Towards Software Defined Layer 4.5 Customization

1st Daniel Lukaszewski  
Naval Postgraduate School  
dflukasz@nps.edu

2nd Geoffrey G. Xie  
Naval Postgraduate School  
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. The 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.

In this paper, we propose a systematic framework of network-wide orchestration and continuous management of protocol customization 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 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 and show that the framework incurs modest processing overhead, at the levels of 3% and 1% for sample customized flows and non-customized flows, respectively.

Index Terms—protocol customization, software defined networks

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 and associated backend data-centers should be able to agree upon and optimize the performance of a common application specific protocol on the fly [4], [5].

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 time scale 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 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 per customization, lacks the agility to support enterprise networks and data-centers, which are large in size and must uphold stringent security and performance requirements at all time [6]. 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 [8]–[10].

A straightforward method of using an SDN controller to support protocol customization is to virtualize all devices in 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 down time 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 [V] 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 [11]–[13] 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 [14]–[16] suggests a need to differentiate application processes when performing customization. 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 [I] — our customization taps constitute a shim layer between the application and transport layers, which we call “Layer 4.5”.

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 send and receive socket functions.

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

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. Finally, Sections V and VI wrap up the paper with a discussion of the limitations of Layer 4.5 customization of devices and future extensions to the framework.

II. RELATED WORKS

We review related work on three fronts. First, we discuss two efforts that investigated orchestration of protocol customizations. Then, we describe related device stack customization methods and efforts that aim to support application transparent customization. Finally, we consider current methods of customization use to address middlebox interference.

Orchestration of Customizations: The Walmart L3AF project [17] and the protocol plugin work [18] 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.

Application Transparent Customization: The Virtual Transport Layer (VTL) [19] and specialized application filtering work of [20] utilize eBPF to perform application customization without modifications to the application. VTL dynamically maps the application’s TCP socket to that of a different protocol to improve network performance, while the authors of [20] specialize socket filters based on the applications owning the socket. Both works provide customization of applications and are complementary to our efforts. The initial prototype of Layer 4.5 did not use eBPF due to concerns about extending eBPF and the Linux kernel to fulfill design requirements, but we believe it is possible to do so.

Layer 4.5 is similar to the service mesh layer present between microservice applications and the transport layer [21]. The service mesh layer utilizes application companion proxies as a vantage point for network customization. Layer 4.5 differs from this work by supporting all current applications, not just those designed to use services provided by application companion proxies.

Middlebox Support: It is well known that the current protocol extension methodology suffers from middlebox interference [8]–[10]. To specifically address interference from middleboxes conducting deep packet inspection, some protocols leverage application encryption [22], [23]. Layer 4.5 customization could also leverage encrypted application traffic to bypass middlebox support, but this is not applicable to all customizations possible with Layer 4.5.

In SDN environments, middleboxes can be virtualized and supported with OpenBox [24], 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.

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. We then expand on how Layer 4.5 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 with TLS and experimenter type messages to provide new functionality.

![Layer 4.5 NCO](image)

Fig. 2: Layer 4.5 NCO consisting of “distribution” and “continuous management” functions, a Customization Information Base (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 Table I parameters and storing all values in the CIB. Each customization module is linked to a device or set of devices via the mod_id parameter. Customized devices use the 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 their place. 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. For instance, not all applications will perform a socket bind call, setting the source IP and port, prior to establishing a connection or sending traffic. 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. These established windows are used by the continuous management functions and trigger the events shown in Algorithm 1.

**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 a confirmation that the module has been unregistered from Layer 4.5 and removed from the host.

![Table I: Parameters Embedded Per-Module](image)

<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 on client)</td>
</tr>
</tbody>
</table>

![Table II: Monitoring and Security Intervals Per-Module](image)

<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>
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.

Algorithm 1 NCO: Continuous Management Logic

```
while True do
    Monitoring Event: //end of a state_req_window
    Perform device state request
    Update active_ts and other state info in CIB

    Security Event: //end of a sec_check_window
    Perform security check of module(s)
    if module(s) failed check:
        then Revoke failed module(s) on device
        Generate alert(s)
    end \if
    Update CIB

Middlebox Event: //flow query from a middlebox
    Perform CIB customization lookup
    if CIB lookup fails:
        then reject flow
        Generate alert(s)
    else Perform query processing
    Update CIB
end while
```

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.

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.

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. 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 deliver NCO commands to specific modules and is further discussed in the next section.

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 L4 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 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>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust_send()</td>
<td>modify outbound message</td>
</tr>
<tr>
<td>cust_recv()</td>
<td>modify inbound message</td>
</tr>
<tr>
<td>state_report()</td>
<td>report monitoring statistics</td>
</tr>
<tr>
<td>sec_respond()</td>
<td>reply to security check</td>
</tr>
</tbody>
</table>

Table III presents the Layer 4.5 module API to conduct the required customization actions. Development of the cust_send and cust_recv functions are the responsibility of the customization developer, while the state_report and sec_respond functions are defined for each network and applied to all customization modules within the network. 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. The state_report function is responsible for reporting module parameters required by the NCO’s monitoring function, such as the last active timestamp. Lastly, the sec_respond function is used to perform the NCO directed security check using the embedded init_key.

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 [25]. 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 [26] techniques or key derivation functions [27] similar to what is done in our previous protocol dialecting work [3].

![Fig. 4: Customization module with embedded security functionality](image)

One particular security function would be to utilize a challenge-response authentication protocol [28] 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...
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 in the network 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 Layer 4.5 and test the overhead of customization distribution and insertion into the TCP/IP stack. The current prototype simplifies the process of attaching a customization to apply only to new sockets, which may require application restart after a matching customization module is registered. (New functionality capable of “hot-swapping” of modules on an active socket is deferred to future work.) The code and testing scripts are made available open-source on Github.

The experiments in this section 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.

A. NCO Module Distribution Overhead

We begin by evaluating the network deployment of a new customization module using the control channel established by the Layer 4.5 NCO and a user-space DCA component. The NCO was written in 900 python lines of code (LOC), and the user-space DCA component was 350 python LOC. 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

1https://github.com/danluke2/software_defined_customization

2All reported line of code values are approximated

the limitations of the NCO, we vary the number of devices on the network for each test.

![NCO Module Deployment Time](image)

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 device. As expected, the deployment time necessary 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 bandwidth available, the majority of deployment time is contributed by the CIB database queries to identify the module for deployment and the updates necessary to reflect such deployment.

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. 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 (LKM), consisting of approximately 2,000 LOC. LKMs were chosen over the newer eBPF capability since it was unclear if eBPF could support all design elements of Layer 4.5 without expanding eBPF capabilities, which would result in kernel modification. An eBPF version of Layer 4.5 should be explored to determine any limitations and performance differences from the LKM approach, but this is left as future work.

Layer 4.5 tapping will inevitably add some overhead to all flows in order to isolate the customized flows. To minimize impact to non-customized flows, the DCA stores customized and non-customized sockets in separate hash tables. When a socket is first detected, a new customization socket is created and a customization lookup occurs to match the socket to a registered customization. If a customization is found, then the customization socket is stored in a customized hash table; else, it is stored in the non-customized hash table. When a subsequent customization socket lookup is performed on an
already processed socket, which is necessary for each send and receive call, we prioritize non-customized flow lookup and, as a result, we expect the non-customized flows to experience minimal overhead. Customized flows will be subjected to the same lookup process and 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 increases the amount of data to be transferred. The module contained 50 LOC for the cust_send function, 80 LOC for the cust_recv function, and a total of 300 LOC.

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 6 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.

Fig. 6: Measured overhead increase of Layer 4.5 socket taps and Layer 4.5 taps with sample customization applied. Green values show mean transfer time and red values percentage of increase above baseline.

From the boxplot, we see that the Layer 4.5 socket tap resulted in approximately 0.38% mean overhead. This overhead primarily comes from the customization lookup process applied during each TCP send and receive call by the client and server. When the aggressive tagging customization is applied to the socket, the 3GB of data is tagged every 1K bytes, which results in approximately 3 million tag insert (server) and 3 million tag delete (client) events and an additional 96MB of data. Each of these tag events resulted in at least two in-kernel memory copy operations, but only a modest 3% mean increase to file transfer time.

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 cust_send function, 15 LOC for the cust_recv function, and a total of 150 LOC.

Fig. 7: Measured overhead increase of Layer 4.5 socket taps and Layer 4.5 taps with sample customization applied. Green values show mean batch completion time and red values percentage of increase above baseline.

Figure 7 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 1.2% 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. 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., 1000 vs. 3 million) and required memory copy operations.

C. Prototyping of Challenge-Response Security Check

We implemented the 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. When the DCA first reports a customization module is registered, 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 and used to encrypt a randomly generated 8-byte challenge using AES encryption. This challenge and corresponding nonce is transmitted to the DCA, which in turn calls the modules sec_respond function with the NCO’s challenge as an argument. The 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. 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 security 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 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 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) applied to identify application ID in DNS packet contents.

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., L4) 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 that 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 65K 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 have experienced that 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 L4 to determine the byte length of the accompanying DNS request. This type of behavior will inhibit the same DNS tagging customization used in the overhead testing or middlebox demonstration since the tag can no longer be inserted at the beginning of the message, before the length field. We could mitigate this behavior by changing the DNS tagging to insert the tag to the end of the request, while also updating the field length to reflect the new message size.

Finally, application layer encryption will limit the types of protocol customizations to some degree. For instance, when data is encrypted prior to the socket receiving the buffer, then any module aiming to modify application data will not have proper access. A module could still insert data into the messages for removal by a middlebox or at the end device prior to decryption, but this assumes that decryption will be performed on the receiving device in user-space after the cust_recv function removes any extra data.

\[\text{Fig. 8: Wireshark capture with Layer 4.5 inverse customization module (i.e., dissector) applied to identify application ID in DNS packet contents.}\]
B. Extensions to Continuous Management Functions

Expanding Middlebox Support: 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 [28] or by tagging traffic flows in the network [29]. Layer 4.5 customization modules have the ability, as demonstrated in Section [V-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.

Supporting Lower Layer Customization: One approach to extend this work to support customization at transport layer or lower would be to incorporate Layer 4.5 module API into eBPF programs performing lower layer customizations. This would allow the NCO to track 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 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 is 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.

VI. CONCLUSION

In this research, 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 specializing enterprise network traffic for security and performance purposes.

ACKNOWLEDGEMENTS

We thank anonymous reviewers for their helpful comments and Gianni Antichi for shepherding the final revisions. This work was partially funded by the U.S. Office of Naval Research (ONR) under the TPCP program. The contents are those of the author(s) and do not necessarily represent the official views of, nor an endorsement, by the U.S. Government.

REFERENCES