NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

SOFTWARE DEFINED CUSTOMIZATION OF NETWORK PROTOCOLS WITH LAYER 4.5

by

Daniel F. Lukaszewski

September 2022

Dissertation Supervisor: Geoffrey G. Xie

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC, 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2022	**3. REPORT TYPE AND DATES COVERED** Dissertation

4. TITLE AND SUBTITLE SOFTWARE DEFINED CUSTOMIZATION OF NETWORK PROTOCOLS WITH LAYER 4.5	**5. FUNDING NUMBERS** RCP62
6. AUTHOR(S) Daniel F. Lukaszewski	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research, Arlington, VA 22203	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

The rise of software defined networks, programmable data planes, and host level kernel programmability gives rise to highly specialized enterprise networks. One form of network specialization is protocol customization, which traditionally extends existing protocols with additional features, primarily for security and performance reasons. However, the current methodologies to deploy protocol customizations lack the agility to support rapidly changing customization needs. This dissertation designs and evaluates the first software-defined customization architecture capable of distributing and continuously managing protocol customizations within enterprise or datacenter networks. Our unifying architecture is capable of performing per-process customizations, embedding per-network security controls, and aiding the traversal of customized application flows through otherwise problematic middlebox devices. Through the design and evaluation of the customization architecture, we further our understanding of, and provide robust support for, application transparent protocol customizations. We conclude with the first ever demonstration of active application flow "hot-swapping" of protocol customizations, a capability not currently supported in operational networks.

14. SUBJECT TERMS software defined networks, protocol customization, agile networks	**15. NUMBER OF PAGES** 131
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

SOFTWARE DEFINED CUSTOMIZATION OF NETWORK PROTOCOLS WITH LAYER 4.5

Daniel F. Lukaszewski
Lieutenant Commander, United States Navy
BS, University of Arizona, 2010
MS, Computer Science, Naval Postgraduate School, 2017

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2022

Approved by:

Geoffrey G. Xie
Department of
Computer Science
Dissertation Supervisor
Dissertation Chair

Joshua A. Kroll
Department of
Computer Science

Pantelimon Stanica
Department of
Applied Mathematics

Mathias N. Kolsch
Department of
Computer Science

Justin P. Rohrer
Department of
Computer Science

Karl Wiegand
USN (Reserves)

Approved by:

Gurminder Singh
Chair, Department of Computer Science

Joseph P. Hooper
Vice Provost of Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The rise of software defined networks, programmable data planes, and host level kernel programmability gives rise to highly specialized enterprise networks. One form of network specialization is protocol customization, which traditionally extends existing protocols with additional features, primarily for security and performance reasons. However, the current methodologies to deploy protocol customizations lack the agility to support rapidly changing customization needs. This dissertation designs and evaluates the first software-defined customization architecture capable of distributing and continuously managing protocol customizations within enterprise or datacenter networks. Our unifying architecture is capable of performing per-process customizations, embedding per-network security controls, and aiding the traversal of customized application flows through otherwise problematic middlebox devices. Through the design and evaluation of the customization architecture, we further our understanding of, and provide robust support for, application transparent protocol customizations. We conclude with the first ever demonstration of active application flow "hot-swapping" of protocol customizations, a capability not currently supported in operational networks.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Introduction **1**
1.1 Thesis 5

2 Design of the Layer 4.5 Customization Architecture **7**
2.1 Network-Wide Orchestration 8
2.2 Automation of Customization of Devices 12
2.3 Strengthening of Security 19
2.4 Support for Middlebox Traversal 20
2.5 Limitations 20
2.6 Summary 22

3 Prototyping and Evaluation **23**
3.1 Distribution Overhead 24
3.2 Processing Overhead 26
3.3 Embedding Security Requirements 33
3.4 Assisting Middlebox Traversal 34
3.5 Insights 36
3.6 Summary 37

4 Enabling Customization of Encrypted Flows **39**
4.1 Motivation 39
4.2 Design of Module Message Buffering 42
4.3 Evaluation 46
4.4 Insights 51
4.5 Summary 51

5 Rotating Customizations in Wide Area Networks **53**
5.1 Motivation 53
5.2 Design of Module Hot-Swapping 55

5.3 Evaluation 61
5.4 Insights 77
5.5 Summary 78

6 Summary of Contributions **79**
6.1 Reproducibility 80
6.2 Future Work 80

Appendix: Background and Related Works **85**
A.1 Network Protocol Customizations 85
A.2 Related Work 93
A.3 Our Previous Work 99

List of References **101**

Initial Distribution List **109**

List of Figures

Figure 2.1 Proposed architecture for centralized control of protocol customization in a network. 7

Figure 2.2 Layer 4.5 Network-Wide Customization Orchestrator. 8

Figure 2.3 Layer 4.5 device architecture. 13

Figure 2.4 General Layer 4.5 tap and customization logic for application send and receive message processing. 18

Figure 2.5 Customization module with embedded security functionality. . . 19

Figure 3.1 Measured latency of distributing a new customization module. . . 26

Figure 3.2 Layer 4.5 application flow tap and customization logic. 27

Figure 3.3 Measured overhead increase of Layer 4.5 socket taps and Layer 4.5 taps with sample customization applied to 1000 short-lived application flows. 31

Figure 3.4 Measured overhead increase of Layer 4.5 socket taps and Layer 4.5 taps with sample customization applied to a single long-lived application flow. 33

Figure 3.5 NCO continuous management log with challenge-response security check. 34

Figure 3.6 Wireshark capture with Layer 4.5 inverse customization module applied. 35

Figure 4.1 Example of client TLS processing failure when customized data is present within the TLS payload. 41

Figure 4.2 Layer 4.5 tap and customization logic with added buffering capability for application receive message processing. 43

Figure 4.3 Layer 4.5 tapping and customization receive flow logic with buffering capability. 44

Figure 4.4 Layer 4.5 with buffering capability measured overhead of 1000 short-lived application flows. 48

Figure 4.5 Layer 4.5 with buffering capability measured overhead of a single long-lived application flow. 49

Figure 4.6 Layer 4.5 with buffering capability measured overhead of a single long-lived encrypted application flow. 50

Figure 5.1 Layer 4.5 capable wide area network. 54

Figure 5.2 Updated Layer 4.5 NCO to support customization rotation. 56

Figure 5.3 General NCO customization rotation process 60

Figure 5.4 Layer 4.5 capable wide area network testbed. 62

Figure 5.5 Layer 4.5 customization immediate attachment log and corresponding Wireshark packet overlay. 64

Figure 5.6 Layer 4.5 customization activation log and corresponding throughput graph. 65

Figure 5.7 Layer 4.5 customization deprecation log and corresponding throughput graph. 67

Figure 5.8 Single device Layer 4.5 customization rotation log and corresponding throughput graph. 68

Figure 5.9 NCO customization rotation for two devices. 70

Figure 5.10 Multiple device Layer 4.5 customization rotation log and corresponding Wireshark packet overlay. 71

Figure 5.11 DNS header with data fields. 73

Figure A.1 TCP/IP network stack with surveyed protocols. 85

Figure A.2 MPLS packet showing 32-bit MPLS header 86

Figure A.3 IPv4 and IPv6 packets showing applicable extension fields for customization 87

Figure A.4 Google IPv6 adoption tracker 88

Figure A.5 TCP packet showing Options field for customization support . . . 89

Figure A.6 UDP packet showing proposed UDP Options field for customization support . . . 90

Figure A.7 TLS over TCP packet vs. TCPLS packet . . . 91

Figure A.8 TLS over TCP packet vs. QUIC packet . . . 92

Figure A.9 Network packet showing INT header and associated metadata inserted between packet header and application data by a network device . . . 96

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Table 2.1 Parameters embedded per-module 9

Table 2.2 Monitoring and security intervals per-module 10

Table 2.3 Layer 4.5 module API 15

Table 3.1 Overhead testing customization module lines of code 29

Table 4.1 Updated customization module lines of code 47

Table 5.1 Layer 4.5 DNS customization middlebox interference results . . . 74

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AES	Advanced Encryption Standard
CIB	Customization Information Base
DCA	Device Customization Agent
DNS	Domain Name System
DPDK	Data Plane Development Kit
eBPF	Extended Berkeley Packet Filter
EDNS(0)	Extension Mechanisms for DNS
HTTP	Hypertext Transport Protocol
HTTPS	Hypertext Transport Protocol Secure
INT	In-Band Network Telemetry
IP	Internet Protocol
L3AF	Lightweight eBPF Application Framework
LAN	Local Area Network
LOC	Lines of Code
MITM	Man-in-the-Middle
MPIP	Multipath IP
MPLS	Multiprotocol Label Switching
MPTCP	Multipath Transmission Control Protocol
MPUDP	Multipath User Datagram Protocol
NCO	Network-Wide Customization Orchestrator
NFV	Network Function Virtualization
NIC	Network Interface Card
PID	Process ID

QUIC	Quick User Datagram Protocol Internet Connection
RFC	Request For Comments
SDN	Software Defined Network
SQL	Structured Query Language
TCP	Transmission Control Protocol
TGID	Thread Group ID
TLS	Transport Layer Security
UDP	User Datagram Protocol
VM	Virtual Machine
VPN	Virtual Private Network
VTL	Virtual Transport Layer
WAN	Wide Area Network
XDP	eXpress Data Path

Acknowledgments

To my dissertation advisor, Professor Xie: I appreciate you believing in me and pushing me to complete my dissertation. Through multiple meetings and paper submissions you helped me evolve my initial idea into a full dissertation-worthy topic.

To my friend and colleague, Karl: Thank you for the recommendation and encouragement to pursue this PhD opportunity. I have learned so much from you and appreciate you agreeing to continue supporting me by joining my committee. I hope I have the chance to work with you again in the future.

To my dissertation committee: Thank you for being part of my PhD journey and imparting your wisdom on me. Each of you provided a unique perspective that challenged me to abstract my ideas and broaden my understanding.

My fellow PhD students: It was not easy to be the only PhD student in the CS department for my first 9 months. I greatly appreciate your assistance with preparing for exams, the countless hours of listening to me talk about my topic, and the reviews you provided to my conference submissions. I hope you continue to support each other and that we can work together again in the future.

Ken and Eric: Thank you for your help in evolving my ideas and testing my code. I enjoyed the frequent progress meetings to help each of you finish your master's thesis and being able to collaborate with you on conference papers.

To my wife Liz: Thank you for your continued love and support while I pursued yet another degree. I am forever grateful to you!

To my son Dillon: It was an interesting experience getting to work in the same office as you for over a year. We had many hours of fun playing games and getting into anime that I will always remember.

: Create a 2/2 Dillon token with infect

THIS PAGE INTENTIONALLY LEFT BLANK

Publications

The following peer-reviewed publications directly contribute to the completion of this dissertation:

- D. Lukaszewski and G. Xie, “Towards Software Defined Layer 4.5 Customization,” in *IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp.330-338. DOI 10.1109/NetSoft54395.2022.9844096

- K. Pittner, D. Lukaszewski, and G. Xie, “An Empirical Study of Application-Aware Traffic Compression for Shipboard SATCOM Links,” in *IEEE Military Communications Conference (MILCOM)*, 2021, pp. 213–218. DOI 10.1109/MILCOM52596.2021.9653123.

- E. Bergen, D. Lukaszewski, and G. Xie, "Data Exfiltration via Flow Hijacking at the Socket Layer," (Publication Pending) *56th Annual Hawaii International Conference on System Sciences (HICSS-56)*, 2023.

- D. Lukaszewski and G. Xie, “Demo: Towards Software Defined Layer 4.5 Customization,” in *IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp.240-242. DOI 10.1109/NetSoft54395.2022.9844104

The following peer-reviewed publications occurred during PhD research and are considered relevant towards the dissertation topic:

- M. Sjoholmsierchio, B. Hale, D. Lukaszewski, and G. Xie, “Strengthening SDN Security: Protocol Dialecting and Downgrade Attacks,” in *IEEE 7th International Conference on Network Softwarization (NetSoft)*, 2021, pp. 321–329. DOI 10.1109/NetSoft51509.2021.9492614.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

Traditionally, protocol customization is about extending existing protocols with additional features, primarily for performance and security reasons. Several network protocols provide customization support using special header fields with yet-to-be-defined values. For example, the Transmission Control Protocol (TCP) provides an Options field that has been used to extend the protocol with features such as multipath capability, timestamps, and selective acknowledgements [1]. Another example is the Internet Protocol (IP) version 6 that provides a Next Header field to allow extending the protocol with multiple header segments and, unlike the TCP Options field, the inclusion of additional headers are not bound to a specific byte length [2]. Despite the built-in support for protocol customization, it can take years for extensions to become standardized and supported by common operating system kernels [3].

It is not just the customization extensions that can take a long time to become standardized. The introduction of new protocols can also take a significant time to be widely adopted or become standardized. First, kernel space protocols must undergo an extensive review process before new code can be incorporated into the operating system. Furthermore, once the protocol has been added to the base operating system it may not be immediately adopted by enterprise networks or utilized by applications. For example, IPv6 was introduced as a solution to the shrinking IP address space and has a Request For Comments (RFC) dating back to 1998, but as of July 2022 the protocol adoption was approximately 40% [4]. Second, user space protocols have more flexibility because they do not require kernel changes, but the standardization of the protocol can still take years. For instance, Quick User Datagram Protocol Internet Connection (QUIC) was designed to address latency-sensitive web services and was introduced as a draft RFC in 2016, but did not become an official RFC until 2021 [5].

Beyond the long deployment and standardization times, protocol customizations also suffer from network interference from middlebox devices [6]–[8]. A middlebox device is any device in-between the two communicating end-devices that processes packets beyond what

©2022 IEEE. Portions of this chapter were previously published. Reprinted with permission from D. Lukaszewski and G. Xie, "Towards Software Defined Layer 4.5 Customization," *IEEE NetSoft*, June 2022.

is performed by a standard router to include intrusion detection/prevention systems and firewalls [9]. These devices are generally controlled by different network providers, which makes it difficult to avoid middlebox interference when introducing functionality to protocols that does not match the RFC standard. Middlebox interference can come in different forms, ranging from completely dropping the traffic to tampering with headers (e.g., changing unknown header fields). For instance, the use of IPv6 customizations have been known to experience middlebox interference resulting in dropped network packets [7]. There are two main approaches taken by protocol developers to avoid middlebox interference: i) build in a fallback mechanism or ii) use encryption.

First, the fallback mechanism can be seen by the Multipath Transmission Control Protocol (MPTCP) and by the QUIC protocol. MPTCP signals its use with a specified TCP Option value during the TCP 3-way handshake. If this Option value is tampered with by a middlebox device, then the connection falls back a normal TCP connection [10]. QUIC is a newer user space transport protocol that runs over a standard User Datagram Protocol (UDP) connection [5]. Middlebox interference to QUIC comes in the form of filtering or restricting UDP traffic, which is generally less common on networks. To account for this potential interference, QUIC provides a fallback mechanism to transition to TCP, which is less likely to be filtered [5].

Second, protocols can use encryption to avoid middlebox interference because middlebox devices are typically unable to decrypt and inspect/filter the packet. This encryption approach is evidenced by QUIC and the newly proposed TCPLS protocol. First, QUIC encrypts the majority of the application data, which prevents middlebox devices from inspecting and tampering with protocol header values [5]. Second, the TCPLS protocol was designed to combine the Transport Layer Security (TLS) protocol with TCP in an effort to bypass middlebox interference to TCP customizations and to also allow expanding TCP beyond what is capable in the standard header [11].

Despite long standardization times and middlebox interference resulting in burdens to deployment, protocol customizations remain relevant in today's networks. Recent work has leveraged Extended Berkeley Packet Filters (eBPFs) [12] modular design and security verification to perform protocol customizations in both user space and kernel space. The Walmart Lightweight eBPF Application Framework (L3AF) project [13] and the proto-

col plugin work [14], [15] 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 layers of the network 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 centralized control and continuous management of deployed customizations that would provide greater customization support to enterprise networks.

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 the Hypertext Transport Protocol (HTTP)) [16], [17] and/or intentionally varying application message formats and messaging patterns [18] 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 [19]–[21].

In both use cases, 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 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 datacenters, which are large in size and must uphold stringent security and performance requirements at all time [22]. Therefore, in this dissertation we explore an approach based on *network-wide orchestration*, by leveraging the growing adoption of Software Defined Networks (SDNs) in enterprise and data-center networks. As we will demonstrate, introducing a Network-Wide Customization Orchestrator (NCO) allows operators to deploy and continuously monitor protocol customization

on all devices from a single vantage point. Furthermore, the NCO can provide the much-needed real-time coordination of middlebox traversal to address well known interference problems [6]–[8] as well as timely mitigation of rogue devices and other types of attacks, such as traffic hijacking.

A straightforward method of using a SDN controller to support protocol customization is to virtualize all devices in the network and deploy completely new Virtual Machines (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. Furthermore, traditional enterprise networks such as Department of Defense networks are not fully virtualized and, thus, could not utilize this method of customization deployment. Therefore, in this dissertation we explore a design that supports dynamic "hot" insertion of software modules to devices to modify the behaviors of these devices on the fly, without rebooting the device or restarting the targeted services. 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 datacenters 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.

Modern operating systems provide a rich set of mechanisms [23]–[25] 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 (e.g., Chrome, Firefox, `wget`, `curl`) invoke the same application protocol (i.e., HTTP). Recent work towards network-application integration [26]–[28] 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 our customization taps constitute a shim layer between the application and transport layers, which we call "Layer 4.5".

1.1 Thesis

The thesis of this dissertation is:

> *A software defined Layer 4.5 protocol customization architecture is feasible and can provide continuous management capabilities, compatibility with modern encryption protocols, and rotating customizations on active application flows.*

This dissertation makes novel contributions through the iterative design and evaluation of the Layer 4.5 customization architecture. The organization of this dissertation to realize these contributions is as follows:

- Chapter 2 presents the design of the Layer 4.5 customization architecture to enable a software defined approach to protocol customization within enterprise and datacenter networks. We provide a high-level design of each component of the architecture to include the continuous management capabilities and how to achieve application transparent, process-level flow customization.
- Chapter 3 implements a prototype of the architecture and evaluates the overhead of customization distribution, tapping the network stack at Layer 4.5, and customizing different types of application flows. Additionally, we prototype the security and middlebox traversal support capabilities the architecture enables.
- Chapter 4 expands the Layer 4.5 architecture design to provide a generalized customization capability that allows customizing applications with strict receive message processing, such as those utilizing TLS for application encryption.
- Chapter 5 demonstrates the Layer 4.5 architecture's ability to perform customization synchronization between multiple devices to include rotating customizations on active application flows. In particular, we account for customization distribution delays present in Wide Area Networks (WANs) and design methods to support customization synchronization. We evaluate our design using a GENI [29] testbed and finish with a cursory evaluation of third-party middlebox interference of Layer 4.5 customized application flows.
- Chapter 6 summarizes the significant contributions presented in this dissertation, addresses reproducibility of results, and provides areas for future work.

- The appendix provides a survey of current network protocols and the customizations applied to each protocol over the years. We then present recent protocol customization work to include our own previous published research.

CHAPTER 2:
Design of the Layer 4.5 Customization Architecture

In this chapter we present the design of the Layer 4.5 customization architecture. Illustrated in Figure 2.1, the architecture 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.

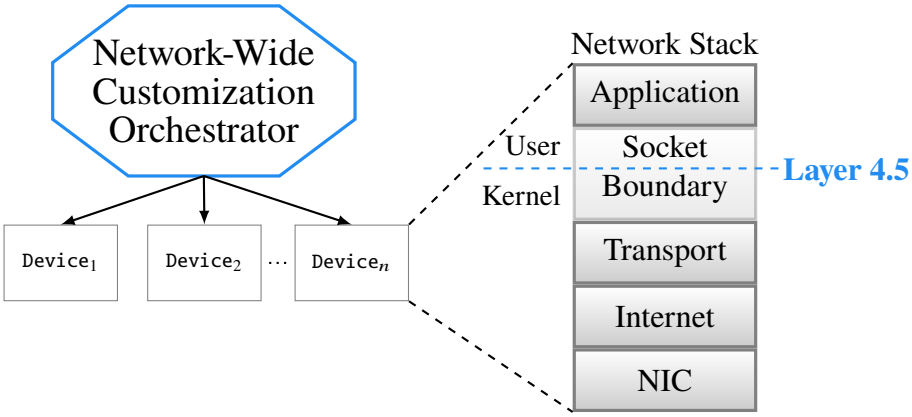


Figure 2.1. Proposed architecture for centralized control of protocol customization in a network.

We begin by discussing the NCO components necessary to provide customization distribution and subsequent continuous management. Next, we introduce the Device Customization Agent (DCA) and associated customization modules to support customization automation on each device. We then expand on how Layer 4.5 supports per-network additional security and middlebox traversal requirements.

©2022 IEEE. Portions of this chapter were previously published. Reprinted with permission from D. Lukaszewski and G. Xie, "Towards Software Defined Layer 4.5 Customization," *IEEE NetSoft*, June 2022.

2.1 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 2.2, 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 the new functionality.

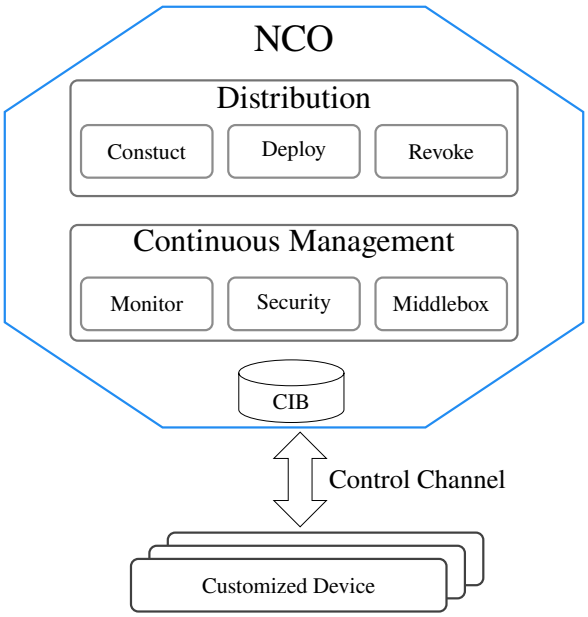


Figure 2.2. Layer 4.5 NCO consisting of "distribution" and "continuous management" functions, a CIB for tracking deployed customization modules, and an encrypted control channel to customized devices.

2.1.1 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 in the network.

Construct function: Responsible for building the per-device customization module to include embedding the Table 2.1 parameters and storing all values in the CIB. Each customization module is linked to a device via the *mod_id* parameter. Each device uses the *mod_id* when communicating with the NCO, thus a per-device unique *mod_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 Subsection 2.1.2 and Section 2.3, 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 (i.e., source IP address and port number, destination IP address and port number, transport protocol). 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, `dnsmasq`) 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 flows. 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. In Section 2.2 we discuss how the application label is tied to a process on a tapped socket.

Table 2.1. Parameters embedded per-module

Parameter	Purpose
mod_id	Per-device unique module ID
active_ts	Timestamp of most recent customization performed
init_key	Initial key for security functions (shared with NCO)
tap_socket	5-tuple ID of socket to tap (application label in place of dynamically generated ports)

Deploy function: Supports the transport of constructed 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. At the specified deployment time, the NCO delivers the customization module over the established control channel and awaits confirmation that the module was installed and registered with Layer 4.5. Upon confirmation, the per-module intervals from Table 2.2 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.

Table 2.2. Monitoring and security intervals per-module

Parameter	Purpose
state_req_window	Period between state report requests
sec_check_window	Period between security checks

Revoke function: Supports the removal of outdated or misbehaving customization modules from a customized device. 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. At this point, new sockets will not be matched against the revoked module and all previously customized active sockets on the device are no longer customized by the revoked module.

2.1.2 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: 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* recorded by the module 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:                 Generate alert(s)
16:         else  Perform query processing
17:         Update CIB
18: end while
```

Security function: Provides a mechanism for adding per-network module security requirements to match a given threat model. At this stage of our design, we consider an attacker who is capable of monitoring all network traffic, but we also assume the attacker does not have the capability 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 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 2.3.

Middlebox function: When the network middlebox is able to support Layer 4.5 and conducts the middlebox processing above Layer 4.5, then the middlebox function works with the deployment function to distribute the appropriate customization module to the middlebox to

allow processing the received message. If the middlebox processes packets in kernel space below Layer 4.5, then customization modules will not be invoked by Layer 4.5 and allow customization processing prior to middlebox processing. To address middleboxes fitting this processing method, we expand the middlebox function responsibilities.

We do not enforce Layer 4.5 customization capability on each middlebox within the controlled network. 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 that 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 triggering an unknown customization alert 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 an alternate method of supporting local middlebox customization processing with pre-installed customization inverse modules in Section 2.4.

2.2 Automation of Customization of Devices

Each device with Layer 4.5 capability supports the automatic installation and removal of customization modules directed by the NCO. Figure 2.3 illustrates the device customization architecture to include the DCA and the customization modules for application transparent insertion into Layer 4.5.

2.2.1 Device Customization Agent

The DCA serves two main functions on the customized device. First, 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.

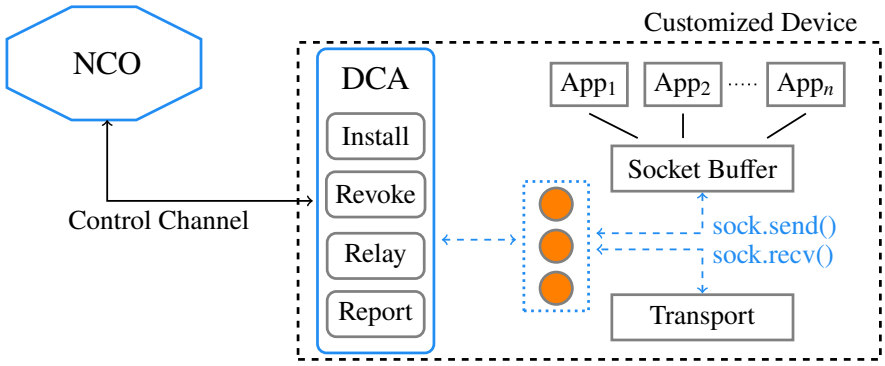


Figure 2.3. Layer 4.5 device architecture. The DCA control channel with the NCO is used to receive and install customization modules (orange circles), which are invoked through the socket-transport tap.

Second, the DCA is responsible for the management of all customizations installed on the device to include the application transparent insertion of each customization in the TCP/IP network stack at Layer 4.5.

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 customization module construction. After initial check-in, the DCA awaits further commands from the NCO, which invoke the appropriate handler function.

Install handler: Invoked when the NCO needs to deploy a previously constructed customization module to the device. The handler accepts a customization module delivered over the control channel, installs it onto the device, and registers it for Layer 4.5 customization. At this point, all sockets matching the customization module *tap_socket* parameters will be customized.

Revoke handler: Invoked when a customization module is marked for removal due to being obsolete/replaced, failure of a security check, or mismatch with corresponding end device. The revoke handler will unregister a customization from Layer 4.5 and delete it from the device so that it can no longer be used. At this point, all sockets previously customized by the revoked module will no longer be customized.

Relay handler: Invoked when the NCO issues a command, such as a security challenge, to a specific module installed on the device. The relay handler is responsible for communicating with the module to issue the command on behalf of the NCO and deliver the module's response to the NCO. The relay handler is further discussed in Subsection 2.2.2.

Report handler: Invoked when the NCO *state_req_window* expires and the NCO requests a state report. The report handler constructs a device-level report listing all registered modules and their use statistics, not previously reported revoked modules, and any device specific information that may be required by the NCO.

After a customization module is installed on the device, the DCA will automatically link application sockets to matching customization modules, which is discussed further in Subsection 2.2.2 and Subsection 2.2.3. More importantly, the DCA ensures that the application is unaware that customization is taking place at Layer 4.5. For instance, consider a customization that removes data received by the transport layer before it reaches the application. On the receive message path of Figure 2.3, the DCA does not simply intercept the application message from the transport layer, pass it to the customization module to modify it, and return the result to the application because the receive message result indicates the number of bytes removed from the transport buffer, which would not match the amount of data inserted into the application buffer received by the application. This same scenario matches the send path when a customization module modifies the amount of data being sent and would result in layer 4 indicating that it sent a different number of bytes than the application intended. These mismatches could result in unexpected application behaviour and, thus, requires the DCA to hide these mismatches from the application to maintain transparency when customizing at Layer 4.5.

2.2.2 Customization Modules

Layer 4.5 customization modules are the basic building blocks for realizing per-process protocol customization requirements. Layer 4.5 customization modules are attached to a socket based on the modules *tap_socket* parameters. Thus, each module includes the socket flow matching parameters of Table 2.1 and standard functions to separate the processing of ingress and egress messages at the sender and receiver, respectively. 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 behaviour at the cost of necessary deconfliction and management of deployed customizations, which occurs on the NCO.

Table 2.3 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 2.3, the *cust_send* and *cust_recv* functions are invoked to perform the necessary customization by the Layer 4.5 tap when a corresponding socket system call is conducted. The *state_report* function is responsible for reporting module parameters required by the NCOs 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*.

Table 2.3. Layer 4.5 module API

Function	**Purpose**
cust_send()	modify outbound message
cust_recv()	modify inbound message
state_report()	report monitoring statistics
sec_respond()	reply to security check

Customization module developers may desire the ability to establish a control channel between two customization modules. For instance, the plugin work [15] used such a channel to negotiate plugin use across two devices. We view control channels at Layer 4.5 to be problematic from a security and overhead point of view. Furthermore, the NCO is utilized to manage deployed customization modules, which nullifies the need for the control channel. Therefore, Modules are not permitted to establish sockets and perform socket send and receive calls. Instead, modules are only permitted to modify the contents of application

messages when the application conducts a socket send and/or receive call. This restriction is placed on the modules to not only prevent forming module control channels but also prevents increased delays to application traffic. Additionally, the restrictions act as a security measure to limit the modules from completely hijacking a socket. Module restrictions are either enforced by the NCO during module construction or through manual review of each customization module.

We now expand on how the module defines the *tap_socket* parameters to match the desired application socket flows. To enable fine-grained application matching, we include both the application name of the current Process ID (PID) and the name associated with the Thread Group ID (TGID). For example, during initial experimentation, we experienced the `dig` application has a PID task name of `isc-worker-0000` and a TGID name of `dig`. Thus, a customization module could target any application using the `isc-worker-0000` PID, which includes `dig`, or the module could specify attaching to only `dig` by using the TGID.

The *tap_socket* parameters are determined from the point of view of an outgoing message on the corresponding device to enable differentiating flows that are destined for the local host. Consider the following two flow matching examples:

1.	Client:	1.1.1.1	**Chrome**	2.2.2.2	80	TCP
	Server:	2.2.2.2	80	1.1.1.1	**	TCP
2.	Client:	**	**dig**	3.3.3.3	53	UDP
	Server:	3.3.3.3	53	**	**	UDP

The first example corresponds to a `Chrome` client connecting to a web server using TCP. The second example applies to a client using `dig` to send a request to a Domain Name System (DNS) server using UDP. In these examples, the client's port number is not known by the NCO since it is dynamically allocated by the host when the socket is created. Therefore, the NCO specifies the applications name for client modules to be used to enable matching the socket. For server modules, the destination port will be unknown for the same reasons and can be set as a wildcard value. The server's application name can also be specified for server modules, but application servers will generally bind to specific IP addresses and port numbers, which allows matching the socket without needing the application name.

Lastly, there are two main types of customization modules:

1. Interactive: May modify the contents of the message buffer. This is the most common type of customization module.
2. Monitoring: Observe the message buffer contents, but do not modify them. These modules are useful for gathering statistical information that could be shared with the DCA/NCO.

This dissertation focuses on the use of interactive modules and the complexities that arise when deploying and monitoring their use within the network. Monitoring modules are similar to current methods, such as those used by Wireshark, but could provide a different perspective via a unique monitoring location. We leave the analysis of Layer 4.5 monitoring modules and their use cases to future work.

2.2.3 General Layer 4.5 Tap and Customization Logic

Now that we have discussed the Layer 4.5 device architecture to include the DCA and customization modules, we can provide a general model for tapping and customizing application flows. Conducting application customization with a process-level granularity may lead to excessive processing overhead. To address this overhead concern, Layer 4.5 is designed to adhere to application behaviour as strictly as possible and only perform customization processing after an application performs a socket send or receive message call. Additionally, we will leverage the application buffer for customization processing whenever possible to avoid unnecessary allocation of additional customization buffers. This means that the customization modules are supplied with the application's message buffer when performing the desired customization operation. Figure 2.4 illustrates the general model for tapping and customizing application send and receive messages. Note that the logic differs for the send and receive message paths because the *tap_socket* information available at the socket-transport tap and the end destination will differ.

The application send path processing (left) is triggered when the application makes a send message call delivering data to the application socket where it is intercepted by the Layer 4.5 socket tap. If the socket has not been processed before, then a customization socket is created. The customization socket creation includes the necessary customization module lookup and attachment if a matching module is found. The customization socket is then

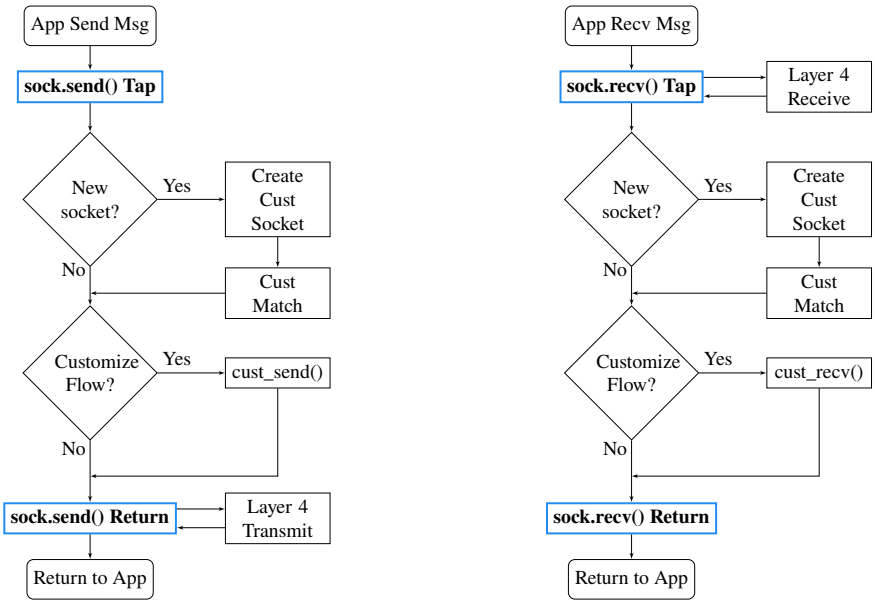


Figure 2.4. General Layer 4.5 tap and customization logic for application send (left) and receive (right) message processing.

checked to determine if the application flow is being customized and if it is, then the flow is diverted to the *cust_send* function for customization processing prior to being delivered to the transport layer.

The application receive path processing (right) is triggered when the application makes a receive message call. The Layer 4.5 socket tap intercepts the application receive message call and performs the transport receive message call before the customization lookup process to ensure all socket parameters are available for customization matching. Since we allow customizing all applications, we must account for those applications that do not bind to a particular socket and, therefore, will not have all socket parameters set prior to receiving the message. The Layer 4.5 tap leverages the previously allocated application socket message buffer to hold the potentially customized message to minimize additional overhead for non-customized flows since these flows will have the data immediately returned to the application. For customized flows, the data will be redirected to the *cust_recv* function for customization processing prior to being delivered to the application.

2.3 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 that verification is performed during the module loading process [30]. Each module deployed in the network is embedded with a function for invoking the desired module security check, as seen in Figure 2.5, and an *init_key* that can be adapted using ratcheting [31] techniques or key derivation functions [32] similar to what is done in previous protocol dialect work [18].

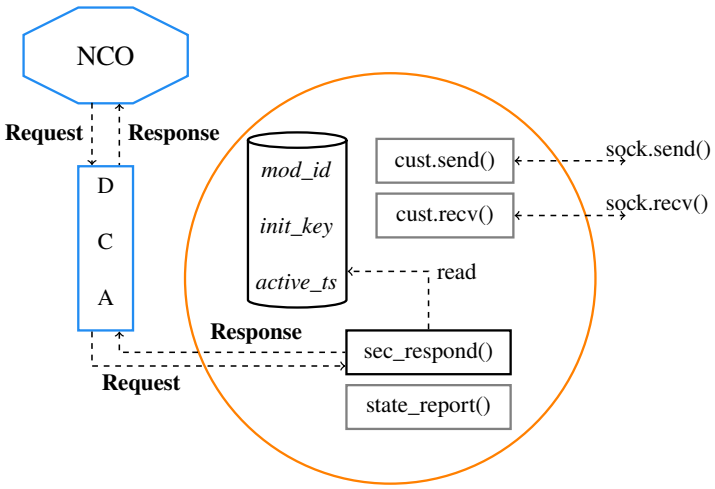


Figure 2.5. Customization module with embedded security functionality.

One particular security function to check the validity of deployed modules would be to utilize a challenge-response authentication protocol [33] 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 challenge message to the module via the DCA relay function. Note that the DCA does not have the capability to decrypt the challenge as the *init_key* and any subsequently generated keys are only present on the customization module and NCO. When the module *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 can then verify the response and either revoke the module due to a failed response or update the CIB accordingly. This security function use case is prototyped in Section 3.3.

2.4 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 the middlebox can perform normal message 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 3.4 contains a demonstration of a middlebox inverse function used during deep packet inspection.

2.5 Limitations

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 or below the transport layer (i.e., layer 4), such as receipt of TCP acknowledgements. This limitation is relevant in situations where one device receives a customized message, but the application does not send a message in response, which does not allow the Layer 4.5 customization module to respond either. For example, consider a HTTP connection that is being customized such that the web server customizes the response to GET request by altering the contents of the file requested. The client does not typically respond to the web server during the request outside of sending TCP acknowledgement messages and,

thus, the client side customization would not be able to reliably respond to the web server customization module. If these triggers are necessary, then a lower layer solution should be used, perhaps in conjunction with a Layer 4.5 customization module.

Limiting module activity to match application socket calls also means that Layer 4.5 customizations need to be designed to fit the unique message processing logic of tapped applications, which can be different for sending and receiving data. For instance, one application could send one IP packet length of data at a time to the socket, while a different application sends a 65 KB buffer to the socket and relies on the lower layers to segment the buffer into chunks that will fit into IP packets. Additionally, when receiving data from layer 4 the application may not always use a constant size receive buffer. At first, the application may expect large amounts of data and pass a correspondingly large buffer to layer 4, but as the amount of expected data decreases, the application buffer may decrease as well. For example, a client requesting a file from a web server expects data corresponding to the file size. As the client gets closer to the expected size, the receive message buffer may be reduced to only request the expected remaining bytes.

The second limitation is that unexpected application behaviour influences the customization module development complexity. During our initial prototyping and testing, we 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 layer 4 to determine the byte length of the accompanying DNS request. Customization module developers will need to account for this type of behaviour to ensure an efficient design is chosen for the customization and the operation of the corresponding application remains intact.

Third, Layer 4.5 only allows for a single customization module to be applied to a matching socket, which enables predictable customization at the cost of necessary deconfliction and management of deployed customizations by the NCO. This also means a customization chain can not be formed on end devices and must be constructed as a singular module on the NCO. If chains could be formed on end devices, then we would need to guarantee each device forms the exact same chain to properly process customizations in the reverse order on the receiving device, which would increase complexity of Layer 4.5 and inhibit adoption.

Finally, application layer encryption will limit the types of protocol customizations to some degree. For instance, when application data is encrypted prior to reaching the socket, 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 customized data.

2.6 Summary

In this chapter, we designed a Layer 4.5 customization architecture to perform application transparent, fine-grained, process-level flow customization. The architecture consists of the Network-Wide Customization Orchestrator (NCO) to coordinate the deployment and continuous management of each customization, the Device Customization Agent (DCA) to automate the installation of customization modules, and the individual customization modules designed with the ability to target specific application flows. In Chapter 3 we build and evaluate an initial prototype of this architecture.

CHAPTER 3:
Prototyping and Evaluation

In this chapter our goal is to implement a prototype of the Layer 4.5 customization architecture presented in Chapter 2 to test customization distribution overhead and the processing overhead associated with Layer 4.5 insertion into the network stack. We begin with an implementation of the NCO and DCA control channel and an overhead evaluation of distributing a customization module over the network to a variable number of devices. Next, we implement and evaluate the overhead of the Layer 4.5 DCA for tapping application socket calls and attaching customization modules to the application sockets.[1] Note, this prototype simplifies the process of attaching a customization module to apply only to new sockets, which may require application restart after a matching module is registered. We finish with a prototype challenge-response NCO security function implementation and an example NCO assisted middlebox traversal.

The experiments in this chapter were performed on a testbed consisting of two Ubuntu 5.13 VirtualBox VMs running on an 8-Core Intel Core i9 MacBook Pro with 64 GB of RAM. To minimize differences between VMs and ensure reproducibility of experiments, we utilized Vagrant [35] and a base VM image configured to support Layer 4.5 installation. Vagrant was configured to allocate each VM 2 CPUs, 8 GB RAM, and a paravirtualized network adapter. The VMs were connected using an internal network configuration with a 1000 Mbps capacity to mimic the speeds that can be expected within internal network communications. Additionally, we did not include any traffic loss on the link or produce additional background network traffic, which allowed testing the overhead without the interference of network congestion. The prototype implementation and testing scripts are made available open-source on GitHub (https://github.com/danluke2/software_defined_customization).

©2022 IEEE. Portions of this chapter were previously published by IEEE. Reprinted with permission from D. Lukaszewski and G. Xie, "Towards Software Defined Layer 4.5 Customization" and "Demo: Towards Software Defined Layer 4.5 Customization," *IEEE NetSoft*, June 2022.

[1] Distribution and overhead results differ from [34] due to code improvements.

3.1 Distribution Overhead

Network customization controlled by a central service is limited to how quickly customization modules can be deployed in the network. Therefore, 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. Note, to simplify the prototype we did not enforce control channel encryption. The NCO was written in 1400 python Lines of Code (LOC)[2] and the user-space DCA component was written in 350 python LOC. The NCO per-device deployment process is outlined in Algorithm 2.

Algorithm 2 NCO: Per-Host Deployment Logic

```
1:  while True do
2:      Query CIB for customization modules marked for deployment
3:      for Module in deploy list do
4:          Deploy module binary to host
5:          if Success then
6:              Remove module from required deploy table
7:              Insert module in deployed table
8:          else
9:              Remove module from required deploy table
10:             Insert module in deployment error table
11:         end if
12:     end for
13:     Sleep interval
14: end while
```

After a customization module is built for a specific device as part of the construction function, the module is marked for deployment to the device either immediately or at a specific time in the future. When the module is marked for deployment, the module binary is distributed to the device using the control channel and a success or failure notification is returned. This notification is used to update the CIB accordingly. If a failure occurs, the NCO removes the module from the deployment required table and reports the error condition by putting the module into a separate module error table. If the deployment is successful, the CIB is updated to reflect the module's deployment and the module is removed from the deployment required table.

[2] All reported line of code values are approximated

To test the NCO distribution overhead we established the CIB as a Structured Query Language (SQL) database, developed a sample customization module, and established the control channel between each device DCA and the NCO. Note that 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. Thus, prior to distribution testing, the customization module was constructed for each device and stored in the CIB for subsequent deployment. For reference, in our testbed the customization module construction time was approximately one second per module.

To understand the distribution limitations of the NCO, we vary the number of devices on the network for each test. Furthermore, we simplify the experiment by emulating multiple devices using a new socket on the client to represent a new device. The client is configured to spawn a new process for each device in the test, create a socket connection with the NCO (i.e., control channel), and then send a unique identifier to appear as a new device from the perspective of the NCO. Since we emulate multiple devices on a single machine and each of these emulated devices runs on the same Layer 4.5 implementation, we do not include the module registration process with Layer 4.5 as part of the deployment test. Instead, we simulate the registration of the module and report a successful installation to the NCO.

Figure 3.1 shows the deployment time results of 15 rounds of distributing a single 600 KB 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 modules marked for deployment (line 2) and the updates necessary to reflect such deployment (lines 6-10). Note that some overhead can be contributed to emulating each device as a separate process and socket on a single machine, but we minimize this overhead by using multiprocessing and allotting the VM multiple processors and sufficient RAM.

To address network scalability, we envision that NCO use can mimic that of SDN networks with multiple SDN controllers that support multiple switches. Thus, operators should utilize multiple NCO instances as supported by SDN platforms such as ONOS [36]. The adaptation of the NCO to a SDN application running on an ONOS controller is left as future work.

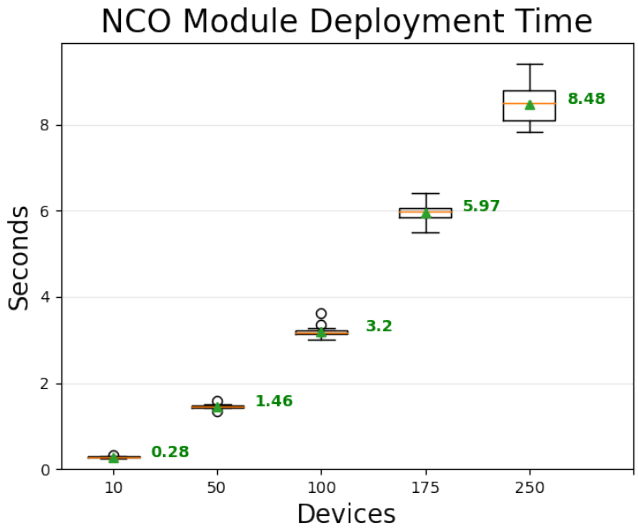


Figure 3.1. Measured latency of distributing a new module to 10, 50, 100, 175, and 250 devices, respectively. Green values show mean deployment time.

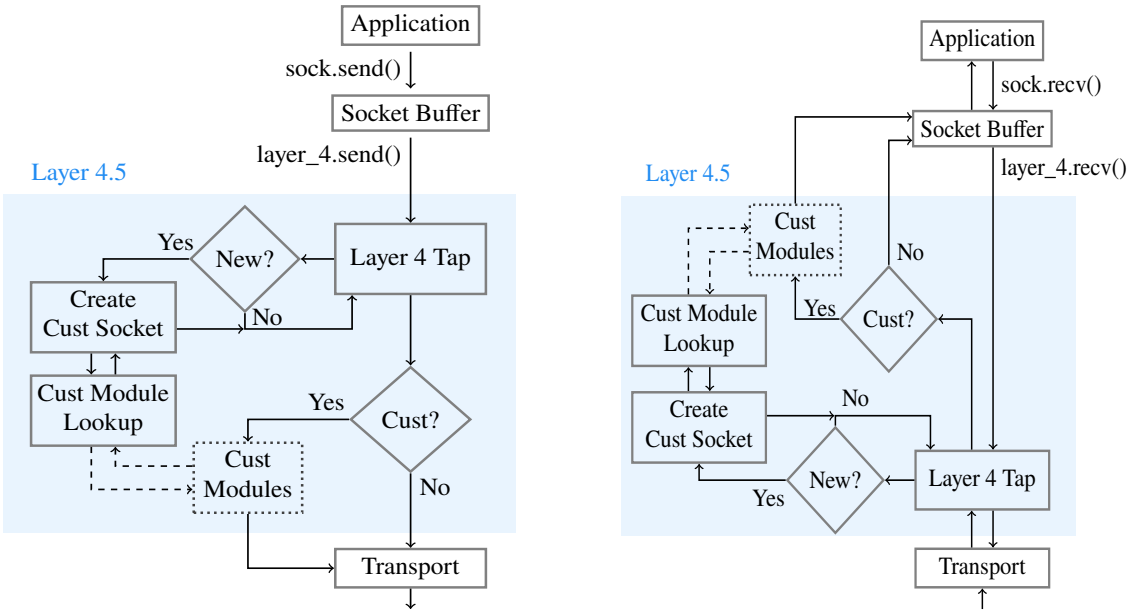
3.2 Processing Overhead

In this section we begin by describing the Layer 4.5 tapping and customization logic to realize application transparent customization. We then discuss the customization modules used for testing the processing overhead of tapping application sockets and subsequently customizing them. Finally, we discuss the overhead measurement results.

3.2.1 Layer 4.5 Prototype

Before we evaluate the processing overhead of Layer 4.5, we first discuss how the Layer 4.5 prototype performs the application transparent socket taps and customization redirection from Subsection 2.2.3. Figure 3.2 illustrates the Layer 4.5 prototype send and receive message processing logic.

The socket taps of Figure 3.2 are accomplished without kernel code modification in two steps. First, Layer 4.5 creates backups of the global function pointers to TCP/UDP send and receive calls, such as `tcp_prot.sendmsg` and `tcp_prot.recvmsg`. Next, Layer 4.5 replaces the global pointer with a pointer to a new function with necessary logic to determine if the socket requires customization. If customization is required, Layer 4.5 will hand application flows off to the matching customization modules for intermediate processing



(a) Layer 4.5 send flow logic

(b) Layer 4.5 receive flow logic

Figure 3.2. Layer 4.5 application flow tap and customization logic. The blue section represents the new Layer 4.5 logic introduced to the network stack.

before resuming TCP/UDP calls through the backup pointers. If the socket is not customized, Layer 4.5 will use the backup pointers to resume the desired application call. Next, we provide additional details specific to the Layer 4.5 send and receive path processing and optimizations for TCP flows.

Send Processing:

Figure 3.2a illustrates the customization process when an application transmits data. The customization flow starts when the socket tap intercepts the transport layer send message call (e.g., `udp_prot.sendmsg`). The first decision the Layer 4.5 tap must make is whether the current socket is new or if it was previously processed. This decision is based on a combination of the socket's internal pointer value and the PID owning the socket. Using these values instead of the *tap_socket* parameters allows differentiating sockets that may be re-using the same IP/port values.

When a new socket is identified, a corresponding customization socket is created and stored for the life of the socket. Each customization socket undergoes a customization module lookup process to attempt matching the socket parameters with a registered customization module (dotted lines). During this lookup process, the *tap_socket* application label is used to match the customization to the application owning the socket and assign values to any wildcard *tap_socket* parameters. If a registered module matches the socket, the module's *cust_send* function pointer is stored in the customization socket for future use. After the lookup process finishes and the customization socket is finalized, the socket tap delivers the application's message buffer to the *cust_send* function or to the transport layer if no customization is necessary.

Receive Processing:

Figure 3.2b illustrates the customization process when an application receives data. The customization flow starts after the socket tap intercepts the transport layer receive message call. Prior to any customization, the socket tap first performs the receive message call on behalf of the application to fill the application's message buffer with the potentially customized message and assign any missing socket parameters (e.g., from a UDP socket). After receiving the message, the socket tap can determine if a new socket is being processed. Similar to the send process, a new customization socket is created initiating a lookup process to identify a customization module matching the receive parameters and the associated *cust_recv* function. After the lookup process finishes, the socket tap either delivers the message to the *cust_recv* function or to the application if no customization is necessary. If the message is customized, then the customization module copies the data in the application's message buffer, processes the customization as necessary, and then overwrites the data in the application buffer prior to returning to the application.

Note that if the send and receive path are both customized for a given application, then only one new customization socket is created. When either the first send or receive message call is performed, the customization lookup process will identify if the customization module applies to both the send and receive path. If this is the case, then the module is attached with the customization *cust_send* and *cust_recv* function pointers and the next send or receive message call will not identify the socket as a new socket for customization lookup processing.

TCP Optimization:

TCP application flow customization processing can be optimized during the new socket processing stage. TCP connections always begin with a connect function call by the client or an accept function call by the server, which handles the TCP 3-way handshake phase of the connection. Thus, by tapping the TCP connect and accept function calls (i.e., (`tcp_prot.connect` and `tcp_prot.accept`) in addition to the send and receive function calls, we can create the customization socket during the 3-way handshake instead of during the first send/receive message call. This results in a slightly longer handshake instead of causing a delay when the application is ready to send or receive data. Since UDP connections do not typically perform connect calls, we still rely on tapping the send and receive message calls to create customization sockets if required.

3.2.2 Customization Modules used for Overhead Tests

To understand the overhead of application customizations, we developed two tagging customization modules that target different types of application flows and perform relatively expensive in-kernel memory copy operations, which are likely to cause the most overhead. Each module will insert 32-byte tags into messages of a targeted application flow at set byte positions (e.g., every 1000 bytes). The tag insertion not only increases the amount of data to be transferred, but it also involves a minimum of two memory copy operations, which are likely required by most customization modules modifying the message contents. An overview of the customization modules is provided in Table 3.1.

Table 3.1. Overhead testing customization module lines of code

	Client Module			Server Module		
Flow Type	*cust_send*	*cust_recv*	Total	*cust_send*	*cust_recv*	Total
Short-lived	15	2	90	2	20	95
Long-lived	2	80	155	60	2	135

Recall from Subsection 2.2.2 that customization module developers are primarily responsible for the *cust_send* and *cust_recv* function logic. Beyond these functions, the modules are standardized to include the necessary flow matching parameters and functions to regis-

ter/unregister with the Layer 4.5 DCA. Thus, in this section we will focus on the *cust_send* and *cust_recv* logic necessary to perform the desired customization.

The first tagging module targets short-lived flows that send only one IP packet worth of data for each transmission. This customization module will insert the customization tag at the front of each application message under the assumption that the receiver will process the message prior to sending additional messages. Additionally, inserting the tag at the front of each message also changes how the message will be processed by non-customized sockets since the message headers will come after the customization tag. Thus, if the message is processed by a non-customized device, such as a middlebox, or if the customization module is not loaded properly by the receiving device, the message will not conform to the protocol standard and will cause processing errors. This module is relatively simple and can be attached to sockets that exhibit this type of behaviour, such as UDP sockets performing DNS requests.

The second tagging module targets long-lived flows that transfer large amounts of data and inserts a tag every 1000 bytes of transmitted application data. The long-lived customization module is more complex than the short-lived module because it must work with larger send and receive message buffers (i.e., buffers with more than a single packet worth of data). Since large amounts of data (e.g., 64 KB) can be sent from the application to the socket, the customization module must split the data multiple times to insert the tags. Additionally, the receiver will process messages of varying sizes dependent on the network conditions. For instance, the application may perform a receive message call when only a small portion of the transmitted data has been received and processed by the transport layer. Since the amount of data being processed can vary, the module must maintain a byte tracker to correctly remove the customization tag every 1000 bytes processed.

3.2.3 Prototype Evaluation

To evaluate the processing overhead of adding Layer 4.5 to the network stack, we conduct a series of experiments using the customization modules of Table 3.1 along with common network protocols. We begin with an evaluation using 1000 short-lived DNS over UDP application flows and then evaluate a single long-lived HTTP over TCP application flow.

DNS over UDP

The first flows we target are 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 an internal lookup that was simplified to respond 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 used the short-lived flow customization module to customize all `dig` generated DNS request to the local DNS server and apply a 32-byte tag to the beginning of each request. Recall that inserting the tag at the front of the request will force tag removal by the `dnsmasq` customization module before a legitimate request can be processed. To ensure we could measure the processing overhead experienced, we decided to conduct batch DNS requests consisting of 1000 different requests to the server, repeated over 15 trials. Figure 3.3 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.

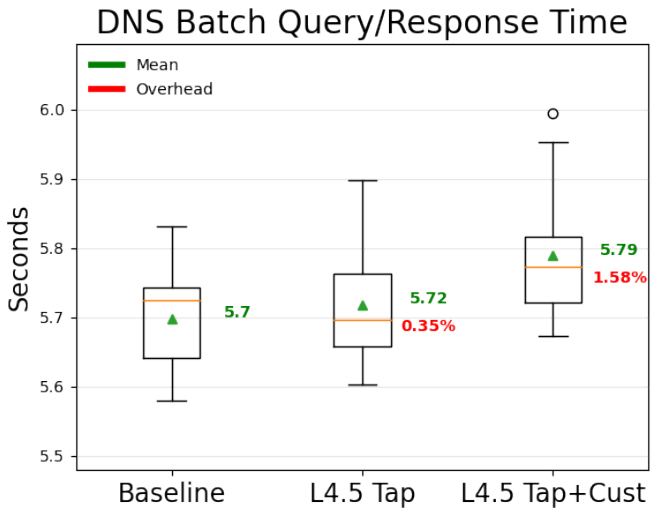


Figure 3.3. Measured overhead increase of Layer 4.5 socket taps and Layer 4.5 taps with sample customization applied to 1000 short-lived application flows.

From the resulting boxplots, we observed negligible Layer 4.5 tapping mean overhead. The minimal overhead was a result of each request being conducted using a new socket, which requires the creation of a customization socket and the subsequent module lookup process.

When each of the 1000 DNS messages are tagged, we see about 1.5% mean increase over the baseline or about 0.09 additional seconds to complete the requests. This additional overhead is a result of the 1000 tag insert (client) and 1000 tag delete (server) events and indicates the customization process did not add significant overhead on top of the tapping overhead.

HTTP over TCP

The next flow we target is a bulk file transfer, represented by a 3 GB Ubuntu image, using HTTP over TCP. When the Layer 4.5 customized `python` server accepts an incoming connection, the customization lookup process identifies the socket corresponding to the registered long-lived customization module. The assigned customization module is designed to track the bytes sent from the server application over TCP, inserting a 32-byte tag every 1000 bytes in a best-effort strategy to ensure at least one 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 3.4 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 to be the same on the client and server after each transfer completed.

From the boxplot, we see that the Layer 4.5 socket tap resulted in approximately 0.04% mean overhead. This negligible overhead primarily comes from the customization lookup process applied during each TCP send and receive call by the client and server and is less than that experienced by the DNS experiment because the customization socket creation process was only performed once. When the aggressive tagging customization is applied to the socket, the 3 GB of data are tagged every 1000 bytes, which results in approximately 3 million tag insert (server), 3 million tag delete (client) events, and an additional 96 MB of data processed. Each of these tag events resulted in at least two in-kernel memory copy operations, but only a modest 3% mean or 0.8 second increase to the file transfer time.

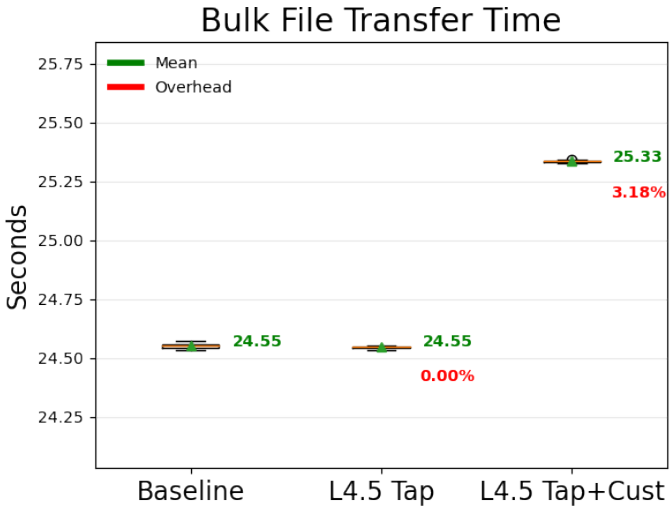


Figure 3.4. Measured overhead increase of Layer 4.5 socket taps and Layer 4.5 taps with sample customization applied to a single long-lived application flow.

3.3 Embedding Security Requirements

To test the NCOs ability to embed network security functionality into customization modules, we implemented the challenge-response customization module functionality (80 LOC) described in Section 2.3 under the threat assumption that each device is secure and the NCO/DCA control channel is protected with TLS. Our prototype implementation starts with the NCO generating a 256-bit module specific key, writing the key into the customization module, compiling the module to binary during the construction phase, and then storing the module and key in the CIB. After the customization module is deployed to the device and the DCA first reports that the customization module is registered, the *sec_check_window* starts in an expired status resulting in a module security check requirement.

When the NCO challenges a module, the module's key is retrieved from the CIB and used to encrypt a randomly generated 8-byte challenge using Advanced Encryption Standard (AES) encryption. This challenge and corresponding nonce is transmitted to the DCA, which invokes the relay handler to call the modules *sec_respond* function with the NCOs challenge as an argument. The module then uses the NCO embedded key to decrypt the challenge, append the NCO embedded *mod_id* to the end of the message, and then encrypt the message using a new nonce. The encrypted response message and corresponding nonce is then

relayed back to the NCO for verification. The NCO decrypts the message and if a failure is detected, the DCA revoke handler is invoked 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, recorded each challenge on the NCO and device via customization module trace logging, and reviewed the logs to ensure proper encryption/decryption was performed. Figure 3.5 provides the NCOs continuous management log results from a successful challenge-response check.

```
NCO_security  Challenge before encrypt as hex is 2f5318c1cb2405d2
NCO_security  Sent challenge request to 2, challenge:
              {'cmd': 'challenge', 'id': 1,
                'iv': '6ead272b71116e01aa5834a3e4c5d284',
              'msg': '6da85e7d6f56944cb3f489ce12863bb2'}
NCO_security  The decrypted response is 2f5318c1cb2405d2000001
NCO_security  Challenge string matches
NCO_security  Module_id matches
```

Figure 3.5. NCO continuous management log with challenge-response security check.

3.4 Assisting Middlebox Traversal

Consider a scenario in which a Layer 4.5 capable host is exhibiting unusual network behaviour, particularly through DNS queries. To investigate if non-standard applications are conducting DNS queries, the NCO operator deploys a new customization module to apply an application identification tag[3] to each DNS query conducted using the `dig` application, which is not normal client behaviour. 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 identification 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) to interpret the application identification tag and display it appropriately. Inverse modules follow a

[3]This tag can be encrypted in a way similar to the challenge-response process, or utilizing other mechanisms to mitigate forgeries.

slightly different construction and deployment process on the NCO. After the inverse module is constructed, an entry in the CIB is generated to include what type of middlebox the inverse module applies to and what corresponding customization module it matches. First, we tie the inverse module to a middlebox type, such as Wireshark, to allow a single module to apply to multiple middleboxes that have the same processing logic. Next, we link the DNS client customization module deployment with Wireshark inverse module deployment by inserting the customization and inverse modules in the CIB inverse module table. Now, when the NCO distributes the DNS customization module to the client, the corresponding inverse module (i.e., dissector script) is also distributed to the middlebox. To accomplish the installation of the inverse module, we expanded the user-space component of the DCA to install the inverse module in the plugin directory of Wireshark.

Figure 3.6 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. When a "standard" request or a reply is processed, the packet format does not match the customization parameters, which results in bypassing the inverse module and performing normal DNS processing.

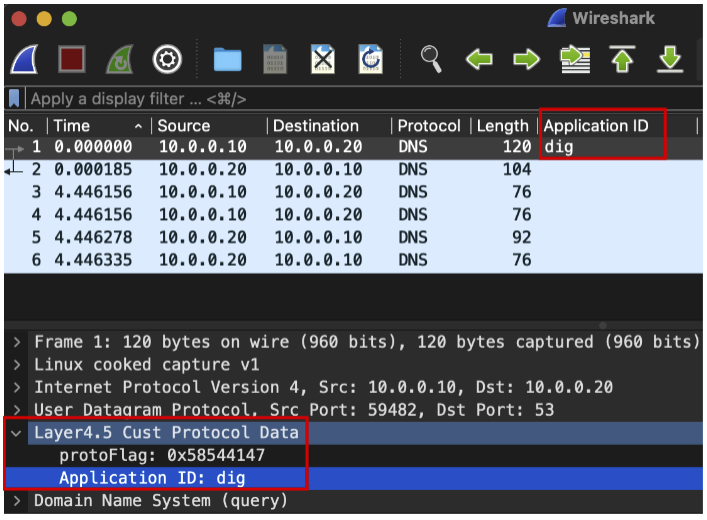


Figure 3.6. Wireshark capture with Layer 4.5 inverse customization module (i.e., dissector) applied to identify the presence of an application identification tag in DNS request packet contents.

3.5 Insights

During our prototype implementation of Layer 4.5 we experienced several challenges. In this section we will highlight the insights we gained while implementing application-transparent protocol customization.

- **PID Tracking:** The PID that created the socket processed by Layer 4.5, was not always the same PID that closed the socket. The result of this behaviour was that we would continue to track sockets that were no longer being used, which contributed to the larger processing overhead reported in [34]. This was primarily seen when conducting DNS requests when the new socket sending the current request would first close the previous request socket.

  We addressed this problem by first attempting to match the PID closing the socket to the stored customization socket hash table entries using the stored hash key. If the search did not return a matching socket, then we searched the entire hash table for a matching socket pointer value instead of the PID/socket pointer hash key. Future work could investigate different hash keys that allow for more efficient customization socket tracking.

- **Wildcard Parameters:** Customization module wildcard values are useful to allow matching parameters that have not yet been assigned to the socket. Additionally, wildcards allow matching multiple sockets. However, some customization modules may want to match multiple sockets but also process each socket differently based on the actual socket parameters. Therefore, we designed the Layer 4.5 socket tap to pass the current socket parameters to the customization module *cust_send* or *cust_recv* function, which allows the customization modules to identify what values the wildcards were matched against prior to performing customization actions.

- **Application Matching:** Attaching a customization module to a specific application may require using the name attached to the PID and/or the TGID. This may require additional work for customization developers to determine how to best attach to the targeted application. In Chapter 5 we expand the Layer 4.5 architecture to support testing customization modules, which should assist developers with correctly matching the application name.

- **Packet Tags:** It can be difficult for a customization module to apply a customization such that it will be present in each transmitted packet. This is because Layer 4.5 is above the transport and IP layers and applications may send large buffers to the socket that will be segmented into packets at the lower layers. Future work could include adjusting the Layer 4.5 send logic to allow the customization module to specify send message increments instead of transferring the entire buffer to the transport layer in a single send message call.

- **TCP Optimization:** The TCP socket that is accepting an incoming connection may not be the same socket that will be used to send and receive traffic. Therefore, the TCP accept function tap does not perform the customization lookup process until after the accept function call finishes and returns the potentially new socket pointer value. Additionally, if this socket is then used under a different PID, then the customization lookup will be repeated on the first TCP send or receive function call.

- **Misconfigurations:** Some applications may be performing a lot of socket calls, possibly due to misconfigurations. Since we tap multiple socket calls, this could result in unnecessary Layer 4.5 processing. For instance, we noticed in our Ubuntu Virtualbox machines that the `vminfo` process would constantly perform socket close calls without corresponding open calls. To determine if this behaviour is occurring, we utilized multiple DEBUG levels for logging to the Layer 4.5 customization log.

- **Middlebox Support:** Different middlebox devices process customized flows at different layers of the TCP/IP network stack, which means Layer 4.5 must support multiple methods to perform inverse customizations. For example, we demonstrated an inverse customization module applied to Wireshark, which receives a copy of the packet from layer 3. Therefore, it was not possible to process the customized flow at Layer 4.5 before Wireshark received the flow.

3.6 Summary

In this chapter, we developed a NCO, DCA, and Layer 4.5 prototype along with two different customization modules. Using the customization modules we evaluated the overhead of distributing and customizing application flows. The overhead of distributing customization modules over the network was primarily caused by the multiple queries and updates to

the CIB. The processing overhead experienced when customizing application flows was the result of the customization lookup process and the required in-kernel memory copy operations when adding or removing data as part of the customization process. Lastly, we prototyped a challenge/response security check compiled into the module during the NCO construction phase and demonstrated middlebox traversal of customized traffic using a customization inverse module.

CHAPTER 4:
Enabling Customization of Encrypted Flows

In this chapter our goal is to apply Layer 4.5 customization to encrypted application flows. We begin by providing a customization use case for encrypted flows followed by a motivating scenario and explanation of encrypted flow processing errors experienced with the initial Layer 4.5 prototype of Chapter 3. Then we design new Layer 4.5 customization logic that allows customization modules to request more data than the application and, as a result, buffer messages. This new customization logic is evaluated using the same experiments and testbed from Subsection 3.2.3 to not only determine the overhead experienced when customizing different types of traffic, but also to ensure proper operation with TLS flows.

4.1 Motivation

The Layer 4.5 evaluation of Chapter 3 focused on the customization of unencrypted application protocols, but many protocols are using encryption to not only add security to the protocol but also to avoid middlebox interference. Note that if Layer 4.5 is used to customize encrypted traffic 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 use case for customizing encrypted traffic flows to aid in network performance and security requirements. Then we finish with an operational scenario customizing encrypted traffic and the resulting customization processing complications, further motivating the need to expand the Layer 4.5 customization capabilities.

4.1.1 Use Case: Traffic Classification

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 [37]. Using traffic classification, a network can prioritize targeted traffic flows to meet quality of service obligations. One method of traffic classification to aid quality of service network routing is to utilize the 6-bit IP differentiated services field codepoint (DSCP)

header value that replaced the original type of service field [38]. However, this field may be unreliable due to the potential for middlebox interference [39], and the 6-bit header space for this field limits its use for fine-grained application specific tags on a per-network basis. From a security perspective, traffic classification can be used to aid network forensics and help identify malicious traffic.

To automate network traffic classification, machine learning techniques are growing in popularity [37]. Supervised machine learning models require labeled data to help train and validate the models. 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. We believe that the Layer 4.5 customization architecture can be used to facilitate gathering labeled traffic classification data for both encrypted and unencrypted traffic flows. We begin by describing the customization module to perform the network traffic classification and embed the classification information into the network traffic for collection.

The customization module can follow the same logic used for adding the application tag to DNS traffic in Section 3.4, but the module socket matching parameters are broadened to match multiple sockets. For traffic classification, we can extend the module to include both the PID and TGID of the application generating the traffic as well as a classification label for the type of traffic (e.g., web, chat, video). Note that not all packets will have a customization tag applied unless specifically designed to do so. When TCP is utilized, we suggest instead to tag only the first packet in the flow. This will simplify customization processing and not waste network bit space with unnecessary tags.

With the use of broader socket matching parameters it is likely that some tagged traffic will be destined for end hosts outside the controlled network. For this reason, an inverse module could be developed for the network router. For instance, we could develop a XDP program to inspect and remove the traffic classification tag from the packet prior to routing outside the network. Alternatively, the network middlebox collecting the traffic may be able to remove the classification tags during the collection process.

We leave the development of the described customization and inverse modules to future work. The use of these modules should be tested with previous machine learning models

for performing network traffic classification of both encrypted and unencrypted traffic to determine if the models can be improved. Additionally, it would be useful to determine how often labeled data should be collected for the network and used to validate or update the models to maintain the desired accuracy thresholds.

4.1.2 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. After some investigation, the operator determines that some applications, in particular encrypted applications, are unable to process incoming customized traffic and will terminate the connection after the error occurs. To understand why this error is occurring on encrypted flows, consider the example TLS flow of Figure 4.1 illustrating the use of the long-lived customization module from Subsection 3.2.2 applied to the server's TLS payload and the subsequent client TLS decryption error. For reference, the long-lived customization module will add a 32-byte tag for every 1000 bytes of transmitted data.

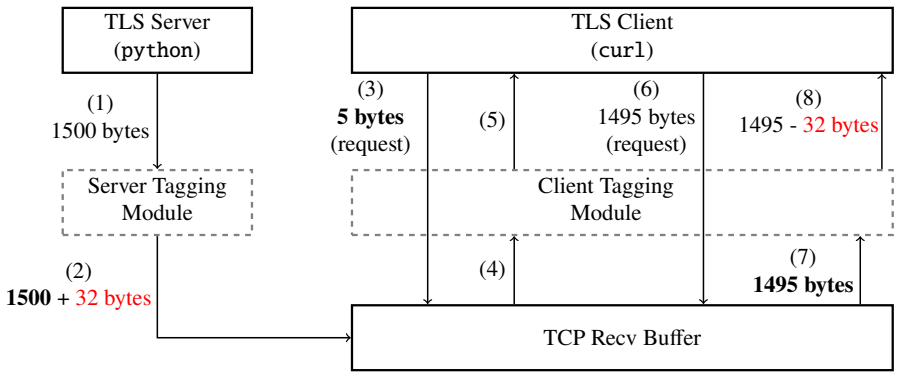


Figure 4.1. Example of client TLS processing failure when customized data is present within the TLS payload. Flow order indicated by numbers in ().

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 because the payload is over 1000 bytes. Layer 4.5 then delivers the customized message to the transport layer and the message is transmitted 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 in the first 5 bytes, the client customization module updates the position tracker and passes the bytes to the application (5). The `curl` application then requests the 1495 encrypted bytes from the transport layer (6), which will leave 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, and it does not request the additional missing bytes. Instead, `curl` attempts to decrypt the 1463 bytes and a decryption error is triggered, resulting in the TLS connection being terminated. This same behaviour was experienced in a parallel effort to evaluate network detection of data exfiltration that leveraged Layer 4.5 as the insertion point [40].

4.2 Design of Module Message Buffering

In this section update the design of the Layer 4.5 customization logic from Subsection 2.2.3 to overcome the customization errors experienced when processing customized encrypted flows. We then implement the new design into the Layer 4.5 prototype and describe in detail how the prototype achieves the design specifications.

4.2.1 Updated Layer 4.5 Tap and Customization Logic

First, note that the errors experienced when processing customized encrypted flows were only caused by receive side processing, which means we do not need to alter send side logic and, thus, we will focus on the receive message tapping and customization logic. Figure 4.2 illustrates the updated Layer 4.5 tapping and customization logic with buffering capability for processing customized application flows.

The application receive path processing is still triggered when the application makes a receive message call but now there may either be data buffered by the customization module

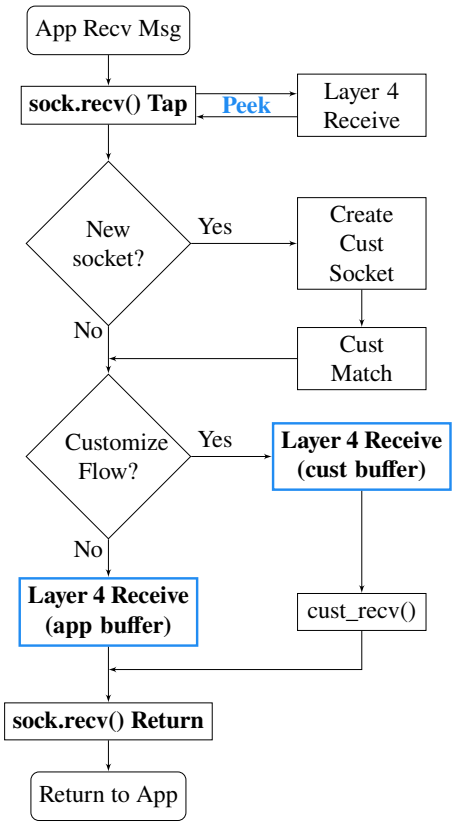


Figure 4.2. Layer 4.5 tap and customization logic with added buffering capability for application receive message processing.

and/or data present in the transport buffer. The Layer 4.5 socket tap intercepts the application receive message call and performs a peek operation to check the transport buffer and fill in any missing socket parameters for the customization lookup process. First, if the application flow is customized, then the receive message call is performed using a new customization buffer that can be larger than the application's allocated buffer to allow retrieving additional data necessary for proper customization processing. Since the customization buffer may be larger than the application's buffer, this also requires the customization module to buffer the additional data until the application is ready to process it. Second, if the application flow is not customized, then we use the application's buffer during the receive message call to prevent unnecessary memory copy operations.

4.2.2 Updated Layer 4.5 Prototype

Following the updated design for Layer 4.5 tapping and customization, Figure 4.3 illustrates the prototype customization processing logic when an application receives data.

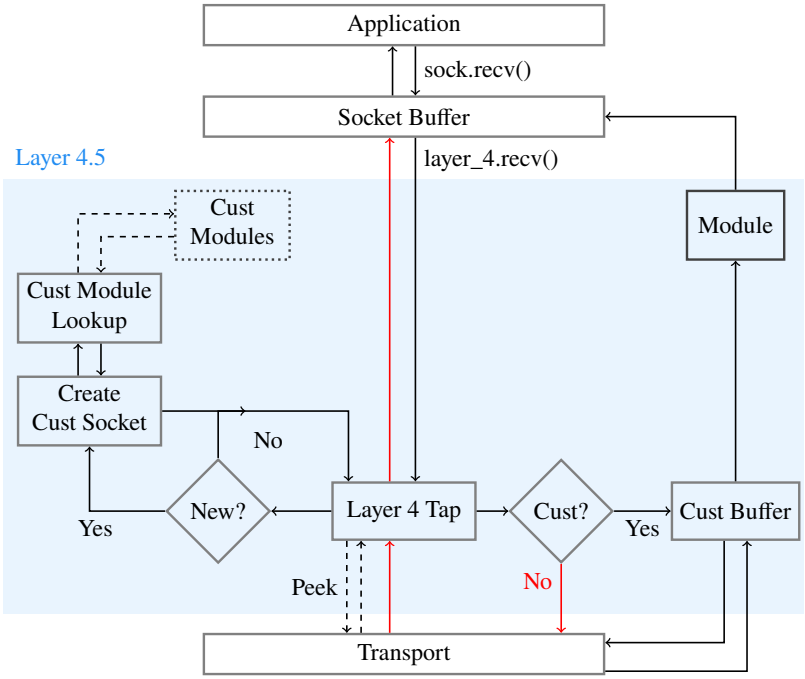


Figure 4.3. Layer 4.5 tapping and customization receive flow logic with buffering capability. The blue section represents the new Layer 4.5 logic introduced to the network stack. The red arrows indicate the path taken when no customization modules apply to the socket.

The customization flow starts after the socket tap intercepts the transport layer receive call. At this point, there are two main socket state possibilities to consider (i) a new socket and (ii) a customizable socket.

New Socket:

First, when a new socket arrives we can't guarantee that the socket 5-tuple parameters have been assigned because we support TCP and UDP sockets and applications are not forced to bind IP addresses or port numbers to the sockets. For this reason, we perform a preliminary layer 4 receive message call using the MSG_PEEK flag and a zero-length buffer. This step accomplishes filling in any missing socket parameters, but does not remove data from the

layer 4 buffer or waste processing time by copying data into the application's buffer. This operation also ensures data is available at layer 4 before proceeding to the customization logic. At this point, we can reliably conduct the same customization socket creation and module lookup process that was previously utilized (Subsection 3.2.1). The newly processed socket will be treated as a customizable socket for all future socket send and receive message calls.

After the new socket process finishes, the socket tap determines if the customizable socket has a matching customization module. If no customization module matches the socket, then the socket is placed in a non-customized state, the layer 4 receive message call is performed using the supplied application buffer, and the buffer is returned to the socket layer. If the socket has a matching customization module, then it is set for customization and we perform the layer 4 receive message call using the customization receive buffer instead of the application buffer. This new buffer, which may be larger than the application buffer, is then sent to the customization module for processing. The customization module is responsible for determining how many bytes will be transferred to the application message buffer prior to delivery to the application and must buffer any remaining bytes until a future application receive call is performed.

Customizable Socket:

When a customizable socket arrives, we need to determine if the socket has a customization module attached to it as quickly as possible because the module may have data buffered for the application. To determine the customization status, we attempt an early customization socket lookup, even though we can't guarantee all socket parameters have been assigned. If the customization socket lookup fails, then it is likely that the socket parameters have not been assigned. Therefore, we must perform a receive message call with the MSG_PEEK flag to fill in the missing parameters before we proceed with processing the customizable socket.

After the customization socket lookup succeeds, then we need to determine if the socket has a customization module attached or if it is a non-customized socket. If it is a non-customized socket, then we simply conduct the transport receive message call using the application's buffer. Otherwise, we are processing a customized socket and we need to determine if the

customization module has buffered data ready for the application by using the customized socket structure parameter for buffered data. If the customized socket structure indicates the module has buffered enough data to deliver to the application, then we skip the transport receive message call and instead serve data from the customization module. Skipping the receive message call to layer 4 prevents application delays which may result if the transport layer does not have any data ready for the application, but the application is in a blocking state waiting for more data. Since the customization module already processed the data the application desires, we can avoid potential delays and serve the request from the customization module. If the customization module does not have enough data to fill the application's buffer, then we proceed with the receive message call using the customization receive buffer.

4.3 Evaluation

In this section we evaluate the updated processing overhead now that we have introduced buffering capability into Layer 4.5. Since the updated customization logic utilizes a customization buffer that must be allocated, we expect the processing overhead to meet or exceed that of Subsection 3.2.3. We begin by describing the updated customization modules developed to account for the new buffering capability and finish with the processing overhead measurement results.

4.3.1 Revised Customization Modules for Overhead Testing

The new Layer 4.5 logic requires customization modules to buffer data when the customization receive buffer contains more data than the application requested, even if the customization module does not require buffering to properly customize the targeted application flow. For this reason, we updated the testing modules described in Subsection 3.2.2. Table 4.1 provides an overview of the updated testing modules.

The first tagging module targets short-lived flows that send only one IP packet worth of data for each transmission. This module is relatively simple and does not benefit from the new buffering capability. Thus, the updated version was simplified to not buffer any data, but this still increased the server module's *cust_recv* function LOC, since additional checks are required to handle various receive message values and avoid corrupting buffer memory.

Table 4.1. Updated customization module lines of code

	Client Module			Server Module		
Flow Type	*cust_send*	*cust_recv*	Total	*cust_send*	*cust_recv*	Total
Short-lived	15	2	90	2	40	115
Long-lived	2	90	170	60	2	140

The second tagging module targets long-lived flows that transfer large amounts of data. The long-lived customization module may benefit from the new buffering capability because the module can now process data in larger chunks. This is particularly useful towards the end of the long-lived flow where the receiver is expecting a known amount of data to be transferred (e.g., the remaining bytes of a requested file). When using the non-buffering logic, if the application requested the specific amount of final bytes expected, but these bytes contained at least one customization tag, the application will be forced to conduct an extra receive message call to retrieve the remaining final bytes. With the new buffering capability the customization receive buffer can retrieve all remaining bytes, remove any customization tags, and then return all requested bytes to the application. Unlike the previous short-lived module, the updated version uses buffering by requesting a customization receive buffer larger than the application buffer, which resulted in an increase to the client module's *cust_recv* function LOC.

4.3.2 Prototype Evaluation

We begin by repeating the DNS experiment using the updated short-lived customization module with a client side `dig` application and server side `dnsmasq` application. 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 4.4 illustrates the overhead of Layer 4.5 socket taps and the overhead of Layer 4.5 taps with the customization applied.

From the resulting boxplots, we observed a more significant Layer 4.5 tapping mean overhead that can be attributed to the two receive message and customization lookup calls performed by the `dig` client. Recall from Section 4.2, we chose to attempt a customization

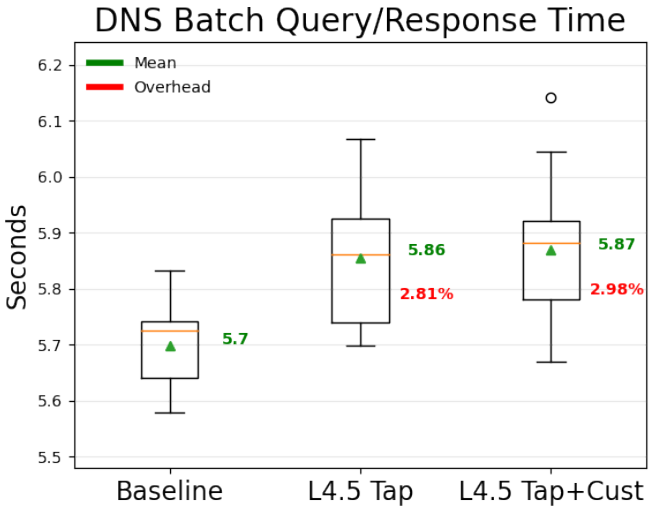


Figure 4.4. Layer 4.5 with buffering capability measured overhead of 1000 short-lived application flows.

lookup prior to the first receive message call to allow early detection of customization since we may be able to serve the requested bytes from the customization module instead of the transport layer. Since we are dealing with UDP sockets in this experiment, each client DNS request is a new socket and the early customization lookup will fail. While this receive message call did not previously add significant overhead, performing this request twice in the receive message path increases the chance of delays caused by the use of process locks that prevent multiple sockets from accessing the customization hash table at the same time.

When each DNS message is tagged, we see a less significant mean increase compared to Subsection 3.2.3. 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 requests 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 1000 receive buffer allocation and free operations, which would likely increase the batch customization overhead.

HTTP over TCP

The next experiment we repeat is the bulk file transfer using the updated long-lived customization module. To determine the customization receive buffer size that should be assigned to the customization module, we reviewed the `curl` client behaviour from the initial experiments. The client side `curl` application was observed to request a maximum of 102400 bytes when conducting a receive message call. Thus, to allow processing more data than previously allowed, the customization module was configured with a customization receive buffer of size 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 3.4 illustrates the model comparison of the overhead of Layer 4.5 socket taps and the overhead of Layer 4.5 taps with the customization applied.

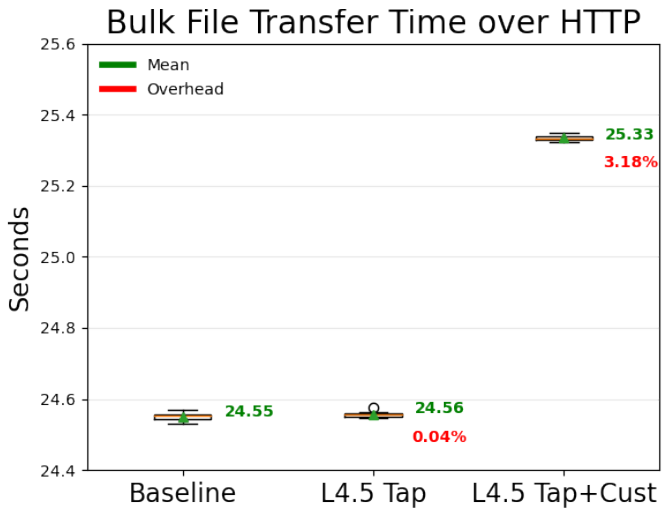


Figure 4.5. Layer 4.5 with buffering capability measured overhead of a single long-lived application flow.

From the boxplot, we see that the overhead experienced after adding the buffering capability did not differ from the initial results. Contrary to the batch experiment, the long-lived application flows benefit from the early customization lookup process and are not subjected

to a second lookup. There was also no significant difference in the amount of data processed during each receive message call since the `curl` application provides a large receive buffer and providing a larger buffer does not result in processing significantly more data each call.

HTTPS over TCP

The last flow we target is an encrypted bulk file transfer. The main goal of this experiment is 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. We utilize the same 3 GB Ubuntu image from the bulk file transfer experiment, but instead use a Hypertext Transport Protocol Secure (HTTPS) over TCP connection. The client still utilizes the `curl` application, but the server module is now attached to a `python` web server application configured to use TLS.

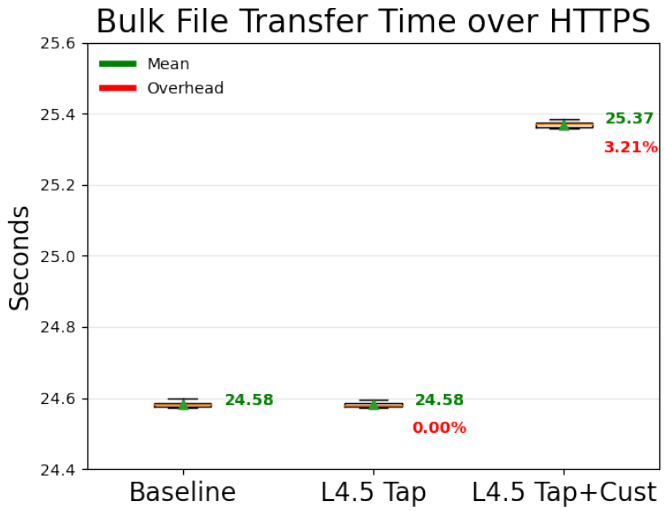


Figure 4.6. Layer 4.5 with buffering capability measured overhead of a single long-lived encrypted application flow.

From the boxplot, we see similar results to that of the unencrypted file transfer. The increased mean file transfer time can be attributed to the overhead of the TLS connection. The key takeaway from this experiment is the ability to customize both encrypted and unencrypted long-lived flows without resulting in unacceptable overhead.

4.4 Insights

The design and evaluation of the buffering capability for Layer 4.5 exposed some new customization challenges. In this section we will highlight the new insights we gained.

- **TLS Dialect:** If a middlebox device is being used to decrypt/encrypt TLS traffic between two devices, we may be able to detect the behaviour using a customization module. We know that the first Layer 4.5 design could not process customized TLS traffic, which means that if we sent a customized TLS message and a middlebox tried to process it without using special processing, then it is likely that the connection would be terminated. We explore this further in Section 6.2.

- **Multiprocessing:** Applications that utilize multithreading and/or multiprocessing may further complicate customization module design. If the customization module is required to maintain a state variable, much like the file-transfer tagging module used in Section 4.3, and multiple processes are using the module at the same time, then we must use mechanisms to synchronize customization processing such as spinlocks.

- **Socket Blocking:** Application sockets may set a NO_BLOCK flag, which allows the socket to wait until layer 4 has data to transfer into the socket buffer. The first buffering implementation did not properly account for this and attempted a socket receive call while holding a spinlock, which resulted in locking up the kernel. Additionally, if the customization module has already buffered all remaining data from the transport layer, an application call with the NO_BLOCK flag set may never return. Therefore, we redesigned the prototype to avoid scenarios that could result in unnecessarily holding locks.

4.5 Summary

In this chapter, we designed a new customization module buffering capability into Layer 4.5 to expand protocol customization capabilities to both encrypted and unencrypted flows. The update to Layer 4.5 was designed to work with all applications, but was targeted towards applications that perform strict receive message processing, such as those using TLS to encrypt messages. Note that we highlighted TLS applications in this chapter because of their common use, but the updated design also expands customization capabilities to applications

that follow similar strict receive message processing, such as `dnsmasq` when using TCP. The updated processing overhead experienced was comparable to that of the initial design for long-lived application flows, but the short-lived flows experienced a 2% increase in overhead for Layer 4.5 tapped connections. Additionally, the buffering capability required more complex customization modules to perform the same customizations possible without buffering.

CHAPTER 5:
Rotating Customizations in Wide Area Networks

Wide area networks present communication delay and synchronization challenges to network customization. First, geographically distant end hosts can have different communication delays (i.e., latencies) or network capacities (i.e., bandwidth), which results in the challenge of synchronizing deployed customization modules. The primary goal of this chapter is to evaluate Layer 4.5 customization use in a WAN to include methods for customization synchronization and "hot-swapping" customizations on an active socket without service interruption.

Second, Layer 4.5 customizations may be subjected to third-party middlebox interference as customized messages traverse the public internet infrastructure. However, we hypothesize that Layer 4.5 traffic will be unaffected by third-party middleboxes since they should not inspect packet data above the transport layer (i.e., layer 4). Therefore, the secondary goal of this chapter is to conduct a brief evaluation of third-party middlebox interference of customized DNS and HTTP application flows.

5.1 Motivation

Consider the WAN in Figure 5.1 with two Layer 4.5 capable networks spread over geographically distant locations. In this network, communications between the NCO and the internal servers is much faster than communications with the external hosts. As a result, the NCO can deploy customization modules to the internal network services much faster than to the external end hosts. Using this WAN as a reference, we now consider several operational scenarios likely to occur in a real-world network that is actively customizing application flows.

Immediate Customization Attachment

A network event occurs and triggers the automatic deployment of customization modules, one of which is to the DNS server deemed as a critical network service. Recall from Chapter 3 that our Layer 4.5 simplified customization attachment to apply only to new

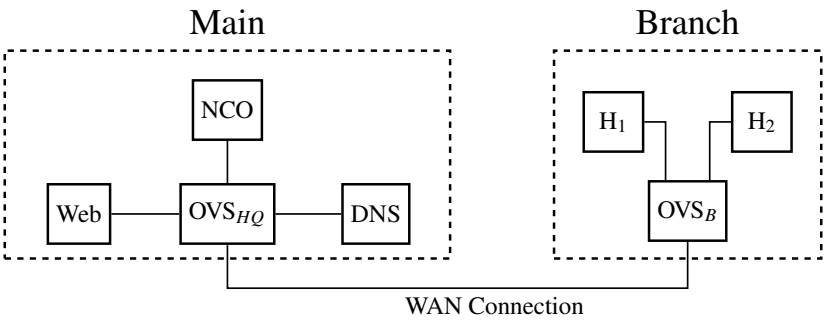


Figure 5.1. Layer 4.5 capable wide area network consisting of a main branch hosting network services and a remote host branch.

sockets. However, in this scenario it is not desirable to restart the DNS server because this may result in unacceptable network disruptions. Therefore, the Layer 4.5 DCA should support installing a customization module on the device and immediately allow matching the module to all sockets, not just new sockets.

Testing Customizations

A network operator would like to test a customization module without actually customizing the application flow. For instance, the operator may want to test that the module attaches to the correct sockets or test only one of the end devices to be customized. In Subsection 2.2.2, these types of modules would be considered as "monitoring" modules, but this classification does not fit the intended purpose of the customization modules being tested. Thus, we expand the Layer 4.5 design to enable deploying a customization module in a deactivated state and then activating it remotely from the NCO.

Customization Replacement for Future Sockets

A customization module has been deployed by the NCO and is actively being used on H_1 when communicating with the web server. After some time, the network operator wishes to deploy a new customization module to H_1 and the web server with the same matching parameters as the current modules, but the current modules should not be removed from any active flows to avoid service interruptions. Instead, the new modules should be applied to all future connections while the previously deployed modules remain attached to any current sockets, but the previous modules should also no longer be considered available for

future sockets. The current design of Layer 4.5 does not support this capability because the only way to remove a module is through the revoke functionality of the NCO and DCA. Additionally, Layer 4.5 does not permit multiple customization modules to be deployed with the same matching parameters since this would result in unpredictable customization attachment. Therefore, we will expand Layer 4.5 to enable the graceful replacement of customization modules.

Customization Replacement on Active Flows

A network operator wishes to deploy a new set of customization modules to H_1 and the web server, but now the operator intends to have the new modules replace the existing modules even if they have already attached to an active socket. This scenario is more complicated than the deprecation scenario because we must be concerned with customization synchronization issues between the two end devices. For instance, if H_1 changes to the new customization module before the web server, then this is likely to cause web server customization processing errors. Therefore, we leverage features from the previous scenarios along with additional new features to develop the capability to perform active flow customization.

5.2 Design of Module Hot-Swapping

In this section we will gradually expand the Layer 4.5 architecture capabilities by adding new features that will enable the NCO to replace the customization module attached to an active flow with a new customization module, without cycling the application. Figure 5.2 depicts the updated NCO distribution and continuous management functions to support such customization efforts.

5.2.1 Attaching to Active Flows

The Layer 4.5 customization architecture design from Chapter 2 did not explicitly dictate how newly installed customization modules should be applied. For the initial prototype, we simplified the process of customization attachment by only applying customization modules to new sockets. Thus, we need to expand the Layer 4.5 design to explicitly allow for the immediate attachment of customization modules if so desired. To accomplish this, we first update the NCO *deploy* function to include the ability to specify if the customization module being deployed should be applied to all sockets or just to new sockets. We allow this

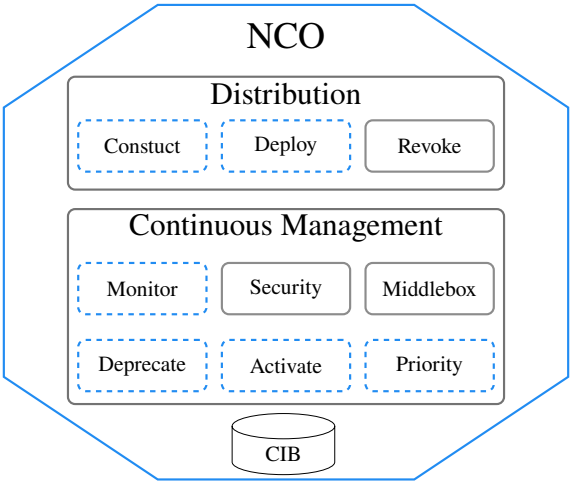


Figure 5.2. Layer 4.5 NCO consisting of updated "distribution" and "continuous management" functions to support customization module rotation. Updated or new functions represented with a dashed blue border.

distinction because there are still scenarios where the new module should only apply to new sockets and not affect existing sockets. Next, we update the DCA *install* handler to accept this new indicator and act accordingly when installing and registering the customization module.

5.2.2 Activating Customization Modules

The initial Layer 4.5 design did not consider the deployment of customization modules in an inactive state. However, it may be desirable to deploy such a module for the purpose of testing a customization prior to full deployment. Additionally, it may also be desirable to simply deactivate a module on an end device in lieu of revoking the module. Thus, to support the remote activation or deactivation of a deployed customization module, we expand the design of the Layer 4.5 NCO, DCA, and customization modules. Algorithm 3 provides the NCOs updated management logic to allow activating/deactivating a deployed customization module.

The NCO is updated to include a continuous management *activate* function that can be used to both activate and deactivate a deployed module. Additionally, the NCO *monitor* function is also updated to process activated flows separately from non-activated flows. This separate

Algorithm 3 NCO: Activate Management Logic

```
1:  while True do
2:    Activate Event: //module marked for activation or deactivation
3:      Invoke DCA activate handler for specified module
4:      Update module activation status in CIB
5:    Monitor Event: //end of a state_req_window
6:      Perform device state request
7:      if Module not activated then
8:        Update testing and state info in CIB
9:      else
10:       Update active_ts and other state info in CIB
11:     end if
12: end while
```

processing allows for non-activated modules to provide different state information that is more helpful for testing purposes and not relevant for modules that have been activated.

The DCA is also updated with an *activate* handler to allow toggling a customization module's activated state. The customization module's activated state is then used within the module when a *cust_send* or *cust_recv* call is performed by a customized flow. At the start of each of these functions, we now include a standardized activation check. If the module is not activated, the customization logic in the function is not performed. Otherwise, the module is in an activated state and the normal customization processing occurs.

5.2.3 Deprecating Customization Modules

Before we tackle customization rotation on an active flow, we first develop the ability to deprecate a deployed module, which will effectively remove the module from the list of available modules for future socket attachment. Deprecating a module does not affect active sockets with the deprecated module attached and is meant to allow a graceful transition to a new module without interfering with the previous customization modules in use. Algorithm 4 provides the general NCOs algorithm for deprecating and monitoring a customization module.

First, we update the NCOs continuous management functions and the DCAs handlers to include a new *deprecate* function/handler. When a module is marked for deprecation by

Algorithm 4 NCO: Deprecation Management Logic

```
1:  while True do
2:    Deprecate Event: //module marked for deprecation
3:      Invoke DCA deprecate handler for specified module
4:      Update module deprecate status in CIB
5:    Monitor Event: //end of a state_req_window
6:      Perform device state request
7:      if Deprecated module not active then
8:        Revoke deprecated module
9:        Update CIB
10:     end if
11:     Update active_ts and other state info in CIB
12: end while
```

the network operator, the *deprecate* function will utilize the NCO/DCA control channel to coordinate the deprecation of the specified module on the customized device. Once the DCA reports the modules deprecated status, the *deprecate* function updates the CIB to reflect the new status and allow the *monitor* function to track the module until ready for automatic revocation.

Since we must continue to monitor the deprecated module, we also update the continuous management *monitor* function to track deprecated modules that are deployed and active, but now in a deprecated state. To support the automatic revocation of deprecated modules that are no longer being used, the *monitor* function must now also track deprecated modules and the sockets they are attached to until the module is no longer actively attached to any sockets on the customized device. Once the deprecated module is no longer actively being used, the *monitor* function will invoke the distribution *revoke* function to completely remove the deprecated module from the customized device.

5.2.4 Attaching Multiple Customization Modules

At this point, the Layer 4.5 customization architecture has been expanded to support deprecating, activating, and immediately attaching customization modules. There is one final feature required to support rotating from an active customization module to a new customization module on the same active socket.

The initial design allowed for a single customization module to be attached to a socket at a time. This design choice ensured each end device had the same customization module attached, and that multiple modules would not be customizing the same flow, possibly in different orders. We now relax this condition, but only slightly, to support transitioning from one customization module to the next. In this new design, we now allow an array of customization modules to be attached to a single socket, but we only allow one customization module to actively customize the socket. This design decision leads to two sub-requirements:

1. Customization modules need a priority assignment to determine their position in the attached list.
2. Customization modules need a mechanism to determine if they should process the flow or skip processing and allow the next module to customize the flow.

First, to implement customization module priority, we update the NCO *construct* function to include setting the module priority at build time, much like the function already does for setting the *mod_id*. Next, we update the NCO continuous management functions to include a *priority* function to update a module's priority level after deployment. After we update the NCO to support setting a module's priority, we then update the DCA to include a *priority* handler. This handler is called by the NCO and is responsible for interfacing with the customization module to change the priority level.

Second, the Layer 4.5 customization logic is updated to allow a customization list to be attached to an active socket. This customization list is sorted by priority level before attaching to the socket and if a customization module priority is updated, then the list is re-sorted to reflect the update. The attached customization list can hold an arbitrary amount of customization modules, but only one customization module is allowed to customize the socket. After the first customization module in the list customizes the socket, no other customization modules in the list will be applied. Therefore, adjusting the customization priority is the primary method for determining which customization module will be applied to the socket first.

Finally, to allow module rotation on an active socket, we need to allow multiple modules to attempt customization during a synchronization time window. To accomplish this, we require customization modules to use an identifying feature during customization such as a

customization tag or ID. When the higher priority customization module attempts to process the flow, if this feature is not detected, then the module must skip customization processing and signal that the next module in the list should be applied to the socket instead. Note that if a customization module does not have an identifying feature, then it can not support this rotating scheme.

NCO Customization Rotation Process

The NCO must orchestrate the rotation of customization modules to ensure a successful transition from one module to the next without service interruption. Using the new features added to the NCO, DCA, and customization modules, we now define one possible process (Figure 5.3) to rotate customizations on an active socket without destroying the connection.

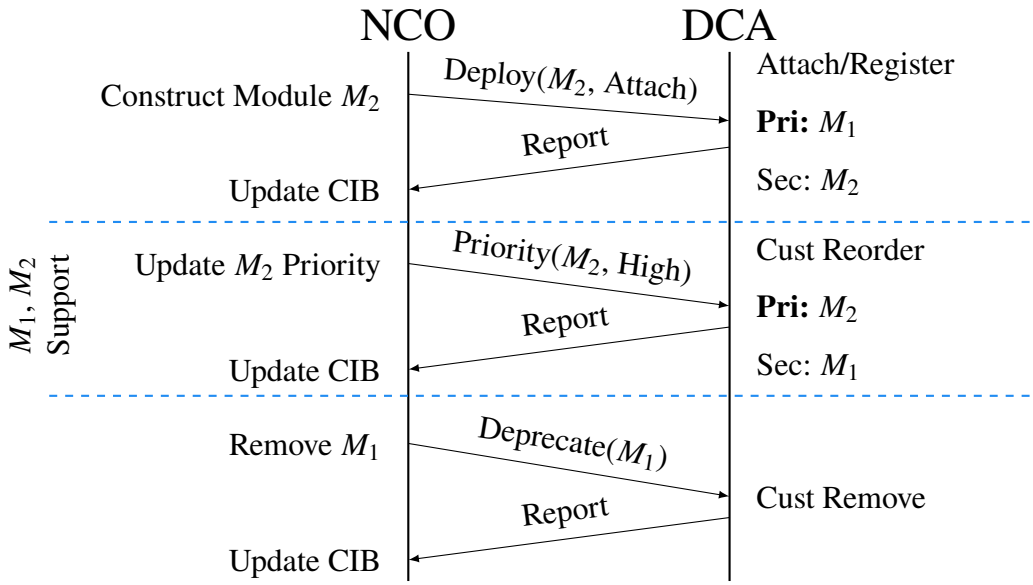


Figure 5.3. General NCO customization rotation process

We make the following assumptions in the customization rotation process:

1. The customization module to be replaced (M_1) was previously deployed to each device
2. M_1 has a high priority level and is in an activated, non-deprecated state
3. The replacement customization module (M_2) has a lower priority
4. M_2 is deployed in an activated state

The process of Figure 5.3 can apply to a single customized device or to multiple customized devices. However, when rotating customizations on multiple devices, the steps must be performed in parallel for each customized device. For example, deploying M_2 should be accomplished for each device prior to updating M_2's priority on each device. Additionally, we prioritize customization clients over servers when choosing the device order to perform the rotation steps. For clarity, we define device roles using the following definitions:

1. *Customization Client*: The end device receiving customized traffic
2. *Customization Server*: The end device sending customized traffic

We now describe the customization rotation process in more detail. First, we deploy M_2 to the client and server with a lower priority setting than M_1 and the immediate attachment flag set. Now the client supports processing flows using both customization modules, but the server is only sending customizing flows using M_1 because it has a higher priority level. After the CIB has been updated to reflect the deployment of M_2, we update the client's M_2 customization priority to cause customization module re-ordering. Now the server should still be using M_1 to customize flows sent to the client, but the client will attempt to process the customized flow using M_2 because it now has a higher priority on the client. However, M_2 will fail to identify the correct customization feature, which results in the transition to M_1 to process the flow. After the CIB is updated to reflect the client's M_2 priority change, we update the server's M_2 customization priority to cause customization module re-ordering. Now the client and server are both using M_2 as the primary customization module, which means it is safe to deprecate or revoke M_1 from each device without causing customization processing errors.

5.3 Evaluation

We begin by evaluating the features developed to achieve our primary goal of customization synchronization and active flow customization rotation. The features developed in this chapter were evaluated in a WAN testbed, Figure 5.4, leveraging the GENI [29] network. The WAN testbed presented uncontrollable delays between the NCO and each DCA, which allowed testing module rotation and the necessary new features to support it in a more real-world operational network environment.

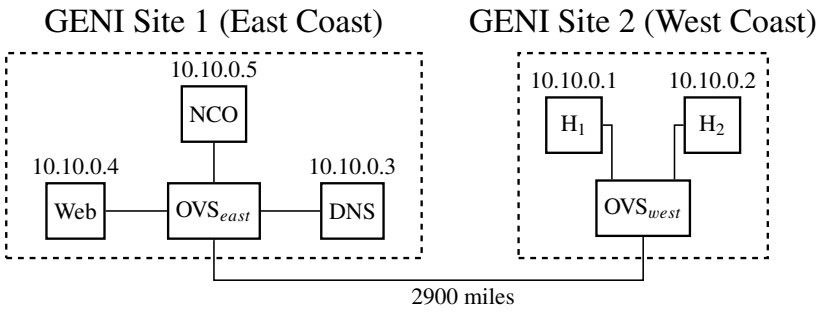


Figure 5.4. Layer 4.5 capable wide area network testbed.

We used the GENI networking environment to set up two Local Area Networks (LANs) at two different universities. The East Coast LAN was established at Old Dominion University with the other located approximately 2900 miles away at the University of Washington. Each LAN was connected using a "stitched" connection, which provided a layer 2 tunnel between each switch. The tunnel allowed testing customization deployment and use without interference from third-party middlebox devices. Note that prior to starting each experiment, all hosts and servers in the WAN except for the NCO were configured with Layer 4.5 capability. Additionally, each NCO/DCA control channel was established with each device DCA awaiting commands from the NCO.

To evaluate the success or failure of each feature, we utilized the Layer 4.5 customization logs and packet captures of all customized traffic. The Layer 4.5 customization logs allowed us to verify deployed customization modules were attached to the appropriate applications, the modules were in the correct state, and that no unexpected errors were occurring. The packet captures of the customized application flows were used for one of two purposes. First, we verified the customized flows matched to the events logged within the Layer 4.5 customization logs. Second, we measured the throughput of the customized flows to show the customization module was actively customizing the flows.

After evaluating the new features, we created a new testbed consisting of one local Layer 4.5 client and 12 different Layer 4.5 GENI nodes hosting public-accessible network services. Using this new testbed, we focused on our secondary goal of evaluating third-party middlebox interference of Layer 4.5 customized flows.

5.3.1 Immediate Attach Testing:

In order to test the ability to attach a customization module to an active socket, we developed a new customization module to match the DNS server's socket parameters. We chose to customize the DNS server because the `dnsmasq` application uses a single socket to process incoming DNS requests and the socket is allocated when the application starts. The new customization module was configured to customize the receive message path and would inspect each incoming DNS request for the presence of a customization signature at the front of the request. If the customization signature was not present, then the customization module signaled to the DCA that the message should be dropped and alert the application of a message processing error without causing an application error. Figure 5.5 provides the Layer 4.5 customization log[4] from the experiment with a corresponding Wireshark capture overlay added to the log.

We began the experiment by starting the `dnsmasq` application on the DNS server (PID=5898) without a customization applied. We then configured H_1 to repeatedly perform DNS requests using the `dig` application to the server at five second intervals with a maximum attempts setting of two and a five-second timeout value. Each DNS request followed the format: `www.test_{batch number}{request number}.com` to allow correlating requests to the Layer 4.5 customization log. Next, we added the DNS server's customization module to the NCO construction and deployment queue. After the DNS server module was installed (1) and loaded onto the `dnsmasq` socket (2), we waited approximately 30 seconds to allow several DNS requests to be received and dropped by the customization module (3)-(6). We then configured the NCO to revoke the customization module from the DNS server, which resulted in the customization module being unloaded (7). At this point new DNS requests were permitted to reach the `dnsmasq` application again and processed accordingly.

Based on the Layer 4.5 logs and corresponding packet capture, we conclude that this experiment has successfully demonstrated the ability to immediately attach a customization module to an active socket.

[4]Layer 4.5 customization logs have been edited to remove unnecessary information and improve readability.

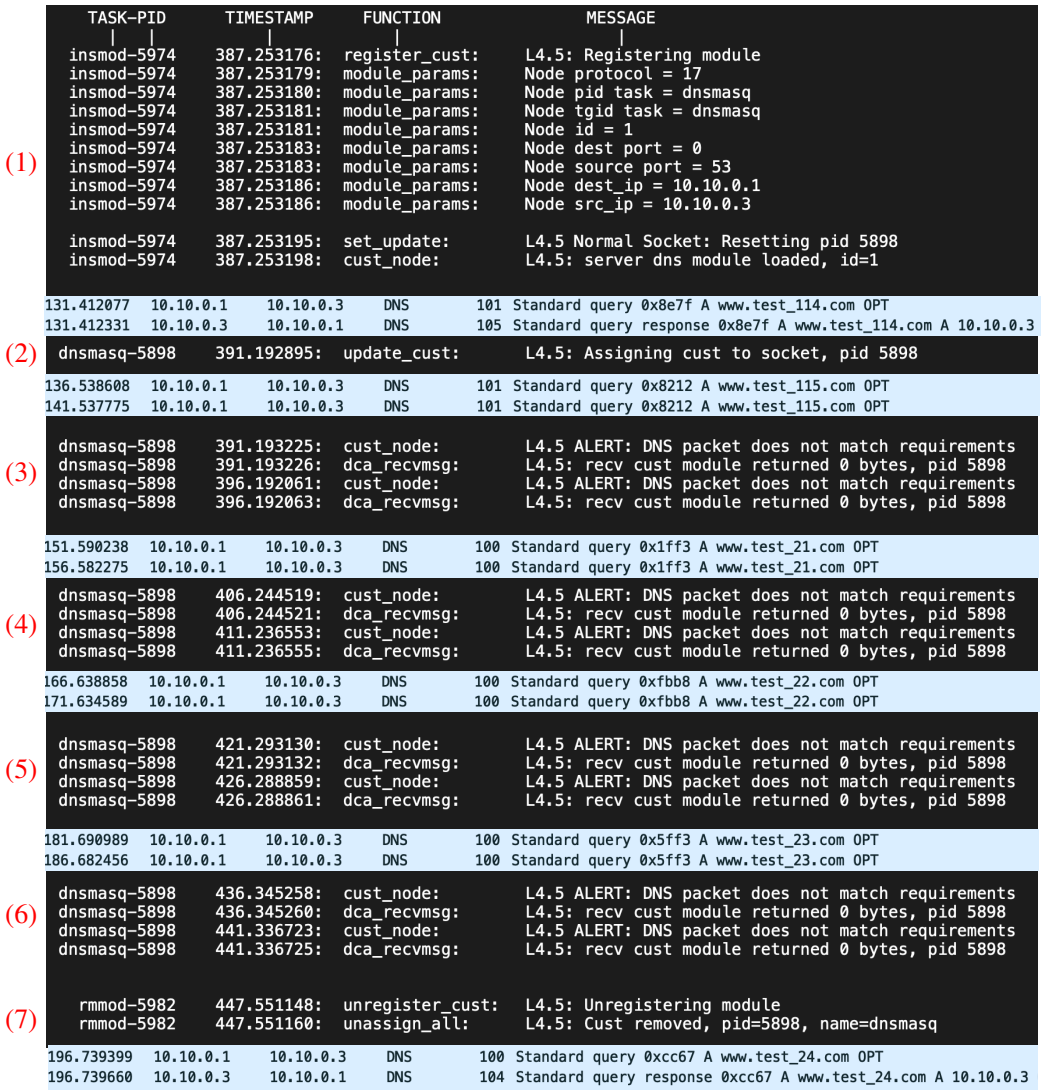


Figure 5.5. Layer 4.5 customization immediate attachment log and corresponding Wireshark overlay.

5.3.2 Activate Testing:

To evaluate the new customization module activation feature, we first deployed a deactivated customization module to the web server. We utilized a new customization that performed rate-limiting on a per-message basis by adding a 100-msec delay to each send message call. This customization module was designed to cause a noticeable packet transmission pattern and significant drop in performance when transferring a large file between the web server and

host. Instead of the previously used 3 GB Ubuntu image, we transferred the Layer 4.5 kernel module file, which is approximately 3.7 MB. Transferring the smaller file with the new customization module allowed for a faster file transfer while still testing the customization module activation. We collected Layer 4.5 customization logs and traffic on the web server throughout the experiment and compared deactivated and activated customization module file transfers in Figure 5.6.

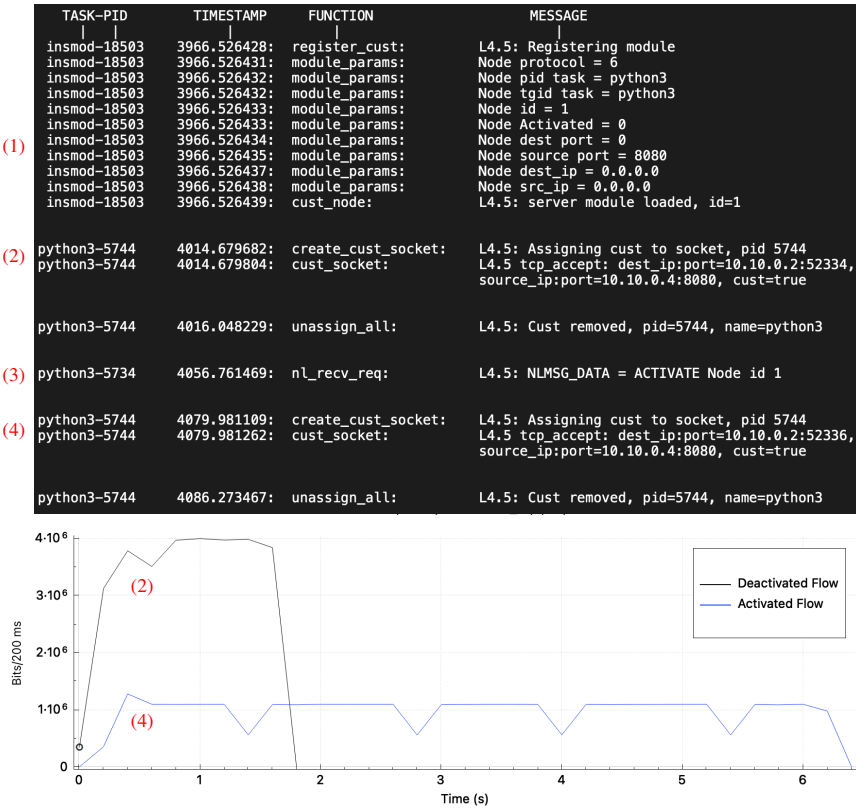


Figure 5.6. Layer 4.5 customization activation log and corresponding throughput graph. Throughput measurements for each flow are time-shifted to start at time zero.

We began the experiment by deploying the 100-msec rate-limiting customization module to the web server. The Layer 4.5 log shows the customization module was registered to match the `python` web server parameters and that the module was in a deactivated state on the web server (1). Next, we performed a file transfer (2) and verified the customization module was attached to the socket, but also that it was not actively customizing the flow. After the

file transfer completed, the NCO issued a command to the web server's DCA to activate the customization module (3). Last, we performed the file transfer again (4) and, as expected, the 100-msec send message delay created a clear pattern in the throughput measurement.

Based on the Layer 4.5 logs and throughput graph, we conclude that this experiment has successfully demonstrated the ability to attach a customization module in a deactivated state and then activate the customization module when ready to apply the customization.

5.3.3 Deprecate Testing:

When we deprecate a customization module it must remain attached to any active sockets that it was previously customizing, but any new sockets undergoing the customization lookup process should not be matched against the deprecated module. Therefore, to test customization module deprecation we performed a file transfer from H_1 using the rate-limiting module with a longer delay of 1000-msec, and after the file transfer began, we configured the NCO to deprecate the module. We continued to monitor the throughput after module deprecation because if the customization module was removed from the socket, then the throughput would drastically improve. Additionally, after the customization module wass deprecated, we configured H_2 to also perform a file transfer to verify that the deprecated customization module was not attached to the new flow. Figure 5.7 shows the web server's Layer 4.5 customization log and the throughput of each flow.

First, we can clearly see that the throughput of H_1 maintained the same pattern throughout the entire download. This throughput pattern was the result of the customization module inserting a 1000-msec send message delay for each application send message call performed. Even after the customization module was deprecated (3) approximately 30 seconds into the connection, the customization remained active for the remainder of the file transfer. Second, since we were running a simple `python` server without multiprocessing, the H_2 file transfer was delayed until the H_1 transfer finishes. When the H_2 application socket was created on the web server (4), we observed that the connection was not customized with the 1000-msec customization module even though it was still loaded on the web server.

Based on the Layer 4.5 logs and throughput graph, we conclude that this experiment has successfully demonstrated the ability to deprecate a customization module without affecting current flows or applying the customization module to future application flows.

```
    TASK-PID      TIMESTAMP    FUNCTION                  MESSAGE
       |   |          |            |                        |
(1) insmod-22597   281.487102:  register_cust:          L4.5: Registering module
    insmod-22597   281.487105:  module_params:          Node protocol = 6
    insmod-22597   281.487106:  module_params:          Node pid task = python3
    insmod-22597   281.487106:  module_params:          Node tgid task = python3
    insmod-22597   281.487107:  module_params:          Node id = 1
    insmod-22597   281.487107:  module_params:          Node Activated = 1
    insmod-22597   281.487108:  module_params:          Node dest port = 0
    insmod-22597   281.487109:  module_params:          Node source port = 8080
    insmod-22597   281.487112:  module_params:          Node dest_ip = 0.0.0.0
    insmod-22597   281.487112:  module_params:          Node src_ip = 0.0.0.0
    insmod-22597   281.487113:  register_cust:          L4.5: Registration time: 1659032476
    insmod-22597   281.487114:  cust_node:              L4.5: server module loaded, id=1

(2) python3-22581  313.454155:  create_cust_socket:    L4.5: Assigning cust to socket, pid 22581
    python3-22581  313.454276:  cust_socket:           L4.5 tcp_accept: dest_ip:port=10.10.0.1:60924,
                                                       source_ip:port=10.10.0.4:8080, cust=true

(3) python3-22576  341.681445:  nl_recv_req:          L4.5: NLMSG_DATA = DEPRECATE Node id 1
    python3-22576  341.681451:  deprecate_cust:       L4.5: Removing module from use by new sockets
    python3-22576  341.681463:  deprecate_cust:       L4.5: Deprecated time 1659032536

    python3-22581  372.843220:  unassign_all:         L4.5: Cust removed, pid=22581, name=python3

(4) python3-22581  372.843382:  new_inet_csk_accept: L4.5: Customization skipped for pid 22581
    python3-22581  372.843386:  cust_socket:          L4.5 tcp_accept: dest_ip:port=10.10.0.2:52404,
                                                       source_ip:port=10.10.0.4:8080, cust=false
```

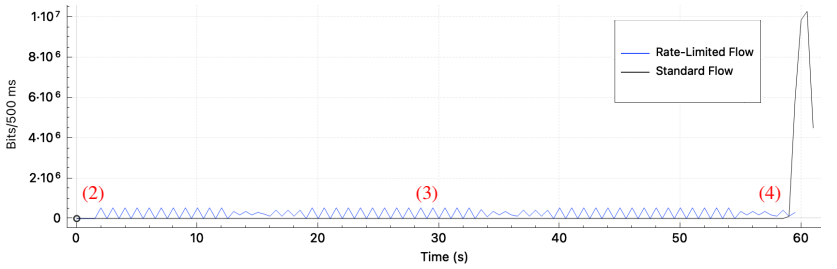


Figure 5.7. Layer 4.5 customization deprecation log and corresponding throughput graph.

5.3.4 Rotation Testing:

To test customization rotation on active sockets we conducted two different experiments using the process from Figure 5.3. First, we rotated a customization module applied to a single device. The goal of this first test was to validate customization rotation when we do not have to account for customization synchronization between a pair of devices. Second, we performed customization rotation using a pair of devices, which means each device must have been capable of using the same customization module to prevent processing errors. After completing the successful rotation of customization modules while maintaining customization synchronization we completed our evaluation towards the primary goal of this chapter.

Single Device Customization Rotation

In our first experiment, we customized the web server using the same rate-limiting 100-msec and 1000-msec customization modules from the previous experiments. We followed the process from Figure 5.3 with the 1000-msec customization module as M_1 and the 100-msec module as M_2. The Layer 4.5 customization log and corresponding throughput measurement for the experiment are shown in Figure 5.8.

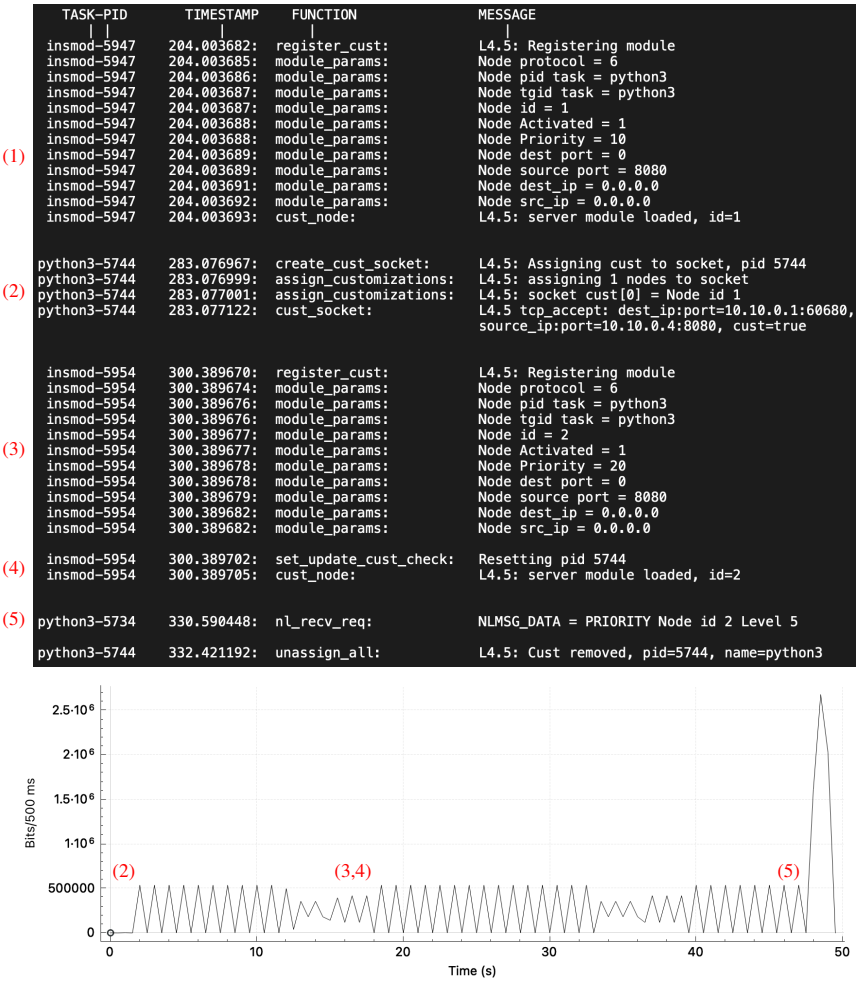


Figure 5.8. Single device Layer 4.5 customization rotation log and corresponding throughput graph.

We began by deploying the 1000-msec customization module with a priority value of 10 to the web server. After the customization module was registered on the web server (1), we initiated a file transfer from H_1 (2). When the file transfer was initiated, we observed in the

customization log that the module was attached to the `python` TCP socket (2), which can also be seen in the throughput measurement by the delayed start. While the file transfer was in progress, we deployed the faster 100-msec customization module with a lower priority level of 20 and the immediate attachment flag set to ensure the module was attached to the active `python` flow. At (3) and (4), the 100-msec customization module was registered on the device and attached to the same `python` socket, but as expected there was not a change in the throughput because the 100-msec customization module had a lower priority value. Last, at (5) the 100-msec customization module's priority was adjusted to a higher value than the 1000-msec module, which resulted in customization module re-ordering and the throughput of the file transfer to significantly improve.

This experiment confirms that we are able to rotate from one customization to the next on a single device. However, we expect that rotating customizations on a single device will be far less common than the need to rotate and synchronize customizations on multiple devices.

Multiple Device Customization Rotation

In our second experiment, we adapted the rotation process in Figure 5.9 to account for two DCAs: (i) the DNS server and (ii) H_1. For the purposes of rotation priority, the DNS server was defined as the customization client and H_1 was defined as the customization server.

Before we started the experiment, we needed two different customization modules with distinct identifying features to deploy to both the DNS server and to H_1. First, we used the same DNS tagging module from Subsection 3.2.3 as M_1 that would insert an application tag to the front of each DNS request. Second, we developed a new DNS customization module as M_2 that compressed the DNS request by removing "unnecessary" bits, much like what was done in [41]. Using these two drastically different customization modules, each module would be able to easily determine if the message being processed matches the expected customization. If customization processing did not match, then the customization module would signal Layer 4.5 to try the next customization module in the attached list. Figure 5.10 illustrates the results of the experiment using the Layer 4.5 customization log with embedded corresponding Wireshark packet capture.

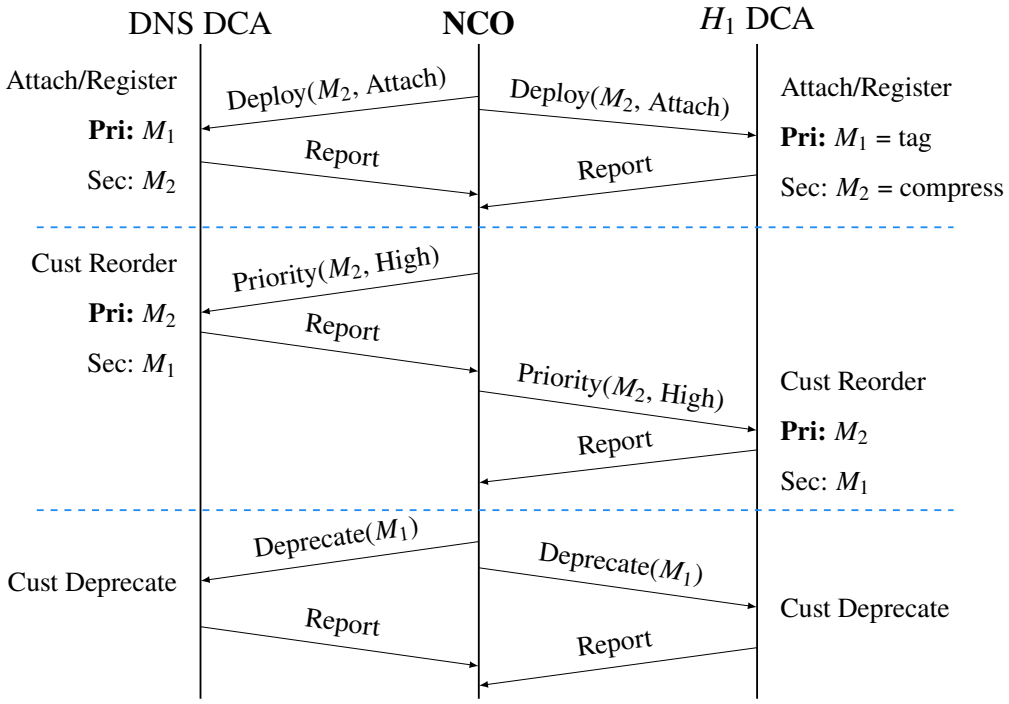


Figure 5.9. NCO customization rotation for two devices.

We began the experiment by starting the `dnsmasq` application (PID=6072) on the DNS server without any customization modules applied. We then deployed the tagging customization module with a priority level of 10 to the DNS server and to H_1. After the customization module was registered on the DNS server (1), we configured H_1 to repeatedly perform DNS requests to the server at five second intervals. Again, each DNS request followed the format: `www.test_{batch number}{request number}.com` to allow correlating requests to the Layer 4.5 customization log.

When H_1 initiated the first DNS request, we observed in the server's log that the module was attached to the `dnsmasq` socket (2). The embedded Wireshark capture (3) shows the front customization application tag for test_11 was processed by the server properly using the tagging customization module. Next, we deployed the DNS request compression module with a lower priority level of 20 to the DNS server and H_1. After the new customization module was registered (4), H_1 and the server were still using the front tagging module because the compression module was attached with a lower priority level.

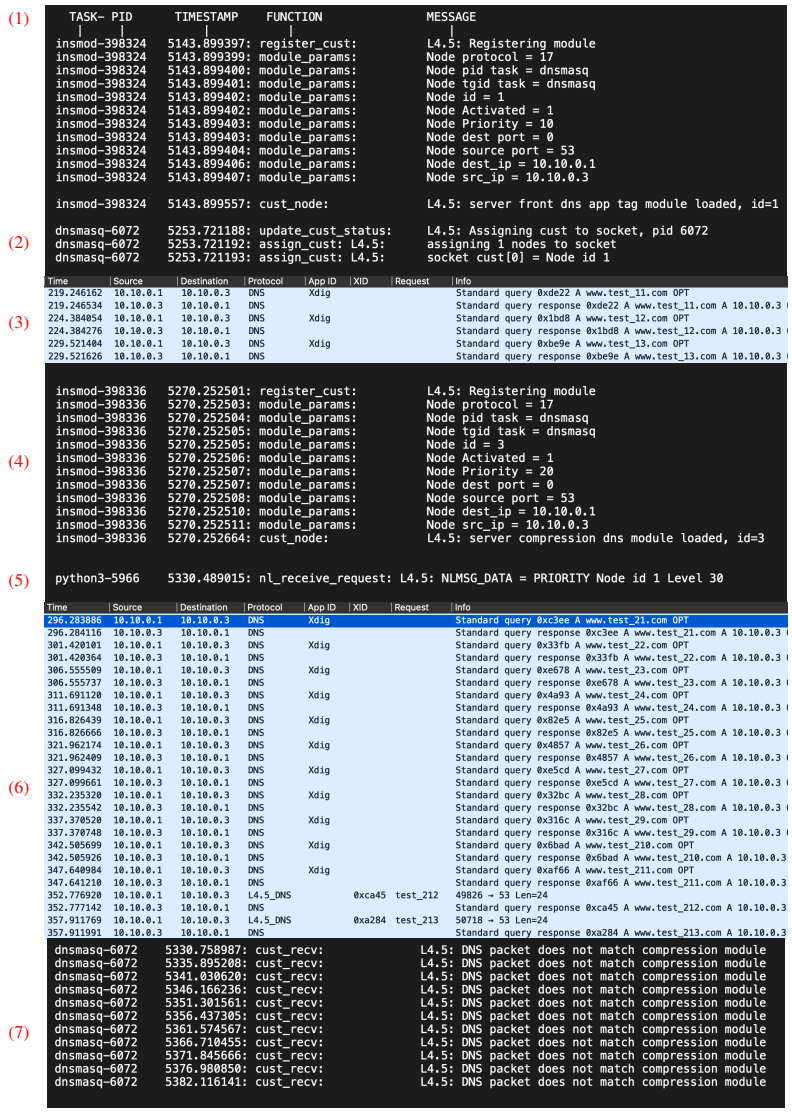


Figure 5.10. Multiple device Layer 4.5 customization rotation log and corresponding Wireshark packet overlay. A customization inverse module to interpret customized flows was utilized with Wireshark.

Next, we initiated the customization rotation from the NCO by changing the priority of the DNS server's tagging customization to 30, which was lower than the compression module. Then at (5), the priority of the tagging module on the server was changed to cause the compression module to be the primary customization module. On the DNS server, the next 11 incoming DNS requests continued to use the tagging customization

(6), but the compression module identified the customization mismatch and signaled for the next customization module to be applied (7). We completed the customization rotation from the NCO by changing the priority of H_1's tagging customization to a lower value, which resulted in both devices using the same primary customization module and no more customization errors generated in the server's log as evidenced by request test_212 being processed correctly by the DNS server.

Based on the Layer 4.5 logs and corresponding packet capture, we conclude that this experiment has successfully demonstrated the ability to rotate from one customization to another on a pair of end devices, while maintaining customization synchronization on active sockets.

5.3.5 Third-Party Middlebox Interference

Layer 4.5 customizations may not always traverse network tunnels or remain within the controlled network, which puts the customization at risk of third-party middlebox interference. However, we hypothesize that Layer 4.5 customized flows will not be affected by third-party middlebox interference because middlebox devices in the open internet primarily process data at or below layer 4. Therefore, to test how the open internet could interfere with Layer 4.5 customized application flows, we built a new experimental testbed utilizing GENI resources from various locations within the United States.

The new testbed utilized a local client Layer 4.5 VM located in Monterey, CA and 12 Layer 4.5 servers with publicly routable IP addresses within the GENI network. Using the new testbed, we conducted customized DNS requests and customized HTTP over TCP file transfers. Note that we did not perform testing over encrypted flows since middlebox interference is unlikely as deep packet inspection tools are unable to decrypt application data unless they are configured to have the TLS session keys. We varied the customizations utilizing different customization schemes in an effort to identify if certain schemes experienced interference while others were permitted. Finally, we determined that a customization was not subjected to middlebox interference if the customized messages arrived at the end-point device and customization processing was successful.

DNS Batch

There are multiple customizations possible for DNS requests. For reference, Figure 5.11 provides the structure of a DNS request. To increase the likelihood for middlebox interference, we targeted several places for customization including the DNS header, queries section holding the domain name being requested, and the end of the request.

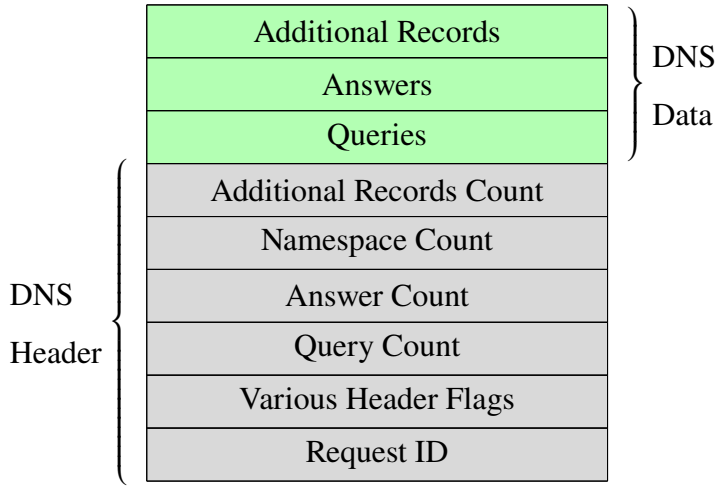


Figure 5.11. DNS header with data fields.

The following customization schemes were used:

1. **Base:** No customization is applied
2. **Front:** 32-byte tag is inserted before the request ID of each DNS request
3. **Middle:** 32-byte tag is inserted between the DNS header and the DNS data sections
4. **End:** 32-byte tag is added to the end of each DNS request after the additional records
5. **Compress:** The DNS request is modified to only include the request ID and the fully qualified domain name from the queries field, similar to the method from [41]

We started with a baseline test without any customization applied to first determine if the DNS server within the GENI network could receive DNS requests at all. We then proceeded with each customization scheme to determine if each could reach the DNS server, even if previous tests failed to reach the server due to interference. Table 5.1 lists the location

of each GENI node hosting a Layer 4.5 DNS server and the results of testing the server against each customization scheme. Each experiment was repeated 5 times and consisted of 5 different DNS requests.

Table 5.1. Layer 4.5 DNS customization middlebox interference results

InstaGENI Node	Base	Front	Middle	End	Compress
Clemson	PASS	**FAIL**	**FAIL**	PASS	**FAIL**
Colorado	PASS	PASS	PASS	PASS	PASS
Cornell	PASS	PASS	PASS	PASS	PASS
Illinois	PASS	PASS	PASS	PASS	PASS
Missouri	PASS	PASS	PASS	PASS	PASS
Northwestern	PASS	PASS	PASS	PASS	PASS
NYU	PASS	PASS	PASS	PASS	PASS
Ohio State Univ.	PASS	PASS	PASS	PASS	PASS
Univ. of Hawaii	**FAIL**	**FAIL**	**FAIL**	**FAIL**	**FAIL**
Univ. of Texas	PASS	PASS	PASS	PASS	PASS
Univ. of Washington	PASS	PASS	PASS	PASS	PASS
Virginia Tech	PASS	PASS	PASS	PASS	PASS

As seen in Table 5.1, the University of Hawaii node was the only node that blocked all DNS traffic when using UDP and port 53. To determine if the failed requests were due to third-party middlebox interference we conducted several additional configurations and tests. First, we changed `dnsmasq` to use port 5353 instead of port 53 and repeated each test. Using the non-standard port, we were able to test each customization scheme without experiencing interference. Second, we switched the transport protocol for DNS from UDP to TCP. Using TCP, we confirmed that the Hawaii node would respond to base DNS requests without filtering. Third, we logged into the Hawaii node and conducted a base DNS request to the Clemson node. Again, this request was filtered when using UDP, but not when using TCP. Last, we utilized the `iperf3` utility to test port 53 communications using TCP and UDP. As expected, we were able to communicate on port 53 using TCP but using UDP resulted in no traffic being permitted. These results indicate that the middlebox interference was most likely at the university's gateway router filtering all UDP traffic on port 53.

The interference experienced by the Clemson node differs from that of the Hawaii node. The Hawaii node filtered all UDP traffic on port 53. However, the Clemson node only filtered malformed DNS traffic as seen by the failed customizations. Again, we used the `iperf3` tool to test UDP communications on port 53 and found that the UDP traffic was filtered. These experiments indicate the presence of a deep packet inspection middlebox (e.g., a network firewall) filtering malformed DNS traffic.

Furthermore, the Clemson node was the only node to allow the base DNS request but also block DNS customization attempts, with the exception being the end DNS customization module. Since the end module places the customization tag after the Additional Records section of the DNS request, the tag is interpreted as part of that section. The Additional Records section can hold Extension Mechanisms for DNS (EDNS(0)) messages, which can have a variety of values and may not be recognized by the DNS server [42]. The tag did not trigger the previously experienced middlebox interference because DNS servers are typically configured to bypass unknown portions of EDNS(0) messages to allow for backward-compatibility.

Based on the results of DNS testing we do not suspect the middlebox interference experienced was from third-party middlebox devices on the open internet. Instead, we believe the interference to be from devices internal to the GENI network. In a WAN environment, these devices would be under the control of the same enterprise network and would therefore be supported by the Layer 4.5 architecture.

DNS End-Customization Expansion

The end customization scheme, due to its unique interpretation, does not require the server to have a corresponding Layer 4.5 module. This behaviour made us curious as to how long the tag could be before a non-customized server would no longer process the request. To test this maximum size, we altered the end customization module to repeatedly insert the customization tag based on a parameter set when the module is loaded. This parameter was gradually increased until the DNS server no longer responded to the request. Since each of the GENI nodes tested in Table 5.1 used the `dnsmasq` application, we assumed each node would have the same limit. All nodes that accepted the end customization, except for the Clemson node, had a limit of 4096 DNS payload bytes. The reason for this maximum size is

the result of a flag set by the transport layer when the application does not provide a buffer large enough to hold all data present in the transport buffer. If a DNS request is received with a larger size than allocated in the application buffer, the "truncated" message flag is set and the request is discarded by `dnsmasq`. Interestingly, the Clemson node had a maximum size of 1514 DNS payload bytes, which is not a power of 2 as would be expected.

We then tested the end customization against the Google public primary (8.8.8.8) and secondary (8.8.4.4) DNS servers to ensure this was not a property of `dnsmasq` and our experimental setup. We did not want to present a perceived attack against public DNS servers, so we tested the end customization against the Google servers until we determined the maximum tag size and did not repeat the experimentation beyond this point. The primary and secondary Google DNS servers each accepted customized queries with a maximum DNS payload of 512 bytes, responding with the same address information as if the request was not customized. This payload size complies with the maximum DNS over UDP size when not using the EDNS(0) extension [42].

HTTP over TCP File Transfer

After a successful DNS request, a web request is typically conducted. For this reason, we also experimented with customized HTTP requests going to the same GENI nodes used in the previous middlebox DNS experiment. Additionally, since file transfers are much longer than DNS requests we utilized different customization schemes that targeted the entire file transfer process. The following customization schemes were used:

1. **Base:** No customization is applied
2. **100x:** 32-byte tag is applied every 3200 bytes
3. **10x:** 32-byte tag is applied every 320 bytes
4. **1x:** 32-byte tag is applied every 32 bytes, a one-to-one ratio
5. **0.5x:** 32-byte tag is applied every 16 bytes, which results in twice as many tag bytes being transferred than file bytes

We chose these byte tagging positions to ensure variability of tag placement within the HTTP payloads, but we also wanted to make sure that some tags would be included in HTTP headers, which would be most likely to cause middlebox interference. Since we were

not evaluating the processing overhead, we used the same 3.7 MB binary file (i.e., the Layer 4.5 kernel module) from Subsection 5.3.2 to allow reasonable file transfer times, but still result in a large amount of customization tags per transfer. Each experiment was repeated 5 times and the tag placement was confirmed by capturing the file download with `tcpdump` on the local client and parsing the capture for customization tags.

The results of each test are omitted since we did not experience any middlebox interference when customizing HTTP, even when customization tags were inserted into the HTTP header. From the file transfer packet captures, we determined that each customization scheme inserted at least one customization tag into a HTTP header, with the 0.5x scheme inserting multiple tags in each header and causing the file transfer to grow to be approximately 11 MB.

5.4 Insights

The design and evaluation of the new features for Layer 4.5 to support customization module rotation introduced some new customization challenges. In this section we will highlight the new insights we gained.

- **Rotation Order:** The device order for customization rotation should not be changed without careful consideration. We chose to prioritize customization clients over servers to support the case when the customization module being deployed will also match non-customized active socket flows. By deploying the module to the client first, the customization module can attempt to process the non-customized message and fail to identify the customization signature. This failure results in the customization module signaling Layer 4.5 to try the next module, but no other module is attached so the non-customized message is returned to the application as desired.

- **TCP Retransmits:** If customized TCP traffic is being sent over a link with a high loss rate, then it is likely that TCP retransmits are common. These retransmitted messages were previously customized by the Layer 4.5 module that was attached at the time of sending the original message. If we try to rotate a customization while retransmits are occurring without supporting both customization modules, then we could end up with the customization client using the new customization module to process traffic that was sent with the old customization module.

- **Out of Order Delivery:** When customizing UDP traffic, we must keep in mind that packet ordering is not guaranteed. Therefore, we need to support multiple customization modules for a set period of time during customization rotation in case an older message arrives after a message using the new customization module.
- **Parameter Matching:** Attaching a customization module in a deactivated state can be useful for determining the PID and/or TGID socket matching parameters for a particular application. Using a deactivated module, the customization developer can test the socket matching parameters without adversely affecting the application.
- **Socket Flags:** During the third-party middlebox testing using the DNS end customization, we identified that UDP applications may drop messages when the transport layer sets the socket "truncation" (i.e., TRUNC) flag. To avoid this flag being set, we can utilize the buffering capability from Chapter 4 to retrieve all the data from the transport layer.

5.5 Summary

In this chapter we introduced several new capabilities to the Layer 4.5 customization architecture design and prototype implementation. The new features to the NCO functions and DCA handlers provided support for customization module immediate attachment, activation, deprecation, and priority changes. These new features were all added to the customization architecture to support transitioning from a previously deployed customization module to a new customization module, both for an active socket and for future sockets. We finished with a cursory evaluation of third-party middlebox interference of Layer 4.5 customized flows. In our evaluation, we experienced minimal interference for customized DNS flows and zero interference of customized HTTP flows. We attribute the DNS interference to a device performing deep packet inspection that would have been within the controlled part of the server network and supported by the NCO.

CHAPTER 6:
Summary of Contributions

This dissertation aimed to provide enterprise and datacenter networks with a software defined protocol customization capability through the use of a Layer 4.5 customization architecture. The key contributions of this dissertation are as follows:

1. **First software defined customization architecture capable of per-process protocol customization, per-network security controls, and aiding middlebox traversal of customized application flows:** We develop and evaluate the first architecture to perform software defined network-wide orchestration of protocol customizations. The architecture not only automates deployment of customization modules to devices but also provides a platform for continuous management features such as liveliness monitoring, per-network security controls, and aid for middlebox traversal. Additionally, we conceptualize Layer 4.5 customization modules to perform application-transparent, fine-grained, process-level flow customization.

2. **Improved understanding and flexible support for application transparent customization:** Performing application transparent customizations at the socket layer has not been studied significantly and presents implementation challenges. We discovered some applications process incoming traffic following expected patterns and deviations result in processing errors, which was especially evidenced by encrypted application flows using TLS. Thus, we expand the proposed customization architecture and provide a generalized customization capability that enables customizing both encrypted and unencrypted application flows without causing application processing errors.

3. **Using the new capabilities of our architecture, we are the first to demonstrate the previously unsupported capability for active flow customization rotation:** Real-world networks must account for communication delays and the resulting customization synchronization issues. Furthermore, networks may expect to periodically

©2022 IEEE. Portions of this chapter were previously published. Reprinted with permission from D. Lukaszewski and G. Xie, "Towards Software Defined Layer 4.5 Customization," *IEEE NetSoft*, June 2022.

rotate customizations in use on both active and future application flows without causing network interruptions. Using our customization architecture, we demonstrate the previously unsupported capability for customization synchronization/rotation in a WAN setting.

6.1 Reproducibility

All code used in this dissertation to develop the Layer 4.5 customization architecture prototype and extensions is made available open-source on GitHub (https://github.com/danluke2/software_defined_customization). The repository is broken up into three branches that match a specific chapter's prototype and experiments, all tagged with version 1.0.0:

- Chapter 3: Main branch
- Chapter 4: Buffering branch
- Chapter 5: Rotating branch

Additionally, to maximize reproducibility of experiments and enable future work extensions, we provide a pre-configured Vagrant [35] hosted VM, the associated Vagrant file using VirtualBox as the hypervisor, VM configuration script, and the necessary bash scripts for each experiment performed. For experiments using the GENI infrastructure, we also provide the topology specification files.

6.2 Future Work

In this section we highlight some areas for future work that focus on the expansion of the Layer 4.5 customization architecture as a whole. We finish with one detailed area of future work to dialect the TLS protocol to combat Man-in-the-Middle (MITM) attacks.

6.2.1 Prototype Extensions

There is some work that can be accomplished focusing on the prototype implementation using different technologies, such as eBPF. It is unclear if the eBPF Linux implementation can be utilized to implement the Layer 4.5 architecture without the need to introduce new eBPF helper functions into the Linux kernel. Therefore, an implementation in eBPF to pinpoint Layer 4.5 design restrictions and challenges would be a useful endeavor. Additionally, there

is some work that expands the prototype evaluations to include different network topologies and a variety of middlebox devices.

6.2.2 Supporting Lower Layer Customization

One approach to extend our architecture to support customization at the 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 at network 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.

6.2.3 Raising the NCO Abstraction

We believe the NCO not only can run as a control application on a 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.

6.2.4 Layer 4.5 Security Analysis

Beyond increasing the programmability of the NCO, it is also worth conducting a security analysis of the NCO since it may introduce new security threats/challenges beyond what is faced by a SDN controller. For instance, the use of customization modules may present a new target for denial of service attacks, particularly during the rotation of modules. The threat model used in this dissertation, Subsection 2.1.2, should be expanded to motivate additional security functionality both within the NCO and in the customization modules it deploys.

6.2.5 Expanding Middlebox Support

The NCO has the potential to aid middleboxes in the understanding of incoming traffic, to include 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 3.4, 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 behaviour, such as control channel establishment with outside devices.

6.2.6 Intent Based Networking

Layer 4.5 customization modules could be classified by intent and deployed by the NCO. For instance, a network operator could request the network to support the collection of data from each host. This intent could be matched to a subset of pre-built customization modules, which are then automatically deployed on the network to support the network monitoring goal. Alternatively, the operator could increase the threat level to the network, which would trigger the deployment of customization modules built to match the specific security level. For instance, in the Department of Defense, the changing of cyber protection conditions (CPCON) could be the intent that triggers the deployment of customization modules [45].

6.2.7 Moving Target Networks

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. Similar to the concept of frequency hopping to reduce the likelihood for traffic interception, customization rotation can be used to change network behaviour frequently, essentially making the network a moving target from an attackers point of view. Rotating customizations in the network frequently could result in degraded network surveillance capabilities as automated tools for analyzing traffic may not be able to properly parse customized traffic. Future research to determine optimal customization rotation rates, schemes, and effectiveness remains to be conducted.

6.2.8 TLS Flow Protection

From Section 4.1, we know that if TLS flows are customized and the updated buffering capability is not used, that any customizations modifying the payload data will result in TLS processing errors. We believe this behaviour can be leveraged to dialect TLS to provide protection against attackers that are not capable of processing the specialized TLS flows. Specifically, we will target attackers attempting MITM attacks between an enterprise network controlled client and server.

TLS MITM:

There are several types of MITM attacks, with each one working differently [46]. For this reason we will focus on a simplified MITM scenario where the attacker has leveraged a vulnerability and successfully positioned themselves between the TLS client and server without being detected. At this point, the attacker is either an active participant in the connection or is passively collecting the encrypted traffic. In the active case, the attacker will intercept and decrypt the received traffic, perform some action with the plain text, and then encrypt the message again before delivery. If the attacker is passively collecting encrypted traffic, then the attacker needs another stage of the attack to simplify the decryption of the collected traffic. In either scenario, we propose that a Layer 4.5 customization can be used to add a dialect to TLS to potentially inhibit the attacker's success.

TLS Dialect:

Using lessons learned from our previous work of dialecting OpenFlow to protect the TLS handshake [18], we design a TLS dialect using Layer 4.5 customization modules to target TLS MITM attacks. First, the Layer 4.5 customization modules are built for the client and server with matching socket parameters to target to all sockets using port 443, which is the standard port for TLS traffic. The customization logic is broken up into two phases (i) the handshake phase and (ii) the encrypted traffic phase. During the TLS handshake, the client and server customization modules do not actively customize the connection and instead monitor the handshake until completion. After the handshake completes, the encrypted traffic phase starts and the customization modules become active to disrupt MITM attacks.

We first consider an attacker that is active in the connection, which means they will be decrypting traffic from the client and server. During the encrypted traffic phase, the TLS

client and server customization modules will add data to the TLS traffic. Now if a MITM attacker is present, then this attacker must be using specialized TLS processing to remove the customization data prior to message decryption to prevent TLS decode errors. We believe it is unlikely for a standard attacker to utilize custom TLS processing and to also know the network specific customization in use during the connection, especially if the dialect is being rotated on a regular basis. Thus, this unpredictable TLS dialect is highly likely to cause attacker errors and alert networks to the possibility of a MITM attack.

Alternatively, if the attacker is collecting encrypted data with the ability to decrypt the traffic offline at a later point, then our customization module will also disrupt the attacker. When the customization module adds data to the encrypted message, this results in the decryption key no longer working to decrypt the traffic. Thus, if the attacker somehow gained access to the decryption key, they must now also determine the changes made to the original encrypted message before successfully decrypting the intercepted traffic.

Development and testing of the Layer 4.5 customization modules to target TLS MITM attack disruption is left as future work. These modules are discussed for general MITM attacks, but testing should involve specific, well-known attacks against TLS to determine their effectiveness and adaptability to combat multiple attacks. Additionally, several MITM attacks include a downgrade attack component, which means the customization modules could be expanded to actively disrupt such attacks during the handshake phase.

APPENDIX: Background and Related Works

In this appendix we survey current network protocols and a sample of the customizations applied to each protocol over the years. Each customization is explored with a focus on challenges and limitation to understand what made them succeed or fail. We then present recent protocol customization efforts to include our previous research.

A.1 Network Protocol Customizations

Figure A.1 illustrates the TCP/IP stack and the network protocols we will discuss in this section, starting with Multiprotocol Label Switching (MPLS) and working up the stack to layer 5 (i.e., the application layer). For each protocol we will expand on the methods used to achieve customization of the protocol and any implementation or middlebox struggles. By reviewing previous protocol customization implementations we inform our research on what may be necessary to succeed and what can lead to deployment hurdles or even failures.

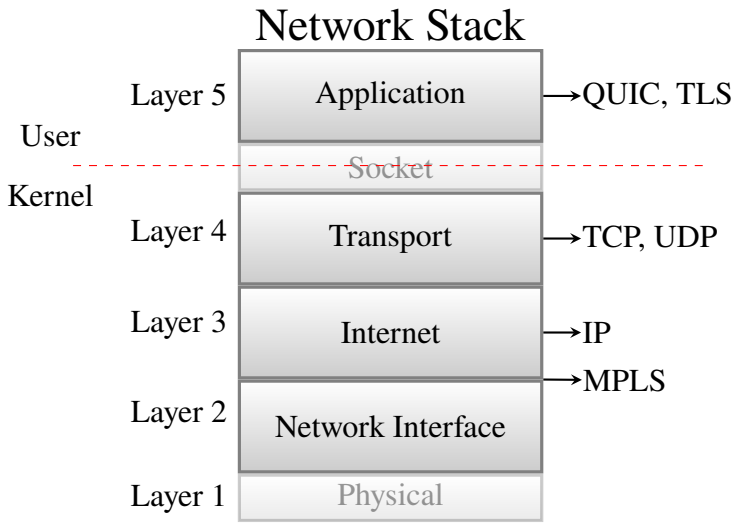


Figure A.1. TCP/IP network stack with surveyed protocols.

A.1.1 MPLS

The MPLS protocol can be viewed as a layer-based upper-layer transparent protocol customization to allow quicker routing decisions within a controlled portion of the network [47]. This protocol is unique because it may not apply to the entire path taken by a network packet because the packets are modified only within controlled portions of the network, without end host knowledge. MPLS works when the first hop router/switch in the MPLS portion of the network inspects the packet and uses the layer 3 address to create and insert a 32-bit MPLS header below layer 3, seen in Figure A.2, for subsequent devices to base routing decisions on without complex routing table lookups. Prior to leaving the MPLS network, the 32-bit MPLS header is removed to allow standard processing by the next-hop device.

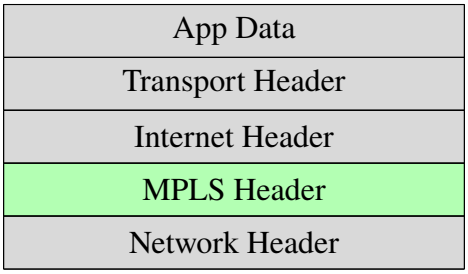


Figure A.2. MPLS packet showing 32-bit MPLS header inserted between layers 2 and 3. Adapted from [47].

The placement of MPLS between layer 2 and 3 headers was a key design decision. The protocol could not have been easily implemented as an extension to layer 2 or 3 protocols without requiring updates to all routing devices in the path, which are likely controlled by multiple service providers. Additionally, the construction of the MPLS header relies on the information in layer 3 and not all routers in the path are guaranteed to use MPLS for routing decisions. Thus, MPLS avoids middlebox interference by only working in defined segments of the network. Based on the current architecture and the struggles that exist to update layer 2 and layer 3 protocols, the MPLS implementation was designed to be used on an as needed basis and operates only in controlled networks.

A.1.2 IP

IP is the primary layer 3 protocol and it has two versions: IPv4 [48] and IPv6 [2]. The IPv6 protocol is an evolution of IPv4 to allow for larger address spaces and to remove features of IPv4 that were no longer desired. The key feature change between IPv4 and IPv6 was how IPv6 can be extended. Figure A.3 illustrates the key differences between IPv4 and IPv6 from a customization perspective.

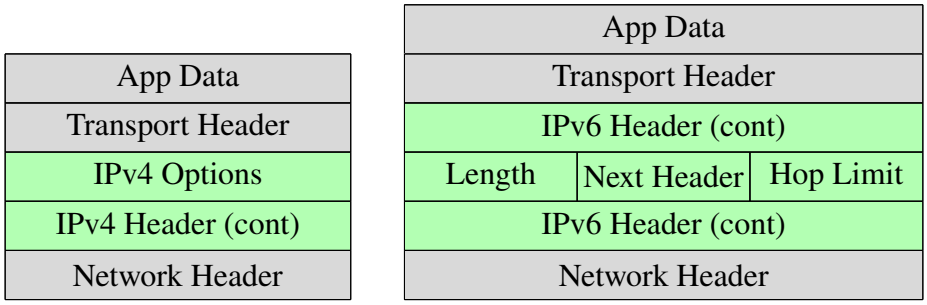


Figure A.3. IPv4 (left) and IPv6 (right) packets showing applicable extension fields for customization. Adapted from [2], [48].

The IPv4 protocol supported extension by using an Options field, while the IPv6 protocol uses a more flexible Next Header field to chain additional headers prior to the layer 4 transport header. Additionally, unlike the IPv4 options, most of the IPv6 follow-on headers are not processed until the end node receives the packets. As seen by Figure A.4, IPv6 has suffered from limited deployment. The IPv6 RFC dates back to 1998, but as of July 2022 the adoption of the protocol was approximately 40%.

The main drawback of layer 3 customization is the potential for middlebox interference, particularly from routers and switches. In 2005, Fonseca et al. conducted experiments to determine if IPv4 packets using options were accepted over various networks [49]. The authors found that multiple networks would drop packets that included IP options, despite being standardized in the RFC. One reason packets may have been dropped was the perceived overhead of repeated processing by routers when the first router did not recognize the option being used [49]. This middlebox problem is also present when using the IPv6 Next Header field for options. RFC 7872 documents real world data regarding the drop rate of IPv6 packets using various Next Header fields [7].

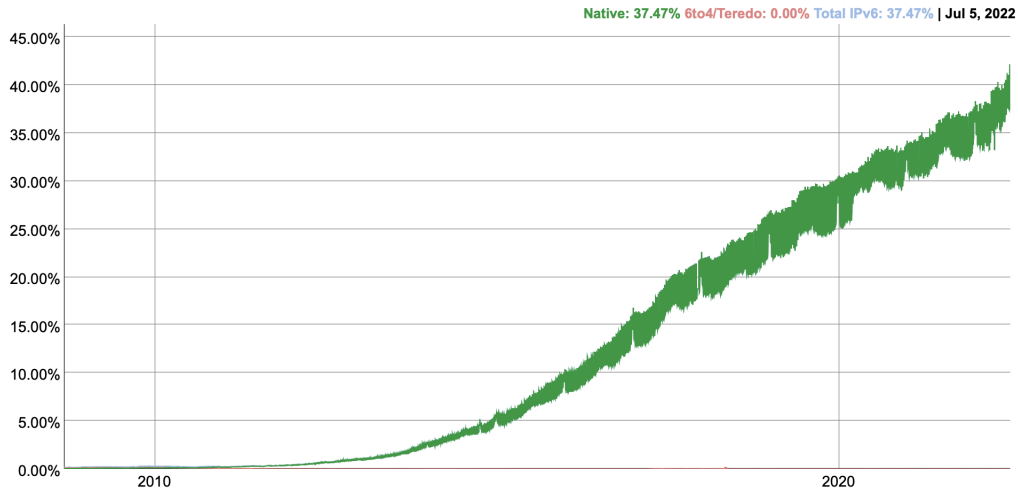


Figure A.4. Google IPv6 adoption tracker. Source [4].

The first customization to IP we discuss is Multipath IP (MPIP) [50]. MPIP was designed to bring multipath support to layer 3 without requiring layer 4 protocols to explicitly request the multipath capability. MPIP has not been extensively studied and is unlikely to be widely accepted due to some fundamental changes it makes at the IP layer. One of these fundamental changes is that MPIP is a connection based protocol and needs to determine and use path statistics. Additionally, to get over middlebox hurdles, MPIP will either create fake TCP handshakes on additional paths or conduct UDP wrapping of TCP packets on additional paths. These methods are clearly outside the bounds of the IP layer and present a significant barrier to its adoption. The MPIP protocol helps illustrate the challenges of middlebox interference influencing the protocol customization design.

A protocol customization at layer 3 that ultimately moved to layer 5 was IP multicast [51]. Yang-hau et al. compared multicast at the IP layer to a new multicast implementation at the application layer. The authors discuss known issues with IP multicast, such as limited deployment due to infrastructure level changes, and the performance penalties of implementation at the application layer. The authors ultimately concluded that shifting multicast to the application layer addresses multiple issues with tolerable performance penalties. This work motivates that trade-offs exist when deciding on the implementation layer for protocol customization and implementations at one layer may fail, but ultimately succeed if moved into a different layer.

A.1.3 TCP

TCP is one of the main protocols at layer 4, and it has undergone significant changes since the initial specification was developed. As seen in Figure A.5, TCP utilizes an Options field, like IPv4, to allow extensions to the protocol, but officially adding TCP option extensions accepted by the networking community is not always a quick process. For instance, TCP has several standardized options defined over various RFCs, some of which have taken upwards of a decade or longer to be widely deployed, such as window scaling and selective acknowledgments [3]. One of the more recent TCP extensions was the development of the application transparent MPTCP, which also took over a decade to be implemented into the Linux kernel [10], [52].

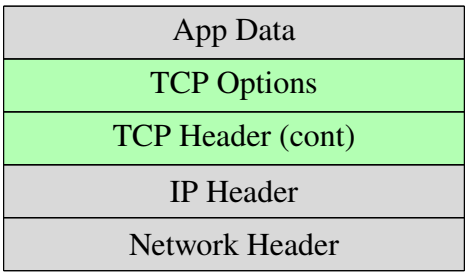


Figure A.5. TCP packet showing Options field for customization support. Adapted from [1].

MPTCP is designed for general use, which means it must perform protocol negotiation to ensure both end-points support the protocol. Additionally, this negotiation phase is subject to middlebox interference as some devices in the network may remove unknown TCP options. Therefore, MPTCP must first signal multipath capability during the TCP 3-way handshake using the defined MP_CAPABLE option. If the end host is not multipath capable or the option is removed by a middlebox device, the connection will default to a standard TCP connection. Otherwise, a MPTCP handshake is conducted and additional subflow connections are established. This fallback mechanism is a common approach taken by protocols to support backward compatibility.

A.1.4 UDP

UDP is a lightweight protocol that could be used in lieu of TCP but, unlike TCP, UDP was initially designed without an Options field. However, there is recent work in progress to incorporate an Option field into the protocol, shown in Figure A.6 [53]. This specification describes how to overload the UDP header Length field to indicate the presence of options inserted after the packet's data section. The length field was chosen because the IP header includes a Total Length field that accounts for the length of the UDP header and data, thus making the field redundant [53]. As with TCP, UDP options may experience middlebox interference from devices inspecting the layer 4 header [54]. Zullo et al. found that checksum calculation and verification was not consistent across the internet, resulting in packets not reaching the destination when the UDP Options field was used. The authors proposed the use of a checksum compensating option to overcome some middlebox interference, with this option ultimately being incorporated into the draft RFC. Similar to other protocol customization efforts, the UDP Options field is taking a significant time to become standardized.

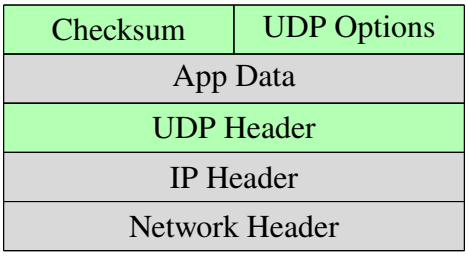


Figure A.6. UDP packet showing proposed UDP Options field for customization support. Adapted from [55].

In [56], we prototyped a Linux kernel extension to support a Multipath User Datagram Protocol (MPUDP) protocol, much like MPTCP. The MPUDP prototype showed promise when applied to Virtual Private Network (VPN) networks, which avoids TCP-over-TCP problems and leverages multiple paths between the VPN client and server. However, this MPUDP implementation faces several barriers to deployment. First, the MPUDP implementation required modification to the kernel UDP function and based on the long MPTCP adoption timeline, it is highly unlikely that a MPUDP implementation requiring kernel modification will be successful.

Second, the MPUDP implementation was written to be generic like the MPTCP protocol and apply to all UDP connections. However, this decision does not fit well for UDP applications because UDP connections are stateless and may be short-lived (e.g., DNS). Thus, a MPUDP solution would need a method to track UDP state and would also need a method to specify which applications should use MPUDP. Both of these requirements are beyond the responsibilities of UDP and raise the complexity of the protocol beyond what is likely to be accepted by the networking community.

A.1.5 TCPLS

In 2020, researchers proposed a new layer 4 protocol based on TCP and TLS, known as TCPLS [11], [57]. Figure A.7 provides a comparison between the standard TLS over TCP packet and that of the new TCPLS packet.

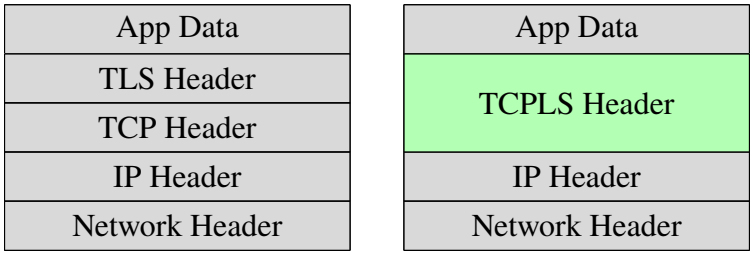


Figure A.7. TLS over TCP packet (left) vs. TCPLS packet (right). Adapted from [57].

TCP was historically extended using the Options field, but this field is limited to a total of 40 bytes, which is further limited by the number of options being used. This limitation to the TCP header length hinders further extensions, especially when multiple extensions are desired during the same connection. TCPLS was designed to extend TCP beyond the maximum header size restrictions, but also to avoid middlebox interference problems experienced at layer 4. First, TCPLS mitigates middlebox interference by using TLS encryption, which common middleboxes are unable to process. Second, TCPLS leverages TLS 1.3 messages to send portions of the TCP header (e.g., options) which allows for arbitrarily long TCP header parameters. The authors of TCPLS desired capabilities of both protocols to work together, but there is no architectural support for cross-layer coupling of protocols

and TCP options alone could not provide for this. It is unclear to what extent TCPLS will suffer from middlebox interference and if it will ultimately succeed as a new layer 4 protocol as its development is at the prototype stage.

A.1.6 QUIC

A newer layer 5 protocol developed to address latency-sensitive web services is the QUIC [5], [58]. As seen in Figure A.8, QUIC incorporates aspects of TLS and TCP while running over a standard UDP connection.

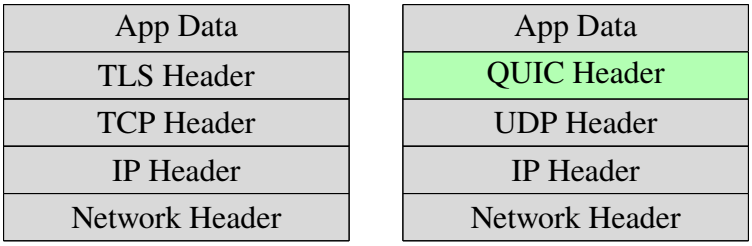


Figure A.8. TLS over TCP packet (left) vs. QUIC packet (right). Adapted from [5].

Similar to TCPLS, the design of QUIC is influenced by middlebox interference and also signals the trouble of extending and integrating lower layer protocols. The protocol aims to decrease middlebox interference by encrypting the header/payload and by running over UDP, but allows falling back to a TCP connection if UDP traffic is being filtered [58]. During the initial design of QUIC, some firewalls would block QUIC traffic on the network without explicit rules to do so [58]. Note that to properly fix firewall issues that filter a protocol unnecessarily, there must be firewall vendor support to update the software, which can be a hindrance to protocol development and adoption.

One of the first customization efforts for QUIC was to introduce multipath support since MPUDP capability does not exist in the Linux kernel [59]. The authors design the QUIC extension using MPTCP as a guide and show the extension can handle packet losses better than MPTCP. The main concern with this customization approach is the increased complexity added to the protocol specification as the extension is not a simple add-on feature to a running QUIC implementation.

A.2 Related Work

In this section we survey related works relevant to protocol customization to include methods for host-based and network-based customization.

A.2.1 X-Kernel

The x-kernel design of the 1990s argued for the use of an object-based framework for protocol implementation [60]. In this framework, objects consisted of protocols, sessions, and messages. The protocol objects were the known protocols, such as TCP and IP. Session objects were an instance of a protocol and included a protocol interpreter and local state information. Message objects would be the packets moving through the session and protocol objects. The authors argue that this design allowed for ease of protocol development, conducting protocol experimentation, and building complex protocols via composition of single-function protocols. The main downside to this framework is that it was designed only for end-host workstations and all protocols are developed/implemented in kernel space.

The x-kernel was focused on a different methodology for developing new protocols since this was the early days of the internet. The Layer 4.5 architecture proposes a different approach for customizing protocols. Just like the x-kernel, Layer 4.5 customization modules are designed like objects to allow easily inserting/removing them from the Linux kernel. Unlike the x-kernel approach, we do not develop an entirely new kernel architecture that should be used instead of existing architectures.

A.2.2 eBPF

eBPF expands the original 1992 implementation by introducing map storage to efficiently share information between the kernel and user space, increased restricted kernel call access, and expanded triggers to include user and kernel functions [12]. eBPF programs are written in user space and compiled into eBPF byte code that can be executed by the kernel at various hook points via an interpreter. Unlike the kernel module approach, eBPF has been designed to take security into account as part of the execution process. eBPF programs must pass through a verifier prior to being allowed to execute on a eBPF virtual machine. The verifier starts by constructing a control flow graph and checking for unbounded loops to determine that all programs will terminate and that dead code is not present [12]. The verifier is also

responsible for determining that the eBPF program will conduct memory accesses only within allowed bounds [24]. These checks limit the flexibility of eBPF programs, but the checks are necessary to restrict potentially insecure programs from running with kernel level permissions.

The use of eBPF has led to relevant research on extending and tuning TCP parameters with a new program called TCP-BPF [25]. TCP-BPF introduces the ability to set TCP parameters on a per-connection basis and adjust them programmatically during the connection. The main use case focused on in the paper is for datacenter connections where the congestion windows and receive buffers can be set much lower than normal connections [25]. There is also follow-on work using this TCP-BPF program to extend the TCP stack [61]. Bonaventure and Tran show that eBPF can be used to implement current TCP options as well as new options not standardized in the protocol. The authors comment on future work that includes making the TCP implementation fully modular and incorporating the eBPF methods used into other protocols such as UDP. The main downside to this use of eBPF is the focused application to the TCP protocol. It remains unclear if this same approach will be applied to other protocols, such as UDP.

eBPF can also be used to perform packet transformations, such as IPv4/IPv6 conversions using hooks at the traffic control stage of packet processing [62]. In this blog post, the author demonstrates using eBPF filters to convert packets leaving the host to an IPv4 address and the incoming IPv4 packets into IPv6 packets. To make these changes on the incoming packets, the eBPF filters must generate the proper headers for IPv6 that are different from the IPv4 headers and then restructure the filtered packet to appear as if it was an IPv6 packet prior to handing off to the TCP/IP stack for processing. In this simple example, the authors are illuminating the potential power of using eBPF to do layer-based customization. However, this particular approach leads to wasteful work being conducted.

Our Layer 4.5 customization architecture prototype did not use eBPF due to concerns about the possible need to extend eBPF functionality and helper functions within the Linux kernel to fulfill design requirements. For example, the eBPF “direct packet access” helper functions are available to the traffic control layer, but are not available at the socket layer. Additionally, eBPF programs must pass verification through the eBPF verifier, which can be quite restrictive. For instance, the eBPF program must be verified to finish, which would

restrict the use of kernel locks that may be required when dealing with applications using multithreading/multiprocessing. However, we believe it is possible to use eBPF at the cost of possible architecture restrictions and kernel modification, which can take a significant amount of time and effort.

A.2.3 XDP and DPDK

Two main packet processing techniques available at the Network Interface Card (NIC) are XDP and the Data Plane Development Kit (DPDK). These mechanisms both allow for the interception of incoming network packets prior to processing by the kernel network stack. DPDK is used to pass control of the NIC to user space, bypassing the kernel network stack and using a user space network stack [63]. XDP, on the other hand, can be used with eBPF to allow eBPF programs written in user space to act on the incoming packets prior to the kernel network stack [64].

These mechanisms may seem useful for implementation of Layer 4.5 since they can immediately perform operations on packets prior to being handled by the kernel stack or user implemented stack. However, both of these approaches lack aspects of the desired design of Layer 4.5. For instance, XDP is designed for operations on incoming packets only and would need to perform Layer 4.5 processing prior to kernel stack processing, which seems like a violation of the kernel layer based processing design. DPDK must implement the network stack in user space and can't rely on kernel level security tools or functions without incurring a context switch penalty. Since this network stack is user defined, we could implement Layer 4.5 entirely in user space using this method. This approach will only apply to endpoints using DPDK and is thus not a general enough approach to consider.

A.2.4 In-Band Network Telemetry

The use of SDNs and a programmable data plane gave rise to the concept of In-Band Network Telemetry (INT) [65]. INT utilizes a programmable data plane to embed telemetry metadata into network packets using different methods. The INT over TCP/UDP method inserts the metadata between the packet header and the application data [66], illustrated in Figure A.9.

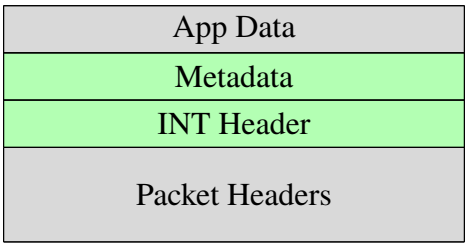


Figure A.9. Network packet showing INT header and associated metadata inserted between packet header and application data by a network device. Adapted from [66].

Each network device adds to the telemetry metadata until the last-hop router receives the packet. At this point, the telemetry data is removed from the packet and delivered to a telemetry server. The insertion of telemetry data between the packet headers and application data is similar to MPLS but avoids conflicts with standard packet processing mechanisms within the network designed to parse packets between layers 2 and 4. Our customization architecture follows the same approach when adding data during protocol customization to also avoid potential interference from common network devices.

The main difference between our architecture and INT is the device performing the customization and the purpose of the customizations. All INT metadata is inserted within the network by programmable devices, such as switches, while our architecture targets the end-host. Additionally, our protocol customizations are written to match specific applications and flows, while INT matches generic network packets for the purpose of performing network measurements.

A.2.5 Network Function Virtualization

Network Function Virtualization (NFV) aims to replace specialized physical middlebox processing with that of software processes on commercial-off-the-shelf equipment [67]. In [68], 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 leverages 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.

A.2.6 L3AF

The Walmart L3AF project [13] aims to support kernel functions as a service via a central repository. The L3AF project utilizes a user space daemon for distribution of eBPF programs targeted at the XDP and traffic control layers of the stack to perform the desired kernel function, such as load balancing and rate limiting at the NIC. The Layer 4.5 customization architecture differs from L3AF in that we target customization of applications, provide network-wide orchestration and management of these customizations, and each customization is applied on a per-process basis. Chapter 2 shows that this continuous management enables novel solutions for security and middlebox traversal.

A.2.7 Application Specific Customization

There are several methods to provide customization capabilities to targeted applications and/or application protocols. First, Bonaventure et al. in [14], [15] make the case that experimentation and feature design of routing protocols is a slow process and argue that protocol design should allow for custom plugins to extend the functionality without changing the protocol standard. The authors implement plugin versions of multiple network protocols and demonstrate the insertion of various plugins during protocol communication, without the need to recompile the protocol. The plugins are achieved by using a user space implementation of eBPF and adding anchor points throughout the protocol to signify points where a given function can be replaced in the protocol execution by another function. Through the use of multiple anchors, a plugin could achieve new functionality within the protocol.

The protocol plugin work provides further insight into customization distribution. The authors design a plugin negotiating and distribution scheme, but just like the L3AF project do not provide a mechanism for the continuous management and monitoring of the plugins in use on the network. Additionally, the main drawback to this form of customization is the time it will take to fully implement and standardize each pluginized application protocol. Layer 4.5 differs from the protocol plugin work by targeting application flows for customizations

to allow remaining transparent to the application being customized. However, the plugin work is able to directly modify application functionality, which Layer 4.5 is not capable of doing.

Second, in [69], the authors describe using the service mesh layer present between microservice applications and the transport layer to achieve customizations. 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.

Last, application proxies can be used to perform protocol customization for a specific application. For example, in [18] 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 must traverse the network stack multiple times resulting in additional processing overhead. Additionally, 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. Through this continuous management, customizations can be rotated transparent to the application.

A.2.8 Virtual Transport Layer

The Virtual Transport Layer (VTL) [70] leverages eBPF functionality at the socket layer to perform application transparent customization. The customization performed by VTL focuses on improving the performance of TCP applications depending on the unique conditions of the network. When an application first creates a TCP connection, VTL intercepts the message and uses a machine learning component to determine which protocol is best for the given application and dynamically maps the application socket to that of the optimal protocol, such as UDP or QUIC. When a message is received by the device, the message is intercepted using XDP to redirect the message to the VTL socket before delivery to the application's TCP socket. The VTL work focuses on one customization capability and is complementary to our efforts.

A.2.9 Middlebox Support

It is well known that the current protocol extension methodology suffers from middlebox interference [7], [10], [49], [54], [58]. To specifically address interference from middleboxes conducting deep packet inspection, some protocols leverage application encryption [11], [58]. 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 [71], the authors develop the OpenBox architecture for SDN environments and leverages the common processing conducted by multiple virtualized packet inspection services (i.e., middleboxes) to reduce redundant processing. Since middlebox interference is a primary concern when performing protocol customizations, the Layer 4.5 customization architecture may be able to integrate with OpenBox to allow proper customization processing by treating protocol customization as a necessary middlebox processing step.

A.3 Our Previous Work

In this section we shift to our previously published, peer-reviewed work during the PhD research timeline involving protocol customization.

A.3.1 Protocol Dialecting

In [18], the authors create multiple protocol dialects for the SDN control plane using the OpenFlow protocol. The dialects are developed to provide additional protection to the TLS handshake and subsequent communications between the SDN switch and controller. The first dialect customizes the transaction ID field of the OpenFlow header, which holds a 32-bit randomly generated value. During the first OpenFlow Hello message from the switch and controller, this value is transparently replaced with a keyed message authentication code that must be verified by each device before proceeding with the TLS handshake. After the verification completes, a second dialect is used to transparently append a 512-bit keyed hash to each message send over the control channel.

Each of the dialects used achieves application transparency by using proxies between the controller and switch. This proxy approach is viable, but requires additional infrastructure, which presents a deployment challenge. Additionally, the proxies add communication overhead since they must intercept each message, process the OpenFlow traffic, customize it, and

then send the traffic to the receiving proxy for verification. The customization architecture of this dissertation follows the application transparent customization, but does not require the use of proxies by moving the customization point to be within the network stack on each end device.

A.3.2 Link Optimization

U.S. Navy ships rely on satellite networks with relatively low bandwidth capability. Since the Department of Defense controls both endpoints of the satellite connection (i.e., ship and shore), this means we can more easily specialize the protocols over this connection. In [41], the authors customized the DNS protocol over a simulated satellite link by removing "unnecessary" bits from each request and reply message. The goal was to transmit the minimum necessary information to successfully perform a DNS request and process the reply. To accomplish this task, the authors developed custom DNS servers on both endpoints of the satellite link that were capable of using the customized DNS protocol. This method allowed the end hosts on the network to utilize the standard DNS protocol and only perform the customization on the resource constrained link.

Given the unique operating environments of naval ships, our generic customization architecture needs to support customization updates from a centralized source. However, similar to the dialect work, this work suffers from overhead and development complexity. For instance, if the DNS protocol was to be further customized, the custom server on both ends would need to be updated before using the new customization, which could be challenging.

List of References

[1] J. Postel, "Transmission control protocol," Internet Requests for Comments, Tech. Rep. 793, 1981 [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[2] S. E. Deering and R. M. Hinden, "Internet protocol, version 6 (IPv6) specification," Internet Requests for Comments, Tech. Rep. 2460, 1998 [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2460

[3] K. Fukuda, "An analysis of longitudinal TCP passive measurements," in *International Workshop on Traffic Monitoring and Analysis*, 2011, pp. 29–36.

[4] Google, "Google IPv6," accessed Jul. 05, 2022 [Online]. Available: https://www.google.com/intl/en/ipv6/statistics.html

[5] J. Iyengar and M. Thomson, "QUIC: A UDP-Based multiplexed and secure transport," Internet Requests for Comments, RFC 9000, 2021 [Online]. Available: https://datatracker.ietf.org/doc/html/rfc9000

[6] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet, "Revealing middlebox interference with tracebox," in *Proceedings of the 2013 Conference on Internet Measurement*, 2013, pp. 1–8.

[7] F. Gont, J. Linkova, T. Chown, and W. Liu, "Observations on the dropping of packets with IPv6 extension headers in the real world," Internet Requests for Comments, Tech. Rep. 7872, 2016 [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7872

[8] S. Huang, F. Cuadrado, and S. Uhlig, "Middleboxes in the internet: A HTTP perspective," in *2017 Network Traffic Measurement and Analysis Conference (TMA)*, 2017, pp. 1–9.

[9] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and issues," Internet Requests for Comments, Tech. Rep. 3234, 2002 [Online]. Available: http://www.rfc-editor.org/rfc/rfc3234.txt

[10] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, and C. Paasch, "TCP extensions for multipath operation with multiple addresses," Internet Requests for Comments, Tech. Rep. 8684, 2020 [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8684

[11] F. Rochet, E. Assogba, and O. Bonaventure, "TCPLS: Closely integrating TCP and TLS," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, 2020, pp. 45–52.

[12] B. Gregg, *BPF Performance Tools*, 1st ed. Boston, MA, USA: Addison-Wesley Professional, 2019.

[13] The Linux Foundation Projects, "L3AF," accessed Jun. 09, 2022 [Online]. Available: https://l3af.io

[14] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure, "Pluginizing QUIC," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 59–74.

[15] T. Wirtgen, C. Dénos, Q. De Coninck, M. Jadin, and O. Bonaventure, "The case for pluginized routing protocols," in *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, 2019, pp. 1–12.

[16] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "TRIMMER: Application specialization for code debloating," in *Proc. 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018 [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3238147.3238160

[17] D. K. Hong, Q. A. Chen, and Z. M. Mao, "An initial investigation of protocol customization," in *Proceedings of the 2017 workshop on forming an ecosystem around software transformation*, 2017.

[18] M. Sjoholmsierchio, B. Hale, D. Lukaszewski, and G. Xie, "Strengthening SDN security: Protocol dialecting and downgrade attacks," in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, 2021, pp. 321–329.

[19] T. Taleb, B. Mada, M.-I. Corici, A. Nakao, and H. Flinck, "PERMIT: Network slicing for personalized 5g mobile telecommunications," *IEEE Communications Magazine*, vol. 55, no. 5, 2017.

[20] H. Derhamy, J. Eliasson, and J. Delsing, "IoT interoperability — On-Demand and low latency transparent multiprotocol translator," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1754–1763, April 2017.

[21] Z.-L. Zhang, U. K. Dayalan, E. Ramadan, and T. J. Salo, "Towards a software-defined, fine-grained qos framework for 5g and beyond networks," in *Proceedings of the ACM SIGCOMM 2021 Workshop on Network-Application Integration*, 2021, pp. 7–13.

[22] Y.-W. E. Sung, X. Sun, S. G. Rao, G. G. Xie, and D. A. Maltz, "Towards systematic design of enterprise networks," *IEEE/ACM Transactions on Networking*, vol. 19, no. 3, pp. 695–708, December 2010.

[23] K. Jones, "Loadable kernel modules," *Login: The Magazine of USENIX and SAGE, Special Focus Issue: Security*, vol. 26, no. 7, November 2001 [Online]. Available: https://www.usenix.org/system/files/login/articles/1832-jones2.pdf

[24] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, January 2020.

[25] L. Brakmo, "TCP-BPF: Programmatically tuning TCP behavior through BPF," in *NetDev 2.2*, 2017 [Online]. Available: https://legacy.netdevconf.info/2.2/papers/brakmo-tcpbpf-talk.pdf

[26] T. J. Salo and Z.-L. Zhang, "Semantically aware, mission-oriented (samo) networks: A framework for application/network integration," in *Workshop on Network Application Integration/CoDesign*, 2020.

[27] D. Lachos, Q. Xiang, C. Rothenberg, S. Randriamasy, L. M. Contreras, and B. Ohlman, "Towards deep network & application integration: Possibilities, challenges, and research directions," in *Workshop on Network Application Integration/CoDesign*, 2020.

[28] U. Naseer and T. A. Benson, "Configanator: A data-driven approach to improving {CDN} performance." in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 1135–1158.

[29] NSF, "What is GENI," accessed Jun. 03, 2022 [Online]. Available: http://www.geni.net/about-geni/what-is-geni/

[30] J. Corbet, "Toward signed BPF programs," accessed Aug. 04, 2022 [Online]. Available: https://lwn.net/Articles/853489/

[31] M. Bellare, A. C. Singh, J. Jaeger, M. Nyayapati, and I. Stepanovs, "Ratcheted encryption and key exchange: The security of messaging," in *Annual International Cryptology Conference*, 2017 [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-63697-9_21

[32] H. Krawczyk, "Cryptographic extraction and key derivation: The hkdf scheme," in *Annual Cryptology Conference*, 2010 [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-14623-7_34

[33] NIST, "Glossary: Challenge-response protocol," accessed January 30, 2022. Available: https://csrc.nist.gov/glossary/term/challenge_response_protocol

[34] D. Lukaszewski and G. Xie, "Towards software defined layer 4.5 customization," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 330–338.

[35] HashiCorp, "Vagrant: Development environments made easy," accessed Jun. 27, 2022 [Online]. Available: https://www.vagrantup.com

[36] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, 2014 [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2620728.2620744

[37] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin, and J. Aguilar, "Towards the deployment of machine learning solutions in network traffic classification: A systematic survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1988–2014, November 2018.

[38] G. Fairhurst, "Update to IANA registration procedures for pool 3 values in the differentiated services field codepoints (DSCP) registry," Internet Requests for Comments, Tech. Rep. 8436, 2018 [Online]. Available: https://www.rfc-editor.org/rfc/rfc8436

[39] R. Barik, M. Welzl, A. Elmokashfi, T. Dreibholz, S. Islam, and S. Gjessing, "On the utility of unregulated IP diffserv code point (DSCP) usage by end systems," *Performance Evaluation*, vol. 135, p. 102036, November 2019. Available: https://www.sciencedirect.com/science/article/pii/S0166531619300203

[40] E. Bergen, "Dynamic data exfiltration over common protocols via socket layer protocol customization," M.S. thesis, Dept. of Comp. Sci., Naval Postgraduate School, Monterey, CA, USA, 2022.

[41] K. Pittner, D. Lukaszewski, and G. Xie, "An empirical study of application-aware traffic compression for shipboard satcom links," in *2021 IEEE Military Communications Conference (MILCOM)*, 2021, pp. 213–218.

[42] J. Damas, M. Graff, and P. Vixie, "Extension mechanisms for DNS (EDNS(0))," Internet Requests for Comments, Tech. Rep. 6891, 2013 [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6891

[43] M. Abbasi, A. Shahraki, and A. Taherkordi, "Deep learning for network traffic monitoring and analysis (ntma): A survey," *Computer Communications*, vol. 170, no. 1, pp. 19–41, March 2021.

[44] X. Wang, "On the feasibility of detecting software supply chain attacks," in *2021 IEEE Military Communications Conference (MILCOM*, 2021 [Online], pp. 458–463. Available: https://ieeexplore.ieee.org/abstract/document/9652901

[45] L. Totimeh and A. Barthel, "Fleet Cyber Readiness: Cyber Operational Response Procedures," accessed Aug. 05, 2022 [Online]. Available: https://www.doncio.navy.mil/chips/ArticleDetails.aspx?ID=14055

[46] S.-W. Han, H. Kwon, C. Hahn, D. Koo, and J. Hur, "A survey on MITM and its countermeasures in the TLS handshake protocol," in *2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2016, pp. 724–729.

[47] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol label switching architecture," Internet Requests for Comments, Tech. Rep. 3031, 2001 [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3031

[48] J. Postel, "Internet protocol," Internet Requests for Comments, Tech. Rep. 791, 1981 [Online]. Available: https://datatracker.ietf.org/doc/html/rfc791

[49] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, "IP options are not an option," Technical report, EECS Department, University of California, Berkeley, Tech. Rep., 2005 [Online]. Available: https://www.academia.edu/12804189/IP_options_are_not_an_option?from=cover_page

[50] L. Sun, G. Tian, G. Zhu, Y. Liu, H. Shi, and D. Dai, "Multipath IP routing on end devices: Motivation, design, and performance," in *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, 2018, pp. 1–9.

[51] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang, "A case for end system multicast," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1456–1471, October 2002.

[52] C. Paasch and S. Barre, "Multipath TCP in the Linux kernel," accessed Jun. 09, 2022 [Online]. Available: http://www.multipath-tcp.org

[53] J. Touch, "Transport options for UDP," Internet-Draft draft-touch-tsvwg-udp-options-18, IETF Secretariat, Tech. Rep. 18, 2022 [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tsvwg-udp-options/

[54] R. Zullo, T. Jones, and G. Fairhurst, "Overcoming the sorrows of the young UDP options," in *2020 Conference on Network Traffic Management and Analysis*, 2020 [Online]. Available: https://tma.ifip.org/2020/wp-content/uploads/sites/9/2020/06/tma2020-camera-paper70.pdf

[55] J. Postel, "User datagram protocol," Internet Requests for Comments, Tech. Rep. 768, 1980 [Online]. Available: http://www.rfc-editor.org/rfc/rfc768.txt

[56] D. Lukaszewski, "Multipath transport for virtual private networks," M.S. thesis, Dept. of Comp. Sci., Naval Postgraduate School, Monterey, CA, USA, 2017.

[57] M. Piraux, F. Rochet, and O. Bonaventure, "TCPLS: Modern transport services with TCP and TLS," Internet-Draft draft-piraux-tcpls-02, Tech. Rep. 02, 2022 [Online]. Available: https://datatracker.ietf.org/doc/draft-piraux-tcpls/

[58] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, and others, "The QUIC transport protocol: Design and internet-scale deployment," in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 183–196.

[59] T. Viernickel, A. Froemmgen, A. Rizk, B. Koldehofe, and R. Steinmetz, "Multipath QUIC: A deployable multipath transport protocol," in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–7.

[60] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, p. 64, January 1991.

[61] V.-H. Tran and O. Bonaventure, "Beyond socket options: Making the Linux TCP stack truly extensible," in *2019 IFIP Networking Conference (IFIP Networking)*, 2019, pp. 1–9, tex.organization: IEEE.

[62] G. Marsden, "BPF: Using BPF to do packet transformation," accessed Aug. 04, 2022 [Online]. Available: https://blogs.oracle.com/linux/notes-on-bpf-6

[63] The Linux Foundation Projects, "Data Plane Development Kit," accessed Jan. 30, 2022 [Online]. Available: https://www.dpdk.org

[64] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, 2018, pp. 54–66.

[65] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li, "In-band network telemetry: A survey," *Computer Networks*, vol. 186, p. 107763, February 2021.

[66] Open Networking Foundation, "INT dataplane specification," accessed Aug. 05, 2022 [Online]. Available: https://p4.org/p4-spec/docs/INT_v2_1.pdf

[67] B. Yi, X. Wang, K. Li, M. Huang *et al.*, "A comprehensive survey of network function virtualization," *Computer Networks*, vol. 133, pp. 212–262, March 2018.

[68] E. Battiston, D. Moro, G. Verticale, and A. Capone, "CHIMA: A framework for network services deployment and performance assurance," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 163–170.

[69] S. Ashok, P. B. Godfrey, and R. Mittal, "Leveraging service meshes as a new network layer," in *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*, 2021, pp. 229–236.

[70] E.-F. Bonfoh, S. Medjiah, and C. Chassot, "A zero-touch solution for transport layer adaptation to applications and networks," in *2021 IFIP Networking Conference (IFIP Networking)*, 2021, pp. 1–9.

[71] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: A software-defined framework for developing, deploying, and managing network functions," in *2016 ACM SIGCOMM Conference*, 2016, pp. 511–524.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California