducting a security analysis of the NCO since it may introduce new security threats/challenges beyond what are faced by an SDN controller. The threat model used for this paper should be expanded to motivate additional security functionality both within the NCO and in the modules it deploys.

VIII. Conclusion

Agile network operation requirements have emerged for enterprise and data center networks. To meet such requirements, we designed and evaluated a Layer 4.5 customization framework to perform fine-grained, process-level flow customization, with an emphasis on network-wide orchestration and the continuous management of each customization. The results are promising and demonstrate potential benefits of a software defined approach to rapidly specializing enterprise and data-center network traffic for security and performance purposes.

REFERENCES


