March 2019

Host-Based Traffic Engineering: Network Endpoints with the Capabilities of SDN-Enabled Switches

Jeffrey Estrada
Worcester Polytechnic Institute

Julian Philippe Lanson
Worcester Polytechnic Institute

Remy Kaldawy
Worcester Polytechnic Institute

Host-Based Traffic Engineering: Network Endpoints with the Capabilities of SDN-Enabled Switches

A MAJOR QUALIFYING PROJECT

Submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science in

Computer Science

by

Remy Kaldawy

Julian P. Lanson

Jeffrey Estrada

March 22, 2019

APPROVED:

Professor Craig A. Shue, Project Advisor

Abstract

As effective internal security monitoring in enterprise networks become more necessary, IT specialists' need for fine-grained control over traffic flow becomes more pressing. The software-defined networking (SDN) paradigm provides a viable solution to the problem of directing network connections through arbitrary paths; however, for an enterprise to support traditional SDN it must upgrade most (if not all) of its switches to modern OpenFlow-enabled models, which is often prohibitively expensive. To date, a practical alternative to the switch-based SDN architecture does not exist. In this paper, we present SHARP, a host-based SDN design that achieves feature parity with traditional switch-based SDNs. SHARP makes use of VLAN tagging and our own overlay-style networking protocol to allow endpoints to dictate how their packets should be routed through the network by switches and other hosts. Our system can provide the same degree of routing control as a switch-based SDN without requiring costly upgrades and vendor lock-in. SHARP additionally surpasses the feature set of switch-based SDNs by enabling coarse-grain routing control at Internet scale. To demonstrate the validity and benefits of our system, we incorporate SHARP into PEACE, a next-generation SDN firewall under development at WPI.

Contents

List of Figures

1 Introduction

The heart of an enterprise IT technician's responsibility is ensuring that the company's network is reliable and secure. Computer networking is so integral to the operation of modern businesses that any unforeseen downtime on an enterprise's networks will temporarily cripple productivity and negatively impact the organization's mission. One way network administrators can avoid such an unfavorable scenario is ensuring that the core of their networks do not become so congested that traffic cannot flow efficiently. Since network traffic volume is unpredictable on short timescales [1], to prevent choke points from forming network administrators must be able to dynamically reroute traffic through alternative paths to reach their destination. In addition to poor load-balancing, weak security is also a detriment to the health of a network. If perimeter defenses such as firewalls and intrusion detection or prevention systems (IDS/IPS) fail to block malware from being downloaded, or if a piece of malware is introduced out-of-band (e.g. via a USB disk), often times there are no additional controls to prevent it from spreading across the entire network. Whether the infection results in paying ransoms, loss of data, or restoring machines from backup, every time a new machine on the network is infected, the problem is compounded. To prevent an isolated incident from engulfing all the hosts on the network, internal traffic needs to be flexibly routed through IDSes positioned within the network, as opposed to at its edge.

Both of these scenarios make it clear that modern IT technicians need a high level of dynamic control over the flow of traffic in their networks in order to perform their jobs properly. Today, software-defined networking (SDN) is the standard mechanism for delivering this capability. In the SDN paradigm, the routing decision process is separated from the data forwarding plane (physically transferring bits from one ingress port to an egress port) and gets offloaded to a remote system called the "controller." To enable SDN networks, most of the large networking devices companies offer switch models with special software that communicates with the remote controller program to obtain routing decisions for each flow they encounter. However, in order to fully support SDN, enterprises must upgrade many of their legacy networking hardware to the modern SDN-enabled models. Each SDN-enabled switch can cost thousands of dollars, rendering such transitions prohibitively expensive. Furthermore, the protocols used by each networking company for communication between switches and the SDN controller differ slightly and are not cross-compatible, so enterprises are forced to lock in to a particular vendor.

Part of the reason SDN-enabled switches are so costly is because they must have a more advanced operating system and CPU than standard switches in order to communicate with the controller and store a cache of routing decision rules. Unfortunately, even with these improvements, an SDN-enabled switch is only powerful enough to query the controller for so many routing decisions per second before its processing speed becomes a performance bottleneck [2]. Of course, one solution is to configure coarse-grain routing rules instead of more expressive per-host, per-protocol rules, thereby increasing the likelihood of a local rule cache hit and decreasing the need for communication with the controller. However, this trade-off defeats SDN's goal of providing IT technicians with arbitrary control over the traffic in their networks.

Since an enterprise's workstations already have powerful CPUs and large amounts of RAM compared to switches, if an SDN could be implemented as a piece of software that would execute on each endpoint, it could save the organization a lot of money. However, there are intuitive reasons why SDN has traditionally been implemented in a switch-based fashion. In a standard network configuration, switches are positioned between multiple computers and other switches, which means they maintain many simultaneous Ethernet connections. Therefore when a switch receives a packet, any decision made about which outbound port to send it through has a direct impact on the path the packet takes to reach its destination. Conversely, hosts are typically connected to the enterprise network via a single Ethernet cable, so there is only one direction they can send traffic. It has been assumed that if an SDN infrastructure based on endpoint-controller communication were implemented, it would not have nearly the same level of routing control as a traditional switch-based

SDN, and thus host-based SDNs have not been explored by the scientific community.

In this paper we demonstrate that a host-based SDN can achieve feature parity with the traditional switch-based design, without costly hardware upgrades. Our system, which we call SHARP, is implemented as a piece of software that endpoints run to selectively encapsulate traffic in an overlay networking protocol of our own design. When combined with a creative use of VLAN tagging, our system is able to provide the same fine-grained routing control as switch-based SDNs. Our host-based SDN design even surpasses switch-based SDN by allowing an enterprise to maintain some level of routing control even after the traffic leaves its own network. To demonstrate the validity and advantages of our system, we implement SHARP as an extension to the PEACE SDN firewall under development at WPI. PEACE is an ideal foundation in that it already has elements of a host-based SDN; the client PEACE program running on each machine intercepts new connection attempts and consults the controller for decisions on whether to block or permit the connection. By incorporating SHARP into PEACE, we allow the firewall controller to not only block traffic, but selectively proxy permitted traffic through an IDS situated virtually anywhere on the network, enabling a more thorough approach to network security than the weak perimeter-based strategy. Our paper makes the following contributions:

- We present SHARP, a host-based SDN design that can achieve the same fine-grain routing control as switch-based SDNs.
- We only use features of legacy networking devices in our design, allowing IT technicians to control traffic flow without hardware upgrades.
- We demonstrate SHARP's applications by incorporating it into the code base of the PEACE Windows SDN firewall.

The remainder of the report is organized as follows. In Section 2, we discuss requisite background knowledge including details about the SDN paradigm, PEACE, and the networking technologies we used to construct our final system. In Section 3, we discuss our final design and implementation of SHARP. Section 4 is an evaluation of our system and in Section 5 we discuss potential security vulnerabilities and future improvements. We finish with an exploration of related work in Section 6 and conclude in Section 7.

2 Background

In this section, we provide an overview of the technologies that we reference and use in the implementation of our project. We first discuss the SDN paradigm and the OpenFlow protocol. We then introduce the architecture of PEACE, the SDN firewall whose functionality we extend in this project. Here, we compare the features of and limitations of PEACE to switch-based SDN networks in order to further motivate our work. Finally, we examine the networking components that we used in our final project design, described in Section 3.

2.1 Software Defined Networking (SDN)

In a traditional network topology, switches and routers have two functions [3]. Firstly, when these network devices receive a packet, they are responsible for determining where next to send the packet. Switches determine which outbound port to send the packet through using ARP caches, whereas routers check an incoming packet's destination address against a table that dynamically maps IP address prefixes to the address of the next router the packet should be forwarded to. At this step, a router may also decide "drop" a packet if it is configured to act as a firewall. The router drops the packet by refusing to forward it on to the next way point. Once the device has decided on the correct port or interface to send the packet through, it is also responsible for physically moving the packet to that port or interface. This involves recording the packet bits from the inbound port into memory, and then writing the stored data "onto the wire" that is connected to the chosen outbound port. Thus, traditional network devices manage both the data plane (movement of data) and the control plane (formulation of routing decisions).

Software-defined networking is a paradigm of computer networking that decouples the control plane from the data plane [4, 5]. A network device that supports SDN remains in charge of physically moving the packet from one port to another; however, it provides an interface for routing decisions to be offloaded to a separate program. When the data plane of an SDN-supporting network device encounters a new network flow (identified by a TCP/IP 5-tuple), it queries the control-plane program for a routing decision. The program that provides instructions to the data plane through the provided interface is called a "controller." While the controller could very well run on the network device itself, the power of the SDN paradigm is that it allows the controller to run on a remote machine, and this is often how SDN networks are configured. Furthermore, it is common for one controller to manage multiple network devices simultaneously.

It is the controller's job to dynamically instruct the devices under its management on how to generate forwarding or routing tables. This type of infrastructure allows a centralized network policy to be realized on the decentralized network topology. Each controller-level routing rule maps one or more attributes of a connection (such as source IP address or transport protocol) to a routing decision. The routing decision may be that packets matching the criteria must be dropped or it may specify which outbound port to use for forwarding relevant packets. Each network device that supports SDN has a local cache of the controller's routing rules which it uses to decide how to handle incoming packets. However, when it encounters a packet that does not match any of the rules in its local cache, it must send the packet to the controller to receive a new routing decision. In this way, a network administrator with just a single controller and a few well-placed SDN-enabled switches can easily orchestrate the flow of traffic for security and load balancing throughout an entire network.

The developers of popular SDN controller programs also provide APIs for changing controller behavior and requesting network statistics [3, 6]. Importantly, this provides a means for converting functionality that has been traditionally reserved for network middleboxes into modular plug-in programs that can be run on the controller machine as needed. Instead of having separate physical firewalls, load balancers, IDS, and network address translation (NAT) devices all in one network, each of these can be replaced by software modules running on a single controller machine. Thus, the main benefits of an SDN are greatly-increased control over the movement of data within a network,

centralization and simplification of network management, and a framework for writing software to influence traffic routing. Today, SDNs are primarily used in enterprises and data centers.

2.1.1 The OpenFlow Protocol

The OpenFlow protocol, first developed at Stanford University in 2008, is a networking protocol designed to enable SDN infrastructure [7]. Although the concept of SDN networks existed prior to OpenFlow, it was the first major protocol designed to standardize the method of communication between a controller and the network devices it manages [8, 9]. In order to communicate using OpenFlow, a network device needs a program called an "OpenFlow agent" to send and receive messages with the controller, as well as modify the device's forwarding or routing tables as instructed by the controller. Today, switches from many leading network device companies such as Cisco, Juniper, and Arista are advertised as "OpenFlow-enabled," meaning that they ship with an OpenFlow agent installed. However, as we mentioned in Section 1, each network device company implements its own variant of the OpenFlow protocol, meaning that two switches from different vendors cannot communicate with the same SDN controller.

2.2 PEACE

PEACE is a next-generation distributed firewall under development at Worcester Polytechnic Institute [10]. The design of PEACE draws from the 2017 paper *DeepContext: An OpenFlow-Compatible, Host-Based SDN for Enterprise Networks* [11]. The authors of DeepContext make the observation that network administrators have access to a limited amount of flow information for configuring policies on the controller in an SDN network built around OpenFlow-enabled switches. A switch is simply an intermediary for other devices' packets; since the controller receives OpenFlow packets from the switch, it can only make routing decisions based on metadata contained in those packets' protocol headers (e.g. MAC addresses, IP addresses, transport protocol ports, etc.). However, when creating routing policies, network administrators may find useful some additional contextual information about a connection, such as the path to the program binary initiating the connection, the program's process ID, and the ID of the user running the program. Only the endpoint that initiated the connection can provide this type of contextual information. DeepContext's main contribution is the Linux implementation of system for providing host-level flow context to a controller in a host-based SDN infrastructure. Briefly, the authors accomplish this by creating a context tracker program and modifying an OpenFlow agent to interact with this program before sending an OpenFlow routing decision request to the controller. A diagram of the DeepContext system-level design can be found in Figure 1.

At a high level, the PEACE SDN firewall is designed similarly to DeepContext. A PEACE client runs on each machine in the network and intercepts new inbound or outbound connection attempts made to or from its host machine. The metadata values for a new connection are compared against a local cache of routing decision rules. When a new connection does not match any rules in the local cache, the PEACE client creates an OpenFlow packet with additional context information and sends it to its assigned PEACE controller for a routing decision. Network administrators can configure the firewall rules in the PEACE controller through a web console.

PEACE is built on the premise that the host-level flow context introduced by DeepContext would enable network administrators to create more detailed (and by extension stronger) firewall rules to protect an enterprise network from exploitation. For instance, to protect against JavaScript-related attacks, a network administrator may want to only allow employees to make HTTP requests if they are using the most recent version of `Google Chrome`, and only if the `--disable-javascript` option is enabled. Or, to prevent malware from being downloaded or spread surreptitiously, perhaps connections should only be allowed if the user has clicked the mouse in the last second. With knowledge of host-level contextual information, these types of fine-grained firewall rules can easily

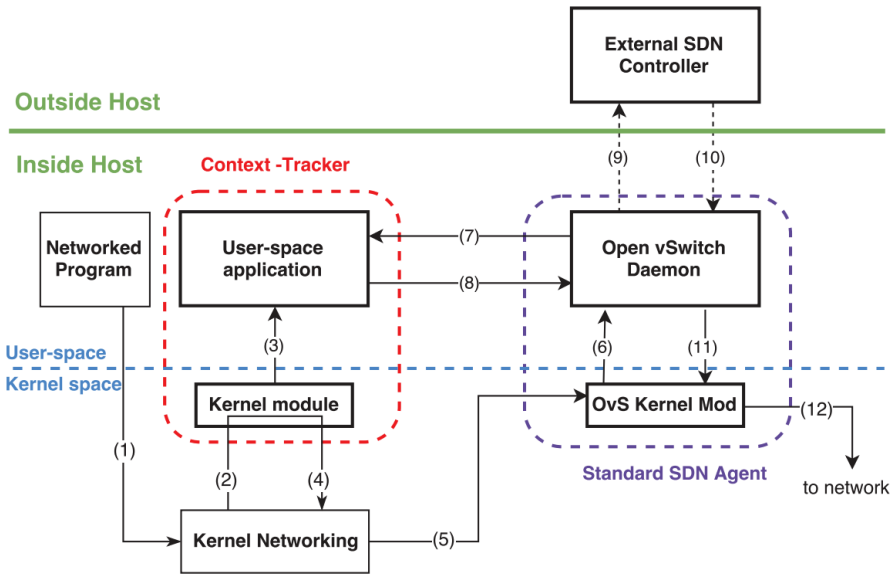


Figure 1: When an application makes a network system call, the Context-Tracker kernel module collects relevant flow context information and stores it in a user-level application. When the first packet for a new flow is intercepted by the Open vSwitch (OVS) OpenFlow agent and the program determines that a controller routing decision is required, the modified OVS agent communicates with the user-level Context-Tracker application to obtain the appropriate context information. That information is appended to the end of the OpenFlow header, and the request is sent to the controller.

be configured. When combined with an SDN architecture, it becomes simple for administrators apply these powerful rules to an entire network from a centralized location.

In addition to increased precision in firewall rules, PEACE provides another benefit over traditional network security configurations. In Section 1 we explain that when a firewall is placed at the boundary of a network, it is unable to filter intra-network traffic because the packets can reach their destination without travelling through it. If an external virus is able to pass through these security controls unnoticed or is introduced out-of-band to an internal host (e.g. via USB disk), it can spread quickly through the network and the firewall is helpless to stop it. Since the PEACE client runs on each host in the network and receives routing decision directly from the controller, PEACE provides network administrators with absolute control over all tentative connections made or received by PEACE-enabled machines, regardless of whether the other endpoint is internal or external.

2.2.1 The PEACE Client

DeepContext was written for the Linux operating system. Since PEACE is designed for commercial use, it is a re-implementation and extension of the DeepContext design in Windows and Mac OS X. For our project, we focus entirely on the Windows implementation of the PEACE client.

A PEACE/DeepContext-style client program has the following main components:

1. **Packet Manipulation Mechanism**

   The client program must be able to actively monitor packets as they are created and processed on the operating system's networking stack. This includes processing all packets in order to recognize when a packet is the first in a new connection and having the ability to block arbitrary packets from being sent or received.

2. **OpenFlow Agent**

   When the client's local cache of rules fails to provide a routing decision for a new connection, the client program must be able to create an OpenFlow packet with the relevant information, send the packet to a controller, and receive and process the controller response.

3. **Connection Table**

   The client program should maintain a data structure that holds all ongoing connections, and the routing decision associated with each one. This allows the packet manipulation mechanism to quickly make routing decisions for all subsequent packets belonging to a new connection.

4. **Context-Tracking Subprocess**

   Part of what makes the PEACE/DeepContext design special is that it provides the controller with host-level flow context for use in routing decisions. In order to provide contextual information, it must be collected from operating system's internal structures and may need to be stored temporarily before it is appended to the OpenFlow packet.

5. **Inter-Process Communication**

   If the previous components are discrete processes, they need a method of communicating context data and routing decisions to each other.

For multiple reasons, the Linux implementation of this design is much simpler than its Windows counterpart. Firstly, DeepContext uses the Open VSwitch (OVS) software, which operates as a full-fledged OpenFlow agent out of the box. On its own, the OVS software performs a majority of the design's networking functionality, including intercepting packets and determining when they belong to a new connection, communicating with the OpenFlow controller to receive routing decisions, and blocking or rerouting packets. When development first began on PEACE, an OVS implementation for Windows did not exist. Although one exists today, it is heavily tied to the Windows Hyper-V virtual machine (VM) manager [12]. Incorporating the Windows OVS into PEACE would require end-users to install Hyper-V and run all their applications in a VM, making the system impractical. Thus, the PEACE developers have opted not to use the OVS software, and have instead written their own OpenFlow agent.

The authors of DeepContext also had access to the Linux NFQUEUE library, which allowed their user-space context-tracking application to intercept packets as they were being processed by the kernel. Without an equivalent user-space library in Windows, PEACE's packet manipulation component is implemented at the kernel level as a device driver using the Windows Filtering Platform (WFP), which we will discuss in detail in Section 2.2.2. At the kernel-level, developers lose many of the memory management abstractions provided by the operating system and code runs at a very high priority level, meaning that page faults and other mechanisms that cause the process to block will crash the operating system. Thus, the process of developing and debugging PEACE had significant challenges that were not present in DeepContext.

2.2.2 Windows Filtering Platform

Before explaining the implementation of the PEACE client in detail, we must explain how the Windows Filtering Platform operates. The WFP is a set of API functions and system services that enable driver developers to intercept and read, modify or block packets at multiple layers of the Windows networking stack [13]. The WFP main page in the online Windows Dev Center explicitly specifies that the WFP can be used to implement various types of security middle-ware such as firewalls, IDSes, and anti-virus programs.

In order to manipulate packets as they traverse the kernel's networking stack, a developer creates a callout object and registers it with the WFP filter engine. The filter engine is the core of the

WFP and orchestrates the delivery of packets to the functions associated with each callout for processing. The filter engine maintains a set of packet filtering layers, that each have an identifier associated with them to indicate the stage of the networking stack at which they exist. Many of the filtering layers correspond with particular sections of the TCP/IP 4-layer reference model, such as `FWPM_LAYER_INBOUND_IPPACKET_V4` or `FWPM_LAYER_INBOUND_TRANSPORT_V4`, which are used to intercept inbound IPv4 packets at the network and transport layers respectively. During callout registration, the developer must specify one of the filtering layer identifiers as a parameter. When a packet is being processed by the networking stack and reaches one of the filtering layers, the filter engine triggers all of the callouts registered with that layer in order.

A callout object is comprised of an identifying GUID number, a set of flags, and pointers to three functions: the `classifyFn`, `notifyFn`, and `flowDeleteFn`. Both PEACE and our project focus primarily on the use of `classifyFn` functions. A `classifyFn` function is used to decide whether to permit a packet to continue to its destination (internally or externally) or block it. It may additionally decide to modify the contents of the packet; however, we will discuss the packet modification process at greater length in Section 2.2.3.1. In order to qualify as a `classifyFn`, the WFP filter engine expects the function to have a specific set of arguments, which will be filled out by the engine at each invocation. The `FWPS_INCOMING_VALUES0* FixedValues` and `FWPS_INCOMING_METADATA_VALUES0* MetaValues` parameters are arrays that hold header fields and packet metadata. Depending on the TCP/IP stack layer the callout is registered at (stream, transport, network, Ethernet), certain pieces of information may or may not be available. For instance, a callout registered at the `FWPM_LAYER_INBOUND_TRANSPORT_V4` layer cannot expect to find the destination IP address stored in these arrays. In this way, the WFP attempts to enforce the separation of the layers in the TCP/IP model. In a `classifyFn`, the `VOID* LayerData` parameter is a pointer to the `NetBufferList` struct that holds the packet itself. Finally, the `FWPS_CLASSIFY_OUT0 * ClassifyOut` parameter is a struct pointer used by the function to declare its decision regarding whether the packet should be permitted or blocked. The struct holds a `actionType` field where the decision is specified, and a `rights` field to indicate whether the packet has the right to alter the packet's `actionType`. The `FWPS_CLASSIFY_OUT0` struct also has a `flags` field, in which the `FWPS_CLASSIFY_OUT_FLAG_ABSORB` flag can be set. If the flag is not set and the packet is blocked, the packet will be subject to event logging and auditing. Setting the flag will allow the packet to be dropped silently, which is useful when performing packet modification.

To register a callout object, a developer first calls `FwpmEngineOpen()`, which returns a handle for further interaction with the WFP filter engine. After starting the registration transaction using `FwpmTransactionBegin()` and adding a new sublayer using `FwpmSublayerAdd0()`, the developer performs the registration by calling `FwpsCalloutRegister()` and `FwpmCalloutAdd()`. To specify the filtering layer at which the callout be triggered, the developer must call `FwpmFilterAdd()` with the corresponding identifier. Finally, the developer calls `FwpmTransactionCommit()` to commit the updates to the filter engine.

We mentioned above that `classifyFn` functions receive packets as `NET_BUFFER_LIST` structs. Each `NET_BUFFER_LIST` holds a linked list of `NET_BUFFER` structs. Each `NET_BUFFER` struct in turn holds a linked list of Memory Descriptor Lists (MDLs) called an MDL Chain. An MDL is simply a data buffer use to store a portion of the packet itself. We believe the purpose of having a chain of MDLs is to allow for large packets to be stored in non-contiguous blocks of memory; however, we are unsure as to the motivation behind designing the `NET_BUFFER_LIST` struct as a list of lists, instead of a single list of MDLs. Regardless, understanding the composition of a `NET_BUFFER_LIST` is important for performing inline packet modification. The proper protocol is to first copy the contents of the original `NET_BUFFER_LIST`'s MDL chains sequentially into a separate buffer. The data stored in the buffer can then be directly modified as needed, and converted into a new `NET_BUFFER_LIST` struct. The function should set the `ClassifyOut->actionType` field to `FWP_ACTION_BLOCK`, and the `ClassifyOut->flags` field to `FWPS_CLASSIFY_OUT_FLAG_ABSORB`. Finally, the function should inject the newly created `NET_BUFFER_LIST` struct into the networking stack using an WFP injection

function. This procedure essentially replaces the original packet with a modified deep copy of itself, and prevents the filter engine from logging that the original packet was dropped.

A WFP injection function is used by a callout's `classifyFn` to re-inject a modified or pended (deferred decision) packet into the Windows network stack [14]. The names of injection functions indicate which TCP/IP stack layer the packet will be injected into, and whether it will be injected on the send or receive path of the stack (e.g. `FwpsInjectTransportReceiveAsync0()`). Since injecting a packet using these functions will cause the callout to be triggered again, `classifyFn` functions should always call the `FwpsQueryPacketInjectionState0()` function to check whether the packet has previously been injected to avoid infinite loops. Depending on which TCP/IP layer injection function is being called, the list of parameters varies; however, some are consistently required. A developer must first provide a `HANDLE injectionHandle`, which has been created by calling `FwpsInjectionHandleCreate0()`. The injection handle used here will also be used when calling `FwpsQueryPacketInjectionState0()` to determine whether a packet has previously been injected. Injection functions also require a `NET_BUFFER_LIST* netBufferList` parameter, which holds the packet to be reinjected. Due to the asynchronous nature of packet injection, developers must pass an injection function a `FWPS_INJECT_COMPLETE0 completionFn`, which is a pointer to a function that will be called once the packet injection is complete. Here a developer will do any necessary dereferencing and buffer freeing associated with the packet injection. Finally, an injection function takes a `HANDLE completionContext` parameter, which holds a pointer to any additional information that should be passed to the `completionFn` function. The WFP filter engine will fill the `completionFn` function's `VOID *context` parameter with the pointer passed in the `completionContext` injection function argument.

2.2.3 The PEACE Client: Under the Hood

Now that we have briefly introduced the relevant parts of the WFP service, we will begin to describe the Windows implementation of PEACE in detail. Overall, the Windows PEACE client is split into three main programs:

1. **The WinSight Driver**

   This kernel driver defines callout functions for blocking or permitting connections based on the first packet in a flow. The driver maintains a table of running processes and their relevant data (PID, UID, etc.), which is combined with 5-tuple flow information extracted from the packet's headers to query the local cache of PEACE rules. The driver also maintains a list of all ongoing connections and connection attempts, along with the routing decision associated with them, so that the decision for re-injected packets can easily be accessed. When a routing decision cannot be made using the local rule set, the WinSight driver elevates the decision to the WinSightService program.

2. **WinSightService**

   This user-space administrator service operates as an intermediary for the various kernel-level and external components of PEACE. Even though the table of process data exists in the kernel driver, in order for the callouts to access its data and make a routing decision, they must send a request for process data to WinSightService, which fetches it for them. Similarly, WinSightService can be called on to fetch context data from the GUI context collector process. In the case that the WinSight driver cannot come to a routing decision using its local rule cache, it must retrieve a decision from the PEACE controller; however, its priority level is too high for direct socket communication. Instead, WinSightService handles encrypting, relaying, and decrypting messages between the driver and the PEACE controller.

3. **The GUI Context Collector**

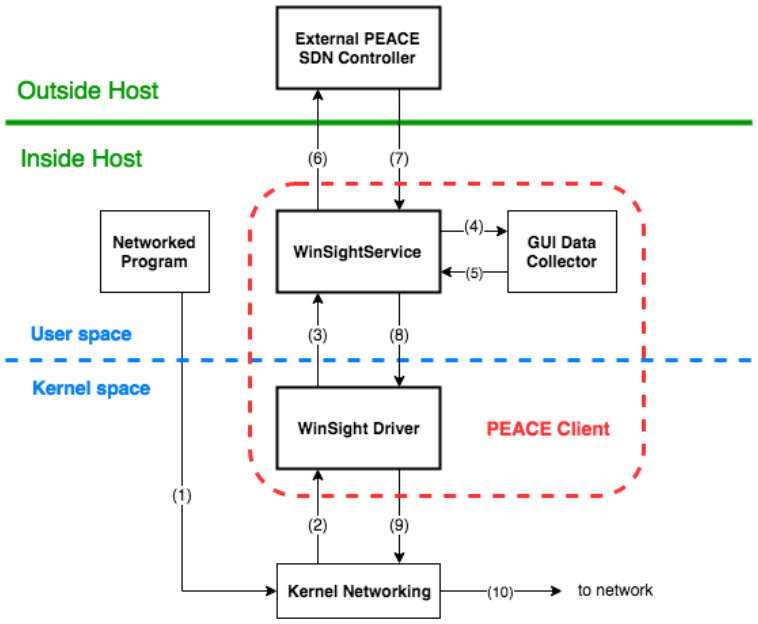


Figure 2: When a program issues a networking system call (1), the WFP filtering engine will determine whether to trigger the ALE-level send path callouts defined in the WinSight driver. If the driver's `classifyFn` function is triggered (2), it attempts to make a routing decision for the flow associated with that packet based on the local rule cache. If no decision can be made, the packet is pended, and a basic OpenFlow packet is created with the packet data. The driver then sends the packet up to the user-space WinSightService program (3), which interacts with a side process (4) to collect the contextual information to be appended to the OpenFlow packet (5). Next, WinSightService sends the OpenFlow packet over an encrypted channel to the external PEACE SDN controller (6). Using its set of firewall rules, the controller decides whether the packet should be blocked or not. It constructs a response OpenFlow packet and sends it to WinSightService (7), which relays the response to the driver (8). If the packet should be allowed, the driver re-injects the packet into the kernel networking stack (9), and the packet is sent on to the network (10).

This user-space program fulfills requests from WinSightService for GUI context data (mouse/keyboard events) related to a particular flow. Although a key part of what makes PEACE unique, the GUI context collector process and its implementation are not relevant to this project and thus is not discussed in detail.

The general flow of packet processing in the PEACE client is presented in Figure 2. Simply, when one of the WinSight driver callouts encounters the first packet for a new connection, the `classifyFn` function pends the packet and makes a request to WinSightService for process and GUI data. With the header information from the packet and the supplied context data, the driver attempts to decide whether to permit the connection or block it using the local cache of rules. If a decision cannot be made, the driver makes another request to WinSightService to send an encrypted OpenFlow packet to the PEACE controller to receive an authoritative routing decision. WinSightService forwards the decrypted response to the driver, which will update the appropriate entry in the connection list with the routing decision, and will re-inject the pended packet if the response was to allow the connection. Now, we will describe how these programs implement the five components of the PEACE design as described in Section 2.2.1.

2.2.3.1 Packet Manipulation Mechanism

As we discussed earlier, the Windows implementation of PEACE uses the Windows Filtering Platform (WFP) kernel-level service to intercept packets and permit or block them. The PEACE driver makes extensive use of a special WFP layer type called Application-Layer Enforcement (ALE), which is specifically designed for flow-level processing [15]. Callouts registered with ALE filter identifiers will be triggered once per new connection, which makes them ideal for implementing a firewall. For TCP traffic, outbound ALE-level callouts will be triggered on a packet generated as a result of a program calling the `connect()` function, while inbound ones will be similarly triggered on a call to `accept()`. For UDP traffic, the ALE level considers a packet as belonging to a "new" connection when it is sent to or received from a new, unique IP address and port tuple.

The PEACE driver has four ALE-level callouts for permitting connection establishment. The `ConnectV4` and `ConnectV6` callouts authorize outgoing connection attempts for IPv4 and IPv6 traffic respectively, while `ReceiveV4` and `ReceiveV6` perform the same function for incoming connection attempts from remote hosts. Although each callout has a separate `classifyFn` function, they all operate in the same fashion. The `classifyFn` functions first check for trivial cases; for instance, re-injected packets and packets on local loop-back should be automatically permitted. After trivial checks have been exhausted, the local and remote IP address / port pairs are extracted from the `FixedValues classifyFn` parameter and stored in `REGULARIZED_ADDRESS_PORT` structs. Additionally, the function locally stores the `transportEndpointHandle` field from the `MetaValues classifyFn` parameter. This field is a handle to the local socket through which the packet that triggered the callout will be sent. The two `REGULARIZED_ADDRESS_PORT` structs and the `transportEndpointHandle` are then passed to the `CheckRemoteEntry()` function, which returns the flow-level routing decision. If the packet belongs to a flow that does not yet have a routing decision (due to the comparatively long process of communicating with the PEACE controller over the network), then the returned decision is to pend the packet. The OpenFlow Agent component of the PEACE driver will perform re-injection of pended packets should the controller decide to permit the connection.

2.2.3.2 OpenFlow Agent

In Section 2.2.1, we defined the OpenFlow agent component specifically as the mechanism for communicating with the controller via the OpenFlow protocol. In the Windows PEACE client implementation, packet manipulation is performed by the WinSight driver via the WFP; however, because the driver is situated in kernel space, it cannot directly communicate with the controller using a socket as a user-space application might. Thus, WinSightService was created in part to handle passing OpenFlow messages between the driver and the external controller. When the WinSight driver cannot make a routing decision for a new connection attempt intercepted by one of its WFP callouts, it saves the first packet and encapsulates a copy in a `OFP_PACKET_OUT` OpenFlow packet struct. The resulting struct is passed to WinSightService (for more information on how WinSightService and the WinSight driver communicate, see Section 2.2.3.5). After receiving the struct, WinSightService calls `encryptOpenFlowBuffer()` to perform AES on the struct bytes using functions from the `bcrypt.h` header file. The resulting packet of encrypted data is sent to the PEACE controller via a socket. When a response is received from the controller, WinSightService calls `decryptOpenFlowBuffer()` to decrypt the packet, and the response is passed to the WinSight driver. Based on the `Action` field of the passed `OFP_PACKET_IN` (which contains the controller's decision), the WinSight driver may re-inject the previously saved packet.

2.2.3.3 Connection Table

Although the ALE layer in WFP is designed for per-flow processing, an ALE-level callout is often triggered multiple times for a single connection. This is partially due to the manner in which PEACE

processes packets, but TCP endpoints will also trigger the callouts repeatedly by retransmitting SYN packets to unresponsive remote endpoints or by resetting the connection. It would be very inefficient if the PEACE driver had to query the local rule cache or PEACE controller every time an ALE-level callout was triggered for the same flow, so the driver maintains a table of active connections and their associated routing decisions.

At a high level, the connection table is implemented as a hashmap of `ENDPOINT_ENTRY` structs. An `ENDPOINT_ENTRY` represents a local socket, and stores a copy of the socket's associated `REGULARIZED_ADDRESS_PORT` struct and `transportEndpointHandle`. Similarly, a `REMOTE_ENTRY` struct represents a remote socket that is connected to a local socket, and contains the `REGULARIZED_ADDRESS_PORT` struct for the remote IP address / port pair. A full connection is defined by a `ENDPOINT_ENTRY` and `REMOTE_ENTRY` pair, but since a local socket may have multiple remote sockets connected to it (e.g. a webserver), each `ENDPOINT_ENTRY` maintains a list of one or more `REMOTE_ENTRY` structs. Because each `REMOTE_ENTRY` struct represents a unique connection, it maintains the `LONG Action` field that holds the routing decision associated with the connection.

When an ALE-level connection authorization callout is triggered on a packet, it's `classifyFn` function obtains a decision for the packet (pend/permit/block) by passing the local and remote `REGULARIZED_ADDRESS_PORT` structs and `transportEndpointHandle` associated with the connection to the `CheckRemoteEntry()` function. If an `ENDPOINT_ENTRY` with the corresponding `transportEndpointHandle` does not exist yet (as in when the first packet sent or received over a local socket is intercepted), the driver creates one with a call to `CreateEndpointEntry()`. This function also adds a `REMOTE_ENTRY` for the current connection to the new `ENDPOINT_ENTRY`. Next, driver locates the `REMOTE_ENTRY` for the connection by passing the remote `REGULARIZED_ADDRESS_PORT` struct to `FindRemoteEntry()`. The driver returns the values stored in the `Action` field of the connection's `REMOTE_ENTRY` struct back to the original `classifyFn` function. If the `Action` field indicates that the routing decision needs to be elevated to the controller, the packet is pended and a `OFP_PACKET_IN` packet is sent up to WinSightService (see Section 2.2.3.2).

The connection table finally needs a method for reaping closed connections so that it does not grow too large. In addition to the four connection authorization callouts described in Section 2.2.3.1, the PEACE driver defines two callouts at the `FWPM_LAYER_ALE_ENDPOINT_CLOSURE_V4` and `FWPM_LAYER_ALE_ENDPOINT_CLOSURE_V6` layers. These callouts trigger upon TCP or UDP socket closure for IPv4 and IPv6 traffic respectively. They pass a `transportEndpointHandle` to the `DeleteEndpointEntry()` function, which removes the local `ENDPOINT_ENTRY` (and any associated `REMOTE_ENTRY`s) from the connection table.

2.2.3.4 Context-Tracking Subprocess

The mechanism for the context-tracking subprocess is not relevant to this work, and thus we omit it.

2.2.3.5 Inter-Process Communication

Communication between user-space programs is performed via standard sockets from the `winsock2` library. The communication mechanism between WinSightService and the WinSight driver is more interesting because traditionally, kernel drivers exist to offer services to user-space applications. In the Windows PEACE implementation, the roles are reversed; the driver program needs to make requests to an application in user-space, namely WinSightService. Unfortunately, a straightforward method for a driver to initiate communication with a user-space application does not exist. For one, this is because kernel programs typically run at a very high priority level and their ability to make blocking function calls is often highly restricted. Furthermore, the OS a cannot rely on a particular third-party user-space application existing on a given machine, since they can be installed and deleted at whim.

The solution devised by the PEACE developers was to re-appropriate the Windows Driver Framework (WDF), the existing service Windows offers for drivers to receive and fulfill requests from user-space [16]. A typical use of the WDF request API is as follows. By calling `WdfIoQueueCreate()`, a driver instantiates a `WDFQUEUE` for storing pending I/O requests that the system sends to the driver, in this case on behalf of a user-space application. Using `WdfIoQueueRetrieveNextRequest()`, the driver can get the next request from the queue, which is stored in a `WDFREQUEST` struct. To get the contents of the returned request, a driver calls `WdfRequestRetrieveOutputBuffer()`. Once the driver has finished processing the request and performing any additional actions, it loads a response into the `WDFREQUEST` struct using `WdfRequestSetInformation()`. Finally, the request response is returned to the OS using one of a few variants of the `WdfRequestComplete()` function.

In the typical scenario, a user-space application that wants to make requests to a driver first calls `CreateFile()`, specifying the name of the device associated with the driver, and receives a `HANDLE DeviceHandle` for future communication with the driver. Next, `CreateIoCompletionPort()` takes the `DeviceHandle` and returns a `HANDLE CompletionPort` struct for notifying the application of asynchronous completion of I/O requests. In order to create a request, the application calls `DeviceIoControl()`, using the `DeviceHandle` and a pointer to a buffer containing the data the driver will need to complete the request. Finally, by using the `CompletionPort` struct, the application can call `GetQueuedCompletionStatus()` to retrieve the I/O request response that was sent by the driver using `WdfRequestComplete()`.

As we mentioned earlier, the PEACE Windows implementation requires an inverted version of this system. To put it simply, the PEACE developers accomplished this by causing WinSightService to send a group of empty requests to the driver immediately after the connection to the driver's device is established. The driver holds onto these empty `WDFREQUEST` structs until it wishes to make a request to WinSightService. In this way, calls made to `WdfRequestComplete()` by the WinSight driver indicate submitting requests instead of responding to them, and calls made to `DeviceIoControl()` in WinSightService indicate responses to those requests.

Now that this mechanism enables inverted communication between the WinSight driver and WinSightService, the two components need a way to communicate the type of request or response that each sent `WDFREQUEST` represents. In the Windows PEACE system, the input and output buffers of any call to `WdfRequestComplete()` or `DeviceIoControl()` are filled with a `WINSIGHT_QUERY_DATA` struct (defined by the PEACE developers). The fields of this struct include another struct called the `WINSIGHT_REQUEST RequestFlags`, which defines a set of flags such as `IsOpenFlowPacket` or `CollectGUIData`; setting the values of these flags is the main method by which the WinSight driver and WinSightService communicate. Since multiple request and response types require additional information, the `WINSIGHT_QUERY_DATA` struct also defines a few data buffer fields. For instance, a response to `CollectGUIData` includes the collected GUI data in the `WCHAR ProcessGUIData` buffer, while a `IsOpenFlowPacket` request contains the OpenFlow packet to be sent to the controller in another buffer.

2.3 Relevant Technologies

Here we will give an overview of the various technologies that we incorporated into our final project design, as described in Section 3.

2.3.1 Encapsulation and Tunneling

In computer networking, encapsulation is the operation of wrapping data from one protocol inside of another in order to allow it to continue travelling across a network [17]. For instance, when a IPv6 packet needs to travel from one IPv6 router to another over a network that only support IPv4, a common practice is for the edge IPv6 routers to encapsulate the IPv6 datagram by prepending an IPv4 protocol header that specifies the next IPv6 router as the destination. When the packet arrives

at the next IPv6 router, the IPv4 header is stripped off and IPv6 routing resumes. Although in this case the encapsulation is necessary, often times encapsulation is simply used to ensure a packet reaches a hop-point (such as a proxy server) before its destination. Simple encapsulation protocols include IP-in-IP and the Generic Routing Encapsulation (GRE) protocol.

Tunneling is the process of using encapsulation to secure network communications. In a tunneling scenario, two hosts on separate secure networks wish to communicate with integrity and confidentiality over an unsecured network. Tunneling protocols such as IPSec encrypt and digitally sign IP packets and prepend additional headers before encapsulating everything in an outer IP packet [18]. Virtual Private Network (VPN) client programs create tunnels to remote servers using IPSec or similar protocols in order to hide the origin of users' traffic.

2.3.2 Overlay Networks

Overlay networks describe any computer network that is built on top of a pre-existing network by defining a set of virtual nodes and links that live on top of physical ones. The Internet itself was originally an overlay network because the virtual links between computers rested on top of existing telephone lines. Overlay networks often define a protocol for communication between the virtual nodes, which is encapsulated in the standard protocol for the physical network. In the example of the early Internet, although the engineers defined a protocol for communication between the computers, the packets had to be converted by a modem into electrical signals that could travel over the telephone lines.

An overlay network must somehow store the mapping between its virtual nodes and the underlying physical nodes, as well as the physical links used to make each virtual link. It must also maintain some policy for the addition of new nodes to the network. Overlay networks will also typically employ a policy to dynamically route packets between its nodes, based on some application-specific heuristic (i.e. security or quality of service (QoS)). For example, Andersen et al. [19] create an overlay network for rerouting packets in the event of router or link faults that affect any given node.

2.3.3 Virtual LANs

A virtual local area network (VLAN) is a virtual partition of a computer network at the link layer of the TCP/IP networking stack. Configuring a VLAN is relatively simple, and enables traffic engineering and network-level access control for connected resources. Often times, it is advantageous for network administrators to be able to specify a subset of machines on a LAN, such that the machines in that subset can only communicate with each other; for instance, an enterprise may want to allow hosts connected to a guest Wi-Fi access point to use the Internet, but now allow them to send packets to company servers or workstations. Without using VLANs, the only way to achieve such a setup would be to physically disconnect the LAN into smaller networks. With VLANs, administrators can easily perform these types of logical separations without modifying the underlying physical topology. Overlay networks and VLANs are related in that a virtual network is "overlaid" on top of a underlying physical topology of switches and links. One could consider a VLAN to be a special kind of overlay network that applies to switches in a LAN.

Each VLAN on a network is identified by a unique number called a VLAN ID, which can range from 1 to 4096. To assign a host to a particular VLAN, a network administrator logs into the switch connected to that host, and changes the VLAN ID of the corresponding port (the default VLAN ID for each port is 1, as VLAN 1 is the default VLAN). When a switch receives a packet from one of its ingress ports, it determines the port's VLAN ID and will only forward the packet through egress ports that share the same VLAN ID. Thus, if one host is connected to a switch by a port with a different VLAN ID than another host, the two machines can never communicate. Additionally, since VLANs are configured entirely on switches, endpoints in the network are never aware of their existence.

Using traditional port-based VLAN configuration, each port can only be assigned one VLAN ID, and thus each physical link can only carry traffic belonging to hosts on one VLAN. If network administrators wanted to allow two core switches to pass each other traffic for five VLANs, they would need to connect the switches with five separate Ethernet cables. Today, many switch implementations support the IEEE 802.1 specification, which allow for a single port to support traffic for multiple VLANs through a technique known as "trunking." To perform trunking between switches, each switch designates one of its ports as a "trunk port," which can be configured by the network administrator to accept traffic for a set of VLAN IDs. Because trunk ports can support multiple VLANs, if a switch receives a packet from a trunk port it cannot immediately identify the associated VLAN ID. Thus, when a switch receives a packet from an ingress port and decides to transmit it over a trunk port, the switch inserts a "VLAN tag" into the packet's Ethernet header containing the ID of the associated VLAN. Once the packet is received by the other switch's trunk port, the VLAN tag is removed, and the switch reverts back to port-based VLAN [20].

2.3.4 Spanning Tree Protocol (STP)

In any network of non-trivial size, IT technicians interconnect core switches cyclically to avoid creating a single point of failure. Unfortunately, this type of configuration invariably creates opportunities for packets to loop between switches indefinitely. Switches communicate by broadcasting requests they receive on one port to all other active ports, but do not maintain the state of those requests in memory. If a network has a loop, requests will be continually broadcast and re-broadcast between switches, flooding the network with traffic and crippling it. To solve this issue, the Spanning Tree Protocol (STP) was created. When switches are configured to use STP, they vote to ignore certain network links such that the network becomes a spanning tree, where there is only one possible network path between each pair of hosts on the network [21]. To configure STP, network administrators may select a switch to act as the root node for the spanning tree, or the switches will elect a root node among themselves. The shape of the resulting spanning tree can change dramatically depending on the root node chosen during STP. Importantly, two VLAN IDs can each have their own root nodes, and thus each VLAN can have a unique spanning tree. An example of this is shown below in Figure 3:

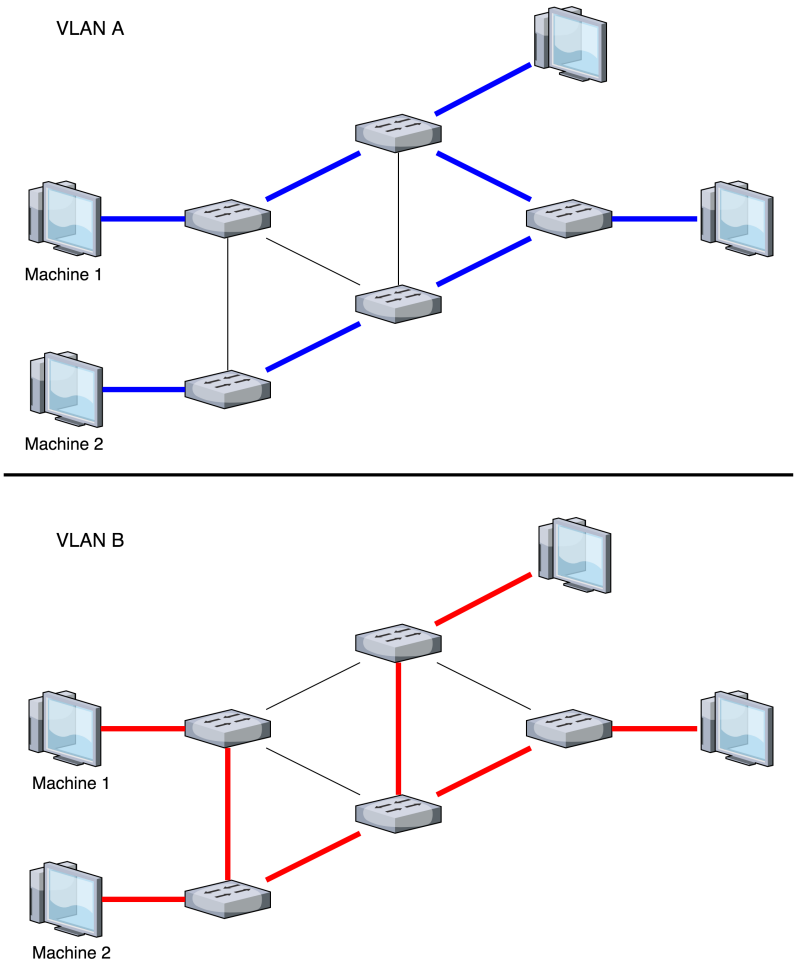


Figure 3: A sample network has two VLANs, A and B. Each VLAN has a unique spanning tree, consisting of the colored network links. Any link that is not colored has been elected by the switches to be disabled. Note that the path traffic must travel for Machine 1 and 2 to communicate is much longer if they are a part of VLAN A than if they are a part of VLAN B.

3 Implementation

In this section we describe the design process and implementation of SHARP, our host-based SDN. We begin with a discussion of the various constraints we had to account for, and some of our early concepts for the system's design that were rejected in favor of our final design. We then explain the final design of SHARP and its operation at a high level. Finally, we spend a majority of the section detailing our experience implementing SHARP as an addition to the PEACE Windows SDN firewall client.

3.1 Design Considerations

In Section 1, we explained that the goal of this project is to allow an endpoint to dictate the route its traffic travels through the network, specifically with the same level of control as a switch-based SDN. Once an endpoint's network interface card (NIC) writes the bits of a packet onto the Ethernet cable, the endpoint loses direct control over the routing of the packet. Therefore, we knew that our final design would involve the host appending additional information to each packet that would communicate its intentions to the other devices involved in transporting the packet to its destination. We also wanted to ensure that SHARP could function without using any features that are not available on legacy switches, since the value of our system is predicated on avoiding networking hardware upgrades. Early on, we pinpointed VLAN tagging as feature of legacy switches that would a valuable design element. As we discussed in Section 2.3.3, in organizations that use VLANs for partitioning the network, a packet's VLAN ID can heavily influence which outbound port a switch decides to forward a packet through. By manipulating the VLAN ID associated with a given packet, we could easily change the route that packet takes through local network.

We also wanted SHARP to allow clients to specify when a packet from their computer should be directed to another machine as an intermediate checkpoint on its way to the destination. In the interest of arbitrary routing control, we designed our system such that a network administrator could specify a virtually unlimited number of intermediate hops. However, there is a good reason why a network administrator would want at least one other machine to read certain network traffic before it reaches its destination. The traditional perimeter-based approach to network security, in which an IDS or IPS is positioned at the network edge, is flawed in that it cannot directly monitor internal connections; malware can be introduced to a network from within (e.g. by a disgruntled employee), and in *Bro: a system for detecting network intruders in real-time*, Paxson outlines multiple strategies for external attackers to sneak malware download traffic past an IDS [22]. Without SDN, a network administrator would struggle to configure the network to allow an IDS to monitor internal traffic without creating a choke point in the network; by designing our system to allow packets to stop at way points, we give network administrators the freedom to place an IDS at any location in their network and selectively route traffic through it.

For a host to send their traffic to a way point, we knew SHARP's final design would need some sort of packet encapsulation. This could involve simply wrapping a packet in an outer set of transport and network level headers, as with IP-in-IP or GRE encapsulation protocols, or a dedicated encrypted tunnel between two way points could be formed, as with the IPSec and L2TP protocols. Early in the design phase of the project we examined the tunnelling protocol approach. We developed PowerShell scripts for automatically setting up an VPN connection using L2TP and a pre-shared key. The script for building a VPN connection is shown here:

```
param(
[String] $Name,  # VPN connection name
[String] $Ip,    # IP address of the VPN server
[String] $Psk,   # Pre-shared key used between server and client
[String] $Uname, # Username for server credentials
[String] $Pword  # Password for server credentials
```

```
)

# If a VPN Connection already exists with the right name, remove it since it might be
    misconfigured
if ((Get-VpnConnection).Name -eq $Name) {
    Remove-VpnConnection -Name $Name
}

# Add the VPN Connection
Add-VpnConnection -Name $Name -ServerAddress $Ip -TunnelType L2TP -L2tpPsk $Psk
    -SplitTunneling -Force -PassThru

# If the VPN Connection is not already connected, connect using credentials received from
    controller
if((Get-VpnConnection -Name $Name).ConnectionStatus -eq "Disconnected") {
   rasdial $Name $Uname $Pword
}
```

The idea behind the VPN tunnel approach was that when the SDN controller responds to a host machine's routing decision query, it would somehow specify a chain of existing VPN tunnels for the packet to travel through. For many reasons, we opted not to use this approach for SHARP's final design. Firstly, relative to other computer operations, building and tearing down VPN connections is a time-consuming process. For efficiency, each computer would have to maintain only a handful of tunnels and they would have to be relatively static; this would severely limit the degree of routing control our design would offer. Secondly, communicating the state of all VPN tunnels on the network to the SDN controller so that it would have an accurate real-time picture of the network topology was a daunting engineering task given the time frame of the project. This task's difficulty is increased by the fact that VPN tunnels can be created, modified, or disabled by end users directly through the Windows OS GUI, so a given tunnel's status could easily change independently of our system. Additionally, we wanted our design to operate as an overlay network, such that any machine running SHARP could be specified as a way point; if we took the VPN tunnel approach for packet encapsulation, each SHARP machine would need to run a VPN server such as OpenVPN or StrongSwan, which would add dependencies and make installation painful. Finally, we wanted to incorporate the entire SHARP implementation directly into the PEACE code base, and so calling a set of scripts was simply an unattractive option.

3.2 Final Design

We will now describe in detail the final design of SHARP, our host-based SDN system. SHARP provides routing control at the data-link and network layers of the TCP/IP model (Layers 2 and 3, respectively) by encapsulating packets in a SHARP header that includes information about the network route the packet should travel. Routing control at the network layer consists of redirecting a outbound packet to a series of 0 or more intermediate machines, which forward the packet to each other before finally delivering the packet to its original destination. For a machine on the network to properly parse and make decisions based on incoming packets with SHARP headers, they must also have an PEACE client program installed. It follows that the set of SHARP-enabled machines on a given LAN form an overlay network, and the only hosts that can be specified as intermediate way points in the route of a packet with a SHARP header are hosts that are part of the overlay network. Henceforth, endpoints that a part of the SHARP overlay network will be referred to as nodes.

For enabling host-based routing control at the data-link layer, SHARP relies heavily on use of VLAN IDs to influence the decisions of switches. In a network using SHARP, a set S of VLANs

is configured such that each VLAN in the set represents a unique spanning tree for the network, and such that there is a VLAN ID to describe any possible non-looping network path between two SHARP nodes. Any VLAN ID's spanning tree describes a set of $n!/(2!(n-2)!)$ paths between hosts, where n is the number of hosts on the network. Furthermore, for a given host, each VLAN's spanning tree describes $n-1$ paths as there are $n-1$ other hosts on the network. Thus we say that one network path between two SHARP nodes is defined by the tuple *(IP ADDR 1, IP ADDR 2, VLAN ID)*. In this way, simply by modifying the VLAN ID of an outgoing packet, a node can completely change the layer 2 route the packet travels to reach its destination.

For a host to efficiently modify its VLAN ID on a per-packet basis requires the insertion of VLAN tags (see Section 2.3.3). Accordingly, in SHARP each node is connected to the network via a trunk port, configured on the switch to accept all VLAN IDs in S. This use of VLANs as a way for endpoints to modify the path of their own traffic is innovative. In modern network, VLANs are typically used to separate hosts. Each host is typically associated with a single VLAN via an access port, and as a result it cannot tell what VLAN it is a part of. Additionally, trunk ports have historically been reserved exclusively for linking switches together, and as such there is little to no support in mainstream computer operating systems for appending VLAN tags. Because of this, as we will discuss later in this section, our system involves a kernel-level program for appending VLAN tags to packets on the send path of the networking queue.

At a high level, SHARP operates as follows. A SHARP node initiates a new connection to a remote IP address. In typical SDN fashion, the first outgoing packet for that connection is pended by a SHARP client program at the network layer, and a local rule cache is consulted to reach a connection-level routing decision. If no local rule matches the connection attributes, the client program sends an OpenFlow packet to the controller to receive an authoritative routing decision. So far this accurately describes the function of the PEACE SDN firewall. Where the two systems differ is in the nature of the OpenFlow response; as a firewall, PEACE is concerned with the binary decision of approving or denying a packet. In the SHARP SDN design, the response from the controller is a route through the network that packets in the connection should follow. A route through the network is represented as a chain of n IP addresses (starting with the source address and ending with the destination address), and the $n-1$ VLAN IDs describing the Layer 2 route a packet should travel between each pair of nodes. Upon receiving a route for a given connection, the SHARP client stores it in a table that maps connection 5-tuples to routes so that the rule cache and controller do not need to be consulted for future packets in the connection flow.

Upon determining the correct route for an outgoing packet P to follow, the SHARP client program encapsulates the packet in a SHARP header H that contains its route (creating packet HP). Based on the route header, the SHARP client determines the next node in the chain to send HP to, and the proper VLAN tag ID to insert in the packet's Ethernet frame header. The next node receives HP, and similarly parses the header to find the next way point and VLAN ID.

The host specified by the final IP address in the chain may or may not be a SHARP node. If the destination is a node, it will simply receive HP, strip the SHARP header off, and inject P into its own inbound networking path so that it reaches the proper application. The receiving SHARP client will also store the reverse of the received packet's SHARP header (H^{-1}) so that it may append it to outgoing packets P^{-1} that are responses to P. If the destination is not a SHARP node, the last node before the destination is responsible for proxying the connection. The proxying node stores H^{-1}, and rewrites P's source IP address to be its own IP address. By doing this, when the destination receives P, P^{-1} packets will be delivered to the proxying node instead of P's original source node. Once P^{-1} is received, the proxying node rewrites it's destination address to be that of the source of the original packet, appends H^{-1}, and starts $H^{-1}P^{-1}$ on its journey through the reverse route to the original sender of P.

3.2.1 The SHARP Header

Here we describe the fields of the protocol header that the SHARP client appends to outgoing packets before redirecting them to a SHARP node. As we discussed in Section 3.2, the SHARP header is responsible for carrying routing information between an endpoint running SHARP and an arbitrary destination. The header is thus responsible for both the course-grained Layer 3 SHARP node path and the fine-grained Layer 2 path which compose the nodal links within the chain of way points. It follows that the SHARP header must maintain both Layer 3 information (namely, IP addresses) and Layer 2 information (namely, VLAN tags) to sufficiently represent the chain. It must also hold metadata to properly traverse and interpret the chain targets. With these considerations in mind, the following diagram illustrates the SHARP header design.

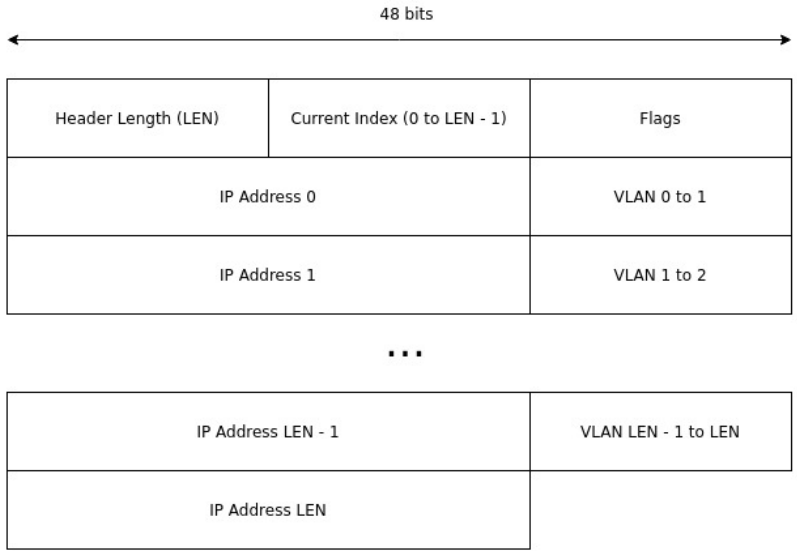


Figure 4: SHARP header specifications.

The SHARP header starts with several metadata values. Each value is described below:

- **Header Length**: Total number of SHARP nodes in the path. Note that the total number of IP addresses in the route will not be equal to the value of this field if the destination is not a SHARP node.
- **Current Index**: Specifies the index of the current node in the SHARP route. Updated by each node as the header is processed. We opted to use C-style indexing in the SHARP header, such that while the packet is at *IP Address 0*, Current Index is equal to 0, and so on.
- **Flags**: A set of flags indicate characteristics of the SHARP route. Currently, two flags are used:
  - **DestinationFlag**: Indicates whether the final node in the route is a SHARP node. The flag bit is 0x01 and the flag is asserted if the destination is a SHARP node.
  - **SendOrRecieveFlag**: Indicates whether the chain represents an original outbound connection or a response to a prior outbound connection. The flag bit is 0x10 and the flag is asserted if the chain represents a response connection.

The remainder of the header consists of IP addresses and VLAN tags. Each VLAN tag corresponds to the connection between the SHARP nodes whose IP addresses precede and succeed it

in the header. For example, *VLAN 0 to 1* represents the Layer 2 connection between the SHARP nodes with *IP Address 0* and *IP Address 1*. By using the current index value, the kernel modules in a given node can retrieve the IP address of the next node in the chain and the VLAN tag associated with the Layer 2 path between those them. It can then organize a new packet to traverse to the next node. The current index value is updated as the packet moves between the nodes that compose the SHARP chain; each node increments the value once it processes the packet.

The composition of the SHARP header also varies based on the destination flag. If the flag is set, then the final destination of the packet is a SHARP node. As a result, the final destination is accounted for when calculating header length field. For a header length value of n, n IP addresses and $n-1$ VLAN tags will exist in the SHARP header. Once the current index reaches $n-1$, the node will determines that it is the intended receiver. It removes the payload from the SHARP packet and sends the payload to its own application layer. However, if the flag is not set, then the final destination of the packet is not a SHARP node. As a result, the final destination is not included when calculating the header length field. For a header length value of n, $n+1$ IP addresses and n VLAN tags will exist in the SHARP header. Since the final IP address corresponds to a non-SHARP node, then the VLAN tag field prior to it is meaningless and is filled with a dummy value. Once the current index field reaches $n-1$, the final SHARP node knows that it must proxy a connection between the true source node and final non-SHARP destination. Details on the proxy behavior can be found in Section 4.x.

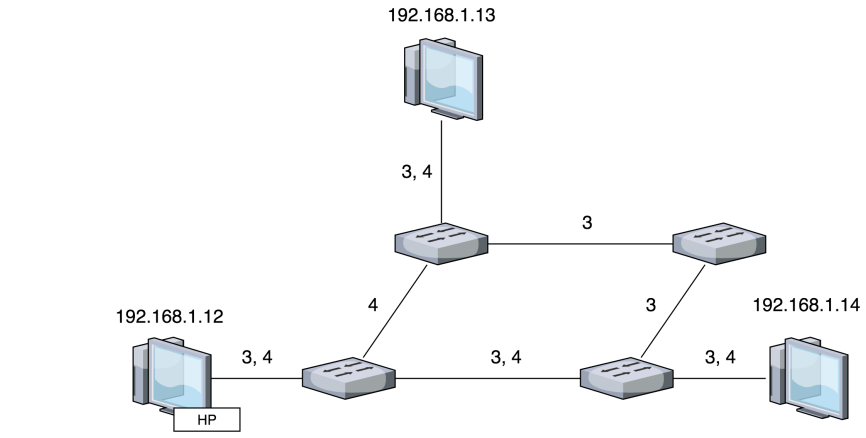
At first it may appear counter-intuitive that we designed the SHARP header such that the *Header Length* field does not always equal the number of IP addresses in the chain. However, in doing so, we gave the SHARP header a useful property; whenever a node parses the SHARP header and finds that $HeaderLength - 1 = Index$, then it knows that the SHARP header has to be stripped off to reveal the inner payload and perform a special function. This happens regardless of whether the *DestinationFlag* or *SendOrReceiveFlag* are asserted. Once the SHARP header has been stripped, the SHARP node can read the flag values to determine what must be done next with the inner packet. This property greatly simplifies some of the logic involved in processing a SHARP packet.

3.2.2 Network-Level Demonstration

We will now walk through two examples of the network-level SHARP header processing that occurs to transport a packet to its destination through the specified route. We remind the reader that the *DestinationFlag* bit is 0x01 and the *SendOrReceiveFlag* bit is 0x10, so a value of 3 in the *Flag* field of the SHARP header indicates that both flags are asserted.

In the first example, starting with Figure 5, the *DestinationFlag* is asserted and so no connection proxying is required. The network includes VLANs with IDs 3 and 4, and all three machines are connected via trunk ports that accept these VLAN IDs.

1. An outbound packet P destined for 192.168.1.14 is intercepted, and a SHARP header H with a length of 3 is appended to it to create HP. See Figure 5.

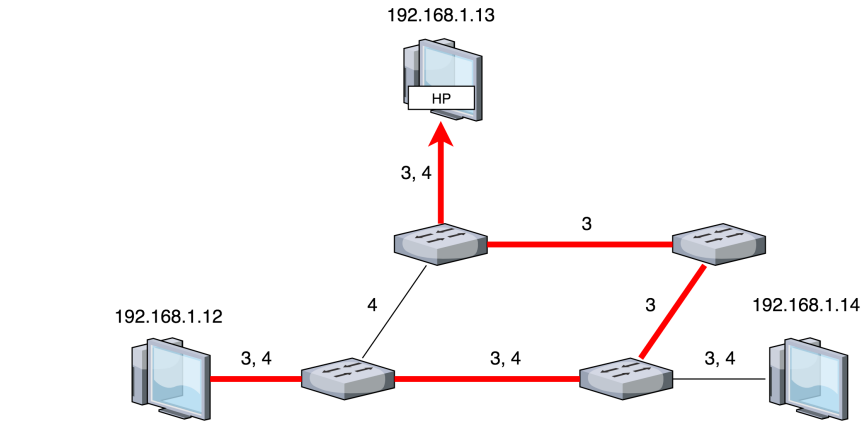


SHARP Header

3	0	1	192.168.1.12	3	192.168.1.13	4	192.168.1.14

Figure 5: A sample network using SHARP, with VLANs 3 and 4. A packet starts at 192.168.1.12 and will travel to 192.168.1.14 following the route in the SHARP header.

2. The source machine parses the newly added SHARP header; it determines that IP Address 192.168.1.13 is the next node in the chain, and that the VLAN to be used for transportation between the current node and the next node is 3. HP is send over UDP to that address, and just before the packet leaves the source machine, a VLAN tag with ID 3 is inserted in the Ethernet frame header. Note that because of the way the VLAN trunk ports have been configured, HP does not take the most direct route to 192.168.1.13. See Figure 6.



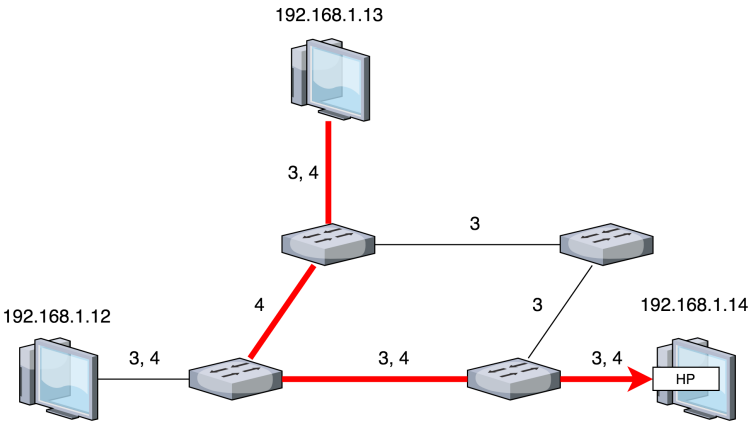
SHARP Header

3	0	1	192.168.1.12	3	192.168.1.13	4	192.168.1.14

Figure 6: HP is sent over UDP to 192.168.1.13, with a VLAN tag ID of 3.

3. 192.168.1.13 receives HP and increments the *Index* field of the SHARP header. Using the new *Index* value, it determines that 192.168.1.14 is the next node and sends HP over UDP to

that address. Again, before the packet leaves, as specified in the SHARP header, a VLAN tag with ID 4 is inserted in the Ethernet header. See Figure 7.



SHARP Header

3	1	1	192.168.1.12	3	192.168.1.13	4	192.168.1.14

Figure 7: *HP* is sent over UDP to 192.168.1.14, with a VLAN tag of 4.

4. 192.168.1.14 receives the packet and increments the SHARP header's *Index* field. The SHARP client finds that the new *Index* is equal to the *Length* - 1, indicating that the packet is at the last SHARP node in the chain and the H needs to be removed to reveal the inner IP packet P. Because the *DestinationFlag* is asserted in H, 192.168.1.14 is P's original destination, so it simply injects P into its own kernel networking receive path. The SHARP client also finds that the *SendOrReceiveFlag* is not asserted in H, meaning that 192.168.1.14 was not the original source of the connection and thus does not already have an H^{-1} to append to outgoing response packets P^{-1} belonging to the connection. The SHARP client reverses the header to create H^{-1} (the result of which is shown at the bottom of Figure 8) and saves it in a data structure that maps connection 5-tuples to SHARP headers. Reversed SHARP headers always have both flags asserted. Since the *DestinationFlag* is asserted in H, the *Length* field of H^{-1} will be the same as in H.

SHARP Header

3	2	1	192.168.1.12	3	192.168.1.13	4	192.168.1.14

Reversed SHARP Header

3	0	3	192.168.1.14	4	192.168.1.13	3	192.168.1.12

Figure 8: A SHARP header with the *DestinationFlag* asserted, and reversed counterpart, to be appended to packets sent by the destination.

5. The process running in 192.168.1.14 for which P was originally destined forms a response packet P^{-1}. Once the packet has an IP header, it is intercepted and the map we discussed earlier is consulted. Since a matching SHARP header H^{-1} for the connection exists, it is appended to P^{-1} to make $H^{-1}P^{-1}$. The packet is then redirected to the next node in the chain, and continues until it is eventually received by 192.168.1.12.

6. When $H^{-1}P^{-1}$ arrives at 192.168.1.12, since the *SendOrReceiveFlag* is asserted, the SHARP client does not add a reversed header to its map. However, because the *DestinationFlag* is asserted, it reinjects P^{-1} into its kernel networking receive path.

In the second example, starting with Figure 9, the *DestinationFlag* is not asserted, meaning that the last IP address in the chain is not a SHARP node, and the last SHARP node must proxy the connection. In this example, the default VLAN ID of 1 must be explicitly shown, and the destination node is connected to its switch with a VLAN 1 access port instead of a trunk port.

1. An outbound packet P destined for 192.168.1.14 is intercepted, and a SHARP header H with a length of 2 is appended to it to create HP. See Figure 9.

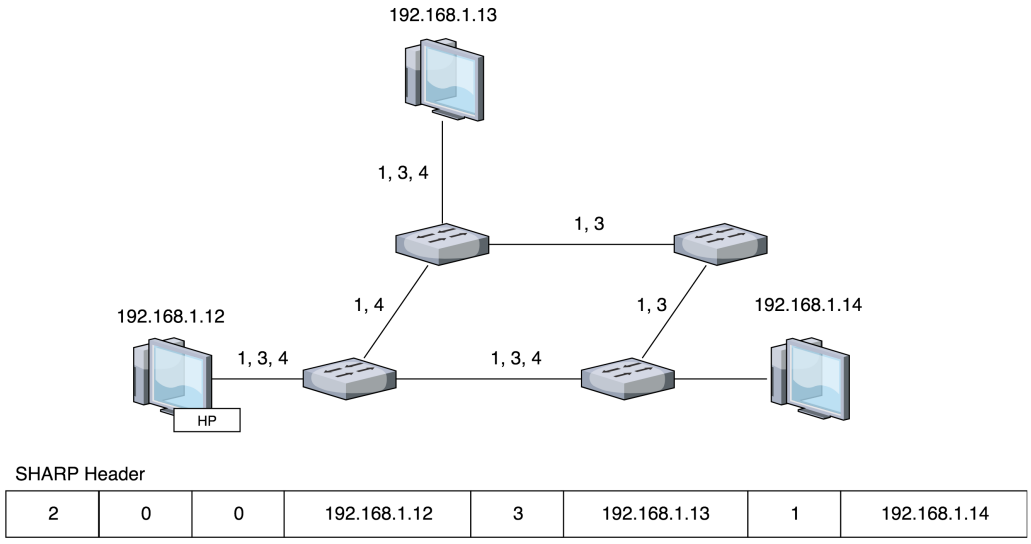


Figure 9: A sample network using SHARP, with VLANs 3 and 4 and the default VLAN 1. A packet starts at 192.168.1.12 and will travel to 192.168.1.14 following the route in the SHARP header.

2. At this point the procedure is similar to that of the first example. The packet is redirected to the next node in the chain, namely 192.168.1.13, and a VLAN tag of 3 is inserted into the Ethernet frame header. Inserting the VLAN tag influences the path the packet HP takes to get to the next node.

3. Once HP arrives at 192.168.1.13, the *Index* field is of H incremented, and it is compared to the *Length* field. Since the *Index* is 1 less than the *Length*, the SHARP client on 192.168.1.13 knows that it must strip H off of the packet to produce P. The *DestinationFlag* is not asserted, and so the SHARP client determines that it must proxy the connection between the true source and destination. The SHARP client parses the last IP address in the header to find the original destination of P, and then overwrites the source IP address in P to be 192.168.1.13. P is then re-injected on 192.168.1.13's outbound kernel networking path. Note that even though a default VLAN ID of 1 is used for the proxied connection, a VLAN tag must still be inserted since the SHARP node is on a trunk port that expects tagged packets. In order to save the proper SHARP header for the return path, 192.168.1.13 reverses H as shown in Figure 10 to produce H^{-1}. Note that the number of IP addresses decreases in H^{-1}, but the *Length* field remains the same value since it represents the number of SHARP nodes in the chain. In a similar fashion to the first example, H^{-1} is stored in a table that maps connection 5-tuples to

SHARP headers. Note that the 5-tuple stored in the map contains 192.168.1.13 as one of the IP addresses, since the original source was overwritten.

SHARP Header

2	1	0	192.168.1.12	3	192.168.1.13	1	192.168.1.14

Reversed SHARP Header

2	0	2	192.168.1.13	3	192.168.1.12

Figure 10

4. The packet P travels normally to 192.168.1.14, which believes that 192.168.1.13 is the source of the connection. The packet is received by a process on 192.168.1.14, which sends a response packet P^{-1} to 192.168.1.13. See Figure 11.

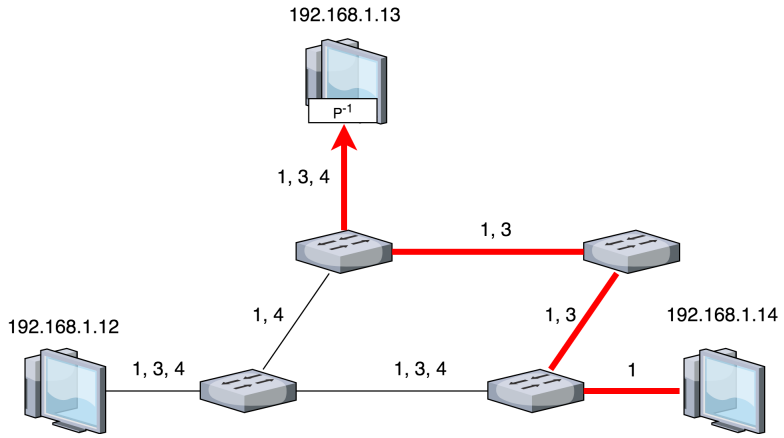


Figure 11: Because 192.168.1.13 modified the source IP address field in P's IP header, 192.168.1.14 is unaware that 192.168.1.12 is the original source of P, and sends its response packet P^{-1} to 192.168.1.13.

5. Once P^{-1} arrives at 192.168.1.13, the SHARP client checks the connection 5-tuple against the connection-to-header map. Since it was populated earlier, the SHARP client finds a header H^{-1} to append to P^{-1}. Parsing H^{-1}, the SHARP client decides to send $H^{-1}P^{-1}$ to 192.168.1.12 over VLAN 3. See Figure 12.

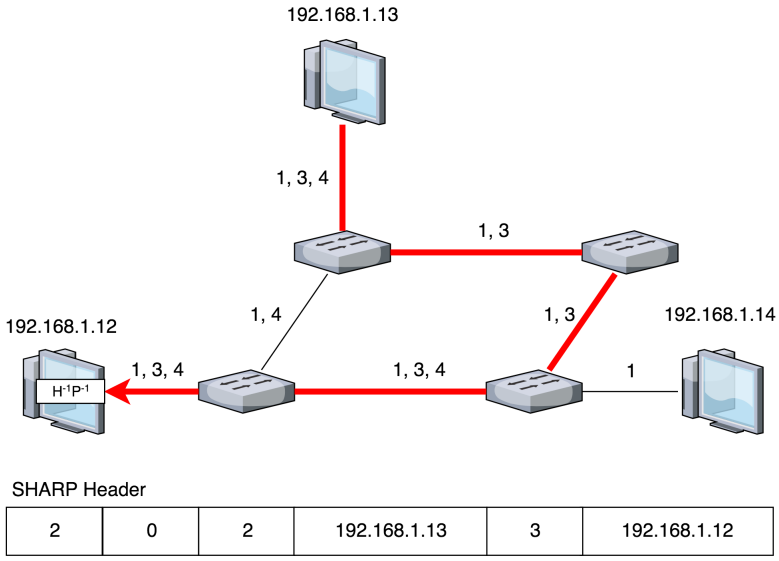


Figure 12: H^{-1} is prepended to P^{-1} and the resulting packet is sent to 192.168.1.12 over VLAN 3.

6. $H^{-1}P^{-1}$ arrives at 192.168.1.12. The SHARP client increments the *Index* field of H^{-1}, and recognizes that it must perform a special action. Since the *SendOrReceiveFlag* is asserted, the SHARP client strips off H^{-1} and injects P^{-1} into its networking inbound path.

3.2.3 SHARP Design Benefits

The Layer 3 routing control offered by SHARP surpasses that of commodity switch-based SDN implementations. Let us examine how a switch-based SDN could approximate SHARP's system of redirecting traffic through a series of intermediate nodes. Flow rules could be written for each OpenFlow-enabled switch to change the outbound port to which they send an incoming packet, essentially causing the switches to force a packet to a different destination machine without changing the destination MAC and IP addresses. However, when the intermediate machine receives the packet, unless IP forwarding is enabled the packet will be dropped because of the mismatch between the packet's destination IP address and the IP address of the machine. Making this strategy work requires enabling IP forwarding on endpoints, and thus the system is not host-independent.

Even if forwarding is enabled, we have not yet taken into account dynamic route changes. Simply forcing a packet to an intermediate node by manipulating outbound port selection is a complicated enough task, but dynamically updating OpenFlow rules to move the packet from hop to hop in the chain is even more so. Although not included in our implementation, our design allows for updates to be made to the SHARP header associated with a connection while the connection is active. In almost any case, a SHARP node can be inserted into or removed from a chain without disrupting the connection itself (an obvious exception to this rule would be removing or changing the source and destination nodes specified in the SHARP header). The only non-trivial exception is that when the *DestinationFlag* is not asserted for the SHARP header of a TCP connection, the proxying node cannot be changed. Since the connection destination believes that the proxying node is the source of the connection, if packets claiming to be part of the connection were to suddenly start being sent from another IP address, the destination would issue a TCP RST packet to reset the connection. In Section 6.3, we discuss some developing technologies that aim to solve this problem.

A further benefit SHARP provides over switch-based SDNs is that it does not require the intermediate nodes or destination to exist in the same LAN as the connection source. As long as a

node outside of the LAN has a SHARP client installed and has a publicly routable IP address, it can be specified as a part of the chain. An example use case for this feature would be network administrators offloading security monitoring to a cloud-based IDS for their enterprise's cloud service traffic. Of course, VLAN tagging has to be disabled for nodes sending SHARP packets to SHARP nodes in other parts of the Internet, however it would be unrealistic to expect decisions made at and endpoint to allow for Layer 2 routing control at the Internet level. In this way, we say that SHARP provides the same level of fine-grain routing control offered by switch-based SDNs within the LAN, while surpassing switch-based SDNs by providing coarse-grain routing control over the Internet.

3.3 Registering Kernel Callouts with WFP

In Section 2.2.2, we gave an overview of the Windows Filtering Platform: how it interacts with the kernel networking queue, the various elements of a callout and its `classifyFn` function, as well as what functions are necessary to register a callout with the WFP filter engine. As we will explain later in this section, much of our implementation of SHARP relies heavily on WFP callouts; since the PEACE driver code base already has a standardized mechanism for registering and unregistering callouts, rather than reinventing the wheel, we decided to use the predefined functions offered by PEACE.

The first step to registering a callout is creating a Global Unique Identifier (GUID) for the callout, which is a randomly-generated 128-bit string. The GUID is then defined as follows:

```
#define WINSIGHT_MAC_OUTBOUND { 0x58de7344, 0x5059, 0x41e1, 0x92, 0x6f, 0x12, 0x19, 0x49,
    0x0b, 0xf7, 0x4f}
```

After creating the GUID and our `classifyFn` function, we can create an instance of the `CALLOUT_DATA` struct defined by PEACE. The struct contains a pointer to the WFP filtering layer the callout should be registered at, the callout GUID, as well as the names of the `classifyFn`, `notifyFn`, and `flowDeleteFn` functions associated with the callout. An example `CALLOUT_DATA` struct definition is as follows:

```
static CALLOUT_DATA MAC_ETHERNET_OUTBOUND = { &FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET,
    {WINSIGHT_MAC_OUTBOUND, 0, MQPKERNEL_OutboundMacClassify, Notify, NULL}, 0,
    L"WinSight Egress", L"Winsight for MAC Egress" };
```

Here, the `classifyFn` function name is *MQPKERNEL_OutboundMacClassify*, the `notifyFn` function is a generic function defined by PEACE called *Notify*, and the `flowDeleteFn` has been set to `NULL`. With a fully defined struct, we now add the following code block to the `SetupCallbacks()` function:

```
if (NT_SUCCESS(status)) {
    status = RegisterCallout(&MAC_ETHERNET_OUTBOUND);
}
```

The `SetupCallbacks()` function is responsible for calling `FwpmEngineOpen()`, `FwpmTransactionBegin()`, and `FwpmSubLayerAdd()` to initiate the process of registering all the defined callouts. The pointer to each `CALLOUT_DATA` struct is then passed in turn to the `RegisterCallout()` function, as demonstrated in the above code snippet. `RegisterCallout()` formats the data contained in the passed struct into a `FWPM_CALLOUT` struct and passses the result into a call to `FwpmCalloutAdd()`, thus completing the callout registration. As part of the clean-up process during an uninstall, the PEACE driver calls its own `CleanupCallbacks()` function, which simply calls `FwpsCalloutUnregisterById()` for each callout that was installed in `SetupCallbacks()`. When creating a new callout, a line is added to `CleanupCallbacks()` as

follows:

```
(void)FwpsCalloutUnregisterById(MAC_ETHERNET_OUTBOUND.CalloutId);
```

3.4 Common Kernel Module Functions

The kernel module implements several distinct operations at different callout layers of the Windows Filtering Platform (WFP). However, large sections of code and functionality are shared between each WFP callout. As a design principle, the WFP offers functionality that can be used at any layer with very little modification. For example, there are unified strategies throughout the WFP to retrieve, modify, and reinject a packet into the network queue. Consequently, our helper functions which supplement WFP's library implicitly change their behavior based on the WFP layer of the caller. As a result, parts of the code common between the callouts are implemented in the same way. These sections can be considered building blocks which form the foundation of any given callout.

3.4.1 Allocating and Freeing Kernel Memory

Kernel space cannot rely on the same memory management abstractions afforded to user space programs. This is because the kernel itself is responsible for implementing memory virtualization guarantees, typically by dividing physical memory into pages and swapping data in page-sized chunks when required by the processor. The kernel has no access to those systems, and instead uses a reserved chunk of privileged memory designated for itself. Without memory management, operations which are commonplace in userspace C, like `malloc` and `free`, become non-trivial in the kernel. Luckily, Windows offers APIs which manage memory in the kernel, abstracting away a lot of the extra complexity.

The Windows API provides calls which allocate and free privileged memory: `ExAllocatePoolWithTag` and `ExFreePoolWithTag`, respectively. ExAllocatePoolWithTag is always called with the `NonPagedPool` parameter, which notifies the call to grab memory from the non-paged (i.e. privileged) memory pool. Allocated buffers are associated with user-generated four-byte tags which serve as identifiers for allocated sections of memory. When freeing a buffer, a call to ExFreePoolWithTag must provide the tag used when allocating that buffer. This memory allocation technique is present throughout the implementation of the kernel modules, and will be referenced throughout the remainder of the implementation.

3.4.2 Retrieving a Packet's Data

At a fundamental level, WFP callouts must receive, inspect, modify, and re-inject packets. Consequently, packets must be transformed into a state where they can be easily analyzed and modified. Fortunately, this process is ubiquitous along the platform's different filtering layers. Thus, the process to transform packets into an modifiable state is reused throughout all of the kernel module's callouts. Specifically, packets must be transformed from their representation in the WFP to a buffer of raw bytes.

As mentioned in the Background, when packets reside within the Windows kernel, they are represented as a *NetBuffers*, stored within a *NetBufferList*. The NetBufferList structure definition is shown below:

```
typedef struct _NET_BUFFER_LIST {
  union {
    struct {
      PNET_BUFFER_LIST Next;
      PNET_BUFFER    FirstNetBuffer;
    };
```

```
    SLIST_HEADER         Link;
    NET_BUFFER_LIST_HEADER NetBufferListHeader;
  };
  PNET_BUFFER_LIST_CONTEXT Context;
  PNET_BUFFER_LIST       ParentNetBufferList;
  NDIS_HANDLE            NdisPoolHandle;
  PVOID                  NdisReserved[2];
  PVOID                  ProtocolReserved[4];
  PVOID                  MiniportReserved[2];
  PVOID                  Scratch;
  NDIS_HANDLE            SourceHandle;
  ULONG                  NblFlags;
  LONG                   ChildRefCount;
  ULONG                  Flags;
  union {
    NDIS_STATUS Status;
    ULONG      NdisReserved2;
  };
  PVOID                  NetBufferListInfo[MaxNetBufferListInfo];
} NET_BUFFER_LIST, *PNET_BUFFER_LIST;
```

Many of these structure parameters pertain to metadata which is not relevant for the task of copying a packet to a data buffer. What is most important is that each NetBufferList can point to another NetBufferList (the *Next* parameter), and that each NetBufferList is in turn composed by a list of *NetBuffers*, the first of which is pointed to by the *FirstNetBuffer* parameter. NetBufferLists may be joined together to represent several packets with similar out of band properties. Thus, to fully read the data of a packet, one must iterate through the list of NetBufferLists, and read through the list of NetBuffers within each NetBufferList. Another parameter which stands out is the *ParentNetBufferList*, which points to the original NetBufferList if a given NetBufferList is a clone. However, we choose to not use the WFP's built in tools for cloning a packet, so in this context the parameter can be ignored.

NetBuffers themselves are made up of a linked list of physical memory chunks called *Memory Descriptor Lists*, or MDLs. When put together, the MDL memory chunks in the list describe a single contiguous chunk of virtual memory. Thus, the MDL keeps track of how a virtual buffer is spread over physical pages of memory. The NetBuffer structure definition is shown below:

```
typedef struct _NET_BUFFER {
  union {
    struct {
      PNET_BUFFER Next;
      PMDL       CurrentMdl;
      ULONG      CurrentMdlOffset;
      union {
        ULONG DataLength;
        SIZE_T stDataLength;
      };
      PMDL       MdlChain;
      ULONG      DataOffset;
    };
    SLIST_HEADER    Link;
    NET_BUFFER_HEADER NetBufferHeader;
  };
  USHORT              ChecksumBias;
  USHORT              Reserved;
```

```
  NDIS_HANDLE         NdisPoolHandle;
  PVOID               NdisReserved[2];
  PVOID               ProtocolReserved[6];
  PVOID               MiniportReserved[4];
  NDIS_PHYSICAL_ADDRESS DataPhysicalAddress;
  union {
    PNET_BUFFER_SHARED_MEMORY SharedMemoryInfo;
    PSCATTER_GATHER_LIST   ScatterGatherList;
  };
} NET_BUFFER, *PNET_BUFFER;
```

Each MDL represents a contiguous virtual buffer of memory. In reality, an MDL may consist of many chunks of physical memory linked together to make the contiguous virtual chunk. This distinction becomes important when designing the tools to read out the data from a packet into a buffer. Fortunately, the WFP API abstracts away any direct interaction with the MDLs to retrieve a packet. From the perspective of the API caller, a NetBuffer is composed of arbitrary chunks of memory which represent the packet's data.

To retrieve the packet's data and copy it to a contiguous buffer, a callout iterates over the NetBuffers which compose the NetBufferList associated with the packet and extract the data stored in the MDLs. The following code snippet illustrates how this is done:

```
while (netBuffer != NULL) {
    bp = NdisGetDataBuffer(netBuffer, NET_BUFFER_DATA_LENGTH(netBuffer), buffer, 1, 0);

    if (bp == NULL) {
        WdfSpinLockAcquire(AtomicMemoryManagementLock);
        ExFreePoolWithTag(bufferHead, BUF_TAG);
        WdfSpinLockRelease(AtomicMemoryManagementLock);
        return;
    }

    if (bp != buffer) {
        RtlCopyMemory(buffer, bp, NET_BUFFER_DATA_LENGTH(netBuffer));
    }

    buffer = buffer + NET_BUFFER_DATA_LENGTH(netBuffer);
    netBuffer = NET_BUFFER_NEXT_NB(netBuffer);
}
```

In this implementation, a unified data buffer is extracted from the MDLs associated with each NetBuffer using the `NdisGetDataBuffer` function. This API call takes in a pointer parameter which suggests a destination address for the copy. Whether the function actually does copy to that address is dictated by whether the MDLs occupy a contiguous chunk of memory. If the MDLs are indeed contiguous, then the function will return a different pointer to the start of the data; if they are instead fragmented, the function will copy the data contiguously into the provided pointer argument, and the return pointer will match the argument. Thus, to ensure that all the data is copied into the same place, we must copy data from the buffer pointed to by the return argument to the buffer pointed to by the aforementioned parameter if their values are misaligned.

The loop begins with the `buffer` variable pointing to the head of the destination buffer. At subsequent iterations of the loop, the pointer to the copy destination is incremented by the length of the prior NetBuffer; that way, the data of sequential NetBuffers is concatenated into the single array. This buffer is allocated from privileged memory, using the techniques described above. The length of the data is determined by summing the length of all NetBuffers in the NetBufferList.

3.4.3 Rebuilding and Re-injecting Packets

When packet data has been copied into a buffer of bytes, the callout can arbitrarily manipulate it to fit its specific purpose. Once the buffer has been modified, it must be repackaged into a new Windows packet and inserted back into the kernel's packet queue. Fortunately, the WFP offers a unified method to do this, which is shared among the callouts. Only slight changes to function names are necessary to apply the technique to a specific filtering layer.

In the previous section, packets were broken down from NetBufferLists to NetBuffers to MDLs to raw data. Conversely, to build a packet from a data buffer, that buffer must be converted into a list of MDLs, which must then be packaged into a list of NetBuffers, and which finally must by linked together into a NetBufferList. Fortunately, WFP offers functionality which performs each step of this packaging process. First, a call is made to `IoAllocateMdl` and `MmBuildMdlForNonPagedPool` to bundle the data buffer into a list of MDLs. The former function allocates an MDL large enough to fit the data buffer. However, the MDL is a virtual abstraction; therefore, the latter function actually maps the MDL to physical chunks of memory. Once the MDL is allocated and mapped, a call to `FwpsAllocateNetBufferAndNetBufferList` packages the MDL into NetBuffers, and those NetBuffers into a NetBufferList.

The packaged NetBufferList can then be re-injected into the kernel queue, at the layer of the given callout. To achieve this, the WFP provides injection API calls for each layer of the SHARP header. Each call takes the current injection handle and context, several pieces of information pertinent to the filtering layer, the NetBufferList, and a completition callback function. The callback function will fire after the packet is successfully re-injected in the kernel queue; it is used to free all intermediate data buffers allocated in kernel memory to produce the packet. The injection function for the outbound Ethernet layer callout is shown below as an example:

```
NTSTATUS FwpsInjectMacSendAsync0(
  HANDLE             injectionHandle,
  HANDLE             injectionContext,
  UINT32             flags,
  UINT16             layerId,
  IF_INDEX           interfaceIndex,
  NDIS_PORT_NUMBER   NdisPortNumber,
  NET_BUFFER_LIST    *netBufferLists,
  FWPS_INJECT_COMPLETE completionFn,
  HANDLE             completionContext
);
```

If the packet fails re-injection, the injection function will return a status other than `STATUS_SUCCESS`, and the completion function will not fire. In this case, the callout must manually free any buffers used to produce the packet.

3.4.4 Parsing a SHARP Packet

As described in the Design section, SHARP packets wrap a packet payload with a SHARP header. As a result, a SHARP packet may have both an external IP, UDP and SHARP header which are actually used to traverse the network, and an internal IP and TCP/UDP header for the payload. The WFP's built in packet parsing tools fail to retrieve all information from the two-tiered nature of the packet. If the packet does not yet have a SHARP header, the WFP will only retrieve information about the payload; conversely, if the packet has a SHARP header, the WFP will only retrieve information about the external IP and UDP headers. Furthermore, WFP has a strict adherence to the OSI model, so a given filtering layer will only expose packet information at that layer. For example, an IP filtering layer will only expose Layer 3 information directly to its related callouts. These restrictions in the WFP make a custom parser necessary to analyze the contents of a packet.

The parser itself writes values to a custom output data structure. This structure supports parsing both SHARP and non-SHARP packets, setting a boolean flag if the given packet belongs to SHARP. The parser also follows the WFP's design principle of making its helpers as ubiquitous as possible across the filtering layers. An enum is passed into the parser to indicate the filtering layer of the caller, and the parser adjusts its offsets based on the layer to retrieve the same values. Generally, the parser looks for IP addresses, source and destination ports, and transport layer protocols (i.e., the data required to build a five-tuple). It also grabs specific SHARP header information, including the header start index, the length of the header, the current hop on the packet's route, and the flags associated with the header. This simplifies areas of the code which must interface with the SHARP header, since parameter retrieval only has to be done once. Shown below is the data structure which the packet parser loads:

```
typedef struct MqpPacketData {
   UINT32 SHARPHeaderSourceIpAddress;
   UINT32 SHARPHeaderDestinationIpAddress;
   UINT32 SHARPHeaderLength;
   UINT32 payloadSourceIpAddress;
   UINT32 payloadDestinationIpAddress;
   UINT32 SHARPHeaderStartIndex; //how far into the packet does the SHARP header start?
   UINT32 payloadStartIndex; //how far into the packet does the payload start
   UINT16 SHARPHeaderSourcePort; //should always be 7001 for outbound packets
   UINT16 SHARPHeaderDestinationPort; //should always be 7000 for outbound packets
   UINT16 SHARPHeaderIndex;
   UINT16 SHARPHeaderHops;
   UINT16 SHARPHeaderVlanTag;
   UINT16 SHARPHeaderFlags;
   UINT16 payloadSourcePort;
   UINT16 payloadDestinationPort;
   UINT8 SHARPHeaderTransportType; //should always be UDP
   UINT8 payloadTransportType; //the transport layer type of the payload
   UINT8 SHARPDestinationType; //the destination type flag of the SHARP header
   UINT8 isASHARPPacket; //is this struct representing a SHARP packet, or not?
   UINT8 isVlanTagged; //is this packet vlan tagged?
} MqpPacketData;
```

Packet parsing is part of the sequence of actions which are common to all of the callouts. Generally, the packet is parsed immediately after its data is extracted into a buffer.

3.5 The ALE Lookup Table

While most of the code implementation fits within the WFP callout paradigm presented so far, one other component exists within the kernel outside of the callouts themselves. This component, called the *ALE Lookup Table*, manages which packets have to be sent to the SHARP daemon to receive SHARP packets. This lookup table interacts with callouts at several filtering layers, including the outbound ALE layer, which populates the table, and the outbound network layer, which references the table. Since the table must interact with more than one callout, it should exist as a standalone entity in the kernel. Furthermore, the table must persistently maintain the connections which need SHARP headers because ALE layer callouts trigger for only the first packet in a stream. This means that the ALE layer will not reliably forward any subsequent packets in a connection, and cannot indicate whether a connection needs SHARP routing on a packet by packet basis. The ALE lookup table allows a connection to be tabulated once and referenced for all subsequent packets in that connection. The diagram below summarizes how the WFP callouts use the ALE lookup table:

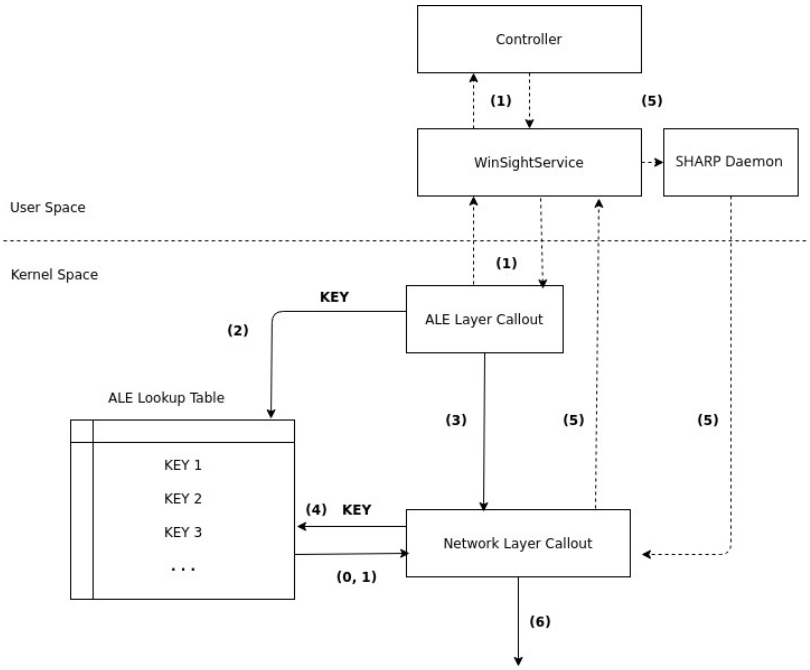


Figure 13: ALE Lookup Table control flow.

Each step of the control flow is labeled in the diagram, and described in detail below:

1. The first packet in a flow is intercepted at the ALE layer callout. The ALE callout sends the packet to user space, and subsequently to a remote SDN-like controller. The controller passes back a decision on whether the flow needs SHARP routing via WinSightService.

2. If the controller confirms that the packet needs SHARP routing, then the ALE layer callout will populate the ALE lookup table. The table entry uses the packet's connection five-tuple (*Src_IP, Src_Port, Protocol, Dst_IP, Dst_Port*) as a key, and asserts a boolean flag as the value.

3. The packet passes through the ALE layer callout and is intercepted at the network layer callout.

4. The network layer uses the packet's five-tuple to reference the ALE lookup table. If an entry with the provided key exists with the asserted boolean flag, the network layer callout sends the packet back to user space to append a SHARP header. Since all subsequent packets in the flow will have the same five-tuple, they will all be sent to user space and given SHARP headers, regardless of whether the ALE layer callout intercepted them.

5. The packets which are sent up to user space interact with WinSightService and the SHARP daemon, which appends a SHARP header to the packet and returns it back to the network layer callout. The SHARP daemon may use both local and remote guidelines to select the routing path expressed in the SHARP header.

6. When the network layer callout sees a packet with a SHARP header, it immediately lets it through. The packet may then pass through to lower filtering layers, and eventually the network.

3.5.1 ALE Lookup Table Design

The ALE Lookup Table itself is a singly linked list which exists in protected kernel memory. It implements a set of functions necessary to initialize, parse, and update any linked list. These calls are listed below, with a brief description of each:

- **initAleLookupTable**: Initializes the ALE lookup table, and populates it with a single empty node to initialize the linked list. This function gets called once, during callout registration. It also is responsible for initiating concurrency data.
- **lookupEntryFromTable**: Passes through the table, looking for any entries matching a given five-tuple key. Passing through the table in this case is the same as traversing a singly-linked list.
- **addEntryToLookupTable**: Adds a new entry to the head of the ALE Lookup Table. The previous head becomes the successor of the new element. Entries are created with five-tuple keys and boolean values.
- **deleteEntryFromLookupTable**: Removes an entry from the table. To remove nodes, a reference is made to the previous/current node as the list is traversed. When the target node is found, it is freed, and the node's predecessor gets linked to the nodes successor.

Other more trivial helpers and macros exist which handle the creation of a FiveTuple key from its component values. Once the table is instantiated, the ALE layer callouts curate the table with `addEntryToLookupTable`, and the network layer callouts retrieve entries with `lookupEntryFromTable`. This functionality is sufficient to acheive the design goals of the lookup table.

The Windows kernel uses multithreading throughout its modules to achieve acceptable performance; concurrency extends to the kernel queue's design, and thus the Windows filtering platform. Thus, the ALE lookup table must handle cases where two threads are trying to access its data. The size and relative access times of the lookup table are small enough where we deemed wrapping the table with a spin lock would sufficiently protect the system from race conditions without significantly affecting performance. Like most other lock mechanisms in the kernel, a single spin lock object is declared in global memory for the table. When a callout wants to access or modify the table, it must claim ownership of the lock. It is incumbent on that callout to release the lock when it has finished using the table.

3.6 VLAN Tag Manipulation

One of the primary responsibilities of the kernel module is to append a VLAN tag to the Ethernet header of all outbound SHARP packets. Consequently, it must also intercept inbound SHARP packets and remove their tags.

3.6.1 VLAN Tag Discovery

Each node running SHARP must append a VLAN tag to outbound SHARP packets to dictate the packet's Layer 2 path to the next SHARP node. To discover what VLAN a packet needs to belong to, the SHARP kernel module performs a lookup on the SHARP header of the SHARP packet. The header is indexed using its *Current Index* parameter to retrieve the tag value. For an index value of n, the SHARP header is indexed by $6 + 4 + 6n$ bytes. Whenever a SHARP packet exits a SHARP node, the kernel module increments its current index value. That way, the subsequent SHARP nodes in the chain can index further into the header to retrieve tags.

3.6.2 The Ethernet Filtering Layer

The kernel module must operate withing the Windows Filtering Platform (WFP) to interact in any way with the kernel's packet queue. Specifically, the module must register callouts at different filtering layers of the WFP which respect the layers of the OSI model. Windows' native network drivers construct network packets by sequentially adding headers one layer at a time. For example, the platform might construct a packet destined as HTTP traffic by first appending a TCP header, then an IP header, and finally an Ethernet header. The WFP allows one to intercept this construction process at different layers of the network model. Callouts are similar to callback functions, such that the callout is called for each packet in the kernel queue when it reaches the layer associated with the callout.

SHARP's kernel module registers a WFP callout at the Ethernet layer with the to append VLAN tags to SHARP packets. While Microsoft's main documentation for the WFP does not indicate that the Ethernet layer can register callouts, one isolated documentation page shows that the WFP offers four Ethernet-related filtering layers [23]. They are listed below, with a brief description of each:

- **FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET**: Intercepts packets at the Ethernet layer on the outbound path. Packets are intercepted *behind* the network card, i.e. before the network interface card has processed the packet.
- **FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE**: Intercepts packets at the Ethernet layer on the outbound path. Packets are intercepted *in front of* the network card, i.e. after the network interface card has processed the packet.
- **FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET**: Intercepts packets at the Ethernet layer on the inbound path. Packets are intercepted *behind* the network card, i.e. after the network interface card has processed the packet.
- **FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE**: Intercepts packets at the Ethernet layer on the outbound path. Packets are intercepted *in front of* the network card, i.e. before the network interface card has processed the packet.

The notion of the network interface card is important when deciding where to place the VLAN callouts. On Windows, Layers 3 and 4 are managed by the OS (in software), while Layer 2 is managed by the network card, on separate piece of hardware. Since the firmware on the network interface cards is fixed, there is a possibility that packet modifications could cause errors at the network card. In those cases, packet modifications have to be introduced after the network interface card has processed them. In our case, however, all modification can be done before the network card processes the packet, so long as the card follows the IEEE 802.1 VLAN specification. To append a VLAN tag to outbound SHARP packets, the kernel module generates a callout at the OUTBOUND_MAC_FRAME_ETHERNET layer of the WFP. This layer allows the module to append VLAN tags to outbound packets before the packets enter the network interface card. The module registers and initializes the callout with the WFP, using the methods described in Section 3.3.

3.6.3 Outbound Tag Insertion

The outbound ethernet callout itself follows a similar workflow as the rest of the callouts implemented in the kernel module. After filtering the packet, the callout retrieves the data stored within the packet as a buffer. If the intercepted packet belongs to SHARP, the callout retrieves the VLAN tag which it must append from the SHARP header. Finally, the callout appends the tag to the data buffer, generates a new Windows packet using the modified data buffer, and *injects* that packet at the Ethernet layer of the kernel queue. Since the callout injects the new packets, the packets now originate from the callout.

Before the callout inspects the packet contents, it performs preliminary filtering on the packet. Specifically, it must always allow re-injected packets to proceed, without any further modification.

Since the callout injects its modified packets at the Ethernet layer of the kernel queue, the callout will necessarily intercept those packets a second time. The callout must allow those packets to proceed, since they are already tagged. The following code snippet checks if a packet had been re-injected:

```
FWPS_PACKET_INJECTION_STATE packetState;
packetState = FwpsQueryPacketInjectionState(InjectionHandleMac, LayerData, NULL);
if (packetState == FWPS_PACKET_INJECTED_BY_SELF) {
   ClassifyOut->actionType = FWP_ACTION_PERMIT;
   if (Filter->flags & FWPS_FILTER_FLAG_CLEAR_ACTION_RIGHT) {
      ClassifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
   }
   return;
}
```

The code snippet checks if the incoming packet originated from the callout itself by checking if the packet's metadata includes the FWPS_PACKET_INJECTED_BY_SELF flag. If it does, the intercepted packet is released back into the kernel queue by setting its FWP_ACTION_PERMIT flag.

The outbound Ethernet callout must then consume the incoming packet and produce a buffer with its contents. This process is shared among SHARP's implemented callouts, so implementation details can be found in Section 3.4, which describe shared code practices. Several peculiarities do arise because packets are being intercepted at the Ethernet layer. Most curiously, when packets are intercepted at the OUTBOUND_MAC_FRAME_ETHERNET layer, they already have 14-byte Ethernet headers appended to them. This is in direct contradiction to Windows' own documentation, which claims that packets intercepted at the OUTBOUND_MAC_FRAME_ETHERNET should only be constructed to the network header [24]. According to the documentation, only the OUTBOUND_MAC_FRAME_NATIVE layer should intercept packets at the Ethernet header. Since documentation is already sparse at best for WFP's Ethernet filtering layers, the documentation in this case may simply be incorrect.

After the packet's data has been retrieved by the buffer, it is parsed for all relevant information. Since the WFP forces applications which use it to follow the OSI model, the WFP's native API only provides information about the packet at the layer at which it was intercepted. For example, the WFP only makes Layer 2 information available to our Ethernet layer callouts through its official API; similarly, only Layer 3 information is available to our network layer callouts. As a result, we implemented a set of custom API calls to perform full inspection of the packet. The behavior and implementation of the custom packet parsing API can be found in Section 3.4.4.

The callout must then modify the data buffer to append a VLAN tag, or release the original packet if it does not have a SHARP header. The packet parsing API has a built-in attribute in its return structure which indicates whether the incoming packet is a SHARP packet. If the attribute is not set, the callout simply sets the FWP_ACTION_PERMIT flag on the incoming packet and exits. Otherwise, it parses the SHARP header to look up the tag value to append. The following function parses the SHARP header to retrieve the tag value:

```
USHORT MQPKERNEL_LookupVlanTag(In PCHAR buffer, In UINT packetLength,
   In MqpPacketData* data)
{
   UNREFERENCED_PARAMETER(packetLength);
   UINT offset = data->SHARPHeaderStartIndex + 6 + 4 + (6 * data->SHARPHeaderIndex);
   return *(PUSHORT)(&buffer[offset]);
}
```

This function grabs the SHARP header start index from the struct filled by the packet parsing API, and follows the rules dictated in Section 3.4.4 to grab the packet. Since the tag is 12 bits,

pointer recasting is used to grab two subsequent bytes from the buffer. This lightweight helper showcases the power of this unified packet parsing API. A single call to the API loaded all relevant values into a data structure, so any other callout helpers no longer bear the burden of inspecting and parsing the packet.

The tag itself is inserted 14 bytes into the packet, between the two MAC addresses of the Ethernet header and the header's two-byte type parameter. The composition of the VLAN tag itself is four bytes. The first byte are always set to 0x8100, to signal the presence of the VLAN tag. The next four bits indicate the packet's priority and congestion dropping rules; they are always set to 0x0. Finally, the last 12 bits of the tag hold the tag's value itself. The following code snippet demonstrates how the callout appends the tag to the packet data buffer.

```
UINT32 tagOffset = 12;

// Shift everything past the VLAN tag down 4 bytes
RtlMoveMemory(buffer + tagOffset + tagLength, // Dest
          buffer + tagOffset,         // Src
          packetLength - tagOffset); // Length

// Insert VLAN tag components:
buffer[12] = 0x81; // TPID - 0x8100
buffer[13] = 0x00;

vlanId = INVERT_PORT_BYTES(vlanId);
RtlCopyMemory(&buffer[14], (PCHAR)&vlanId, 2); //final byte and nibble is the VLAN id
```

The `RtlMoveMemory` call creates space in the packet to place the VLAN tag, while the subsequent writes to the data buffer load in the VLAN tag, as specified in the paragraph above. Once the tag has been appended, the callout follows the standard process to repackage the data buffer into a Windows packet, as described in Section X.Y. If the repackaging successfully completes, the packet is re-injected back into the kernel queue's outbound Ethernet filtering layer. WFP provides an API call designed to inject packets at this layer:

```
NTSTATUS FwpsInjectMacSendAsync0(
  HANDLE             injectionHandle,
  HANDLE             injectionContext,
  UINT32             flags,
  UINT16             layerId,
  IF_INDEX           interfaceIndex,
  NDIS_PORT_NUMBER   NdisPortNumber,
  NET_BUFFER_LIST    *netBufferLists,
  FWPS_INJECT_COMPLETE completionFn,
  HANDLE             completionContext
);
```

`FwpsInjectMacSendAsync0` conforms to the behavior of the rest of WFP's re-injection API calls by taking a constructed packet and an injection context, and firing a user-generated callback to cleanup memory if the injection succeeds. The WFP handles memory management if the packet is re-injected successfully; if it fails, the callout frees memory itself. It should be noted that re-injected packets will be intercepted for a second time by the callout; however, the callout permits these packets to continue unmodified, since the `FWPS_INJECTED_BY_SELF` flag has been set in their metadata.

3.6.4 Network Interface Card (NIC) Behavior

Any Ethernet-layer callout must accommodate the fact that the Ethernet layer is handled not only by the host OS but also by a separate piece of hardware called the network interface card (NIC). The network interface card is responsible for translating the software representation of the packet into a transmission over the physical network medium. Since one of the card's responsibilities is the management of MAC addresses, it must implement the IEEE 802 specifications and involves itself in the link layer. SHARP's kernel modules are guaranteed to work on any host whose NIC has implemented the IEEE 802.1 specification; specifically, the card must be able to pass tagged packets up to the OS. However, there is no clear standard which dictates how the card must present tagged packets to the host. Therefore, it is the task of the Ethernet callouts to conditionally implement different types of behavior and maintain the ubiquity of the kernel modules across all Windows systems.

The NIC of the machines that we tested SHARP on will strip VLAN tags from any incoming tagged packets by default. However, they can optionally be configured to leave the VLAN tags unaltered when passing packets up to the OS. For Intel's IEEE 802.1 network cards, this behavior is called *Monitor Mode*, and can be set as a registry option from Windows itself. The kernel modules could conditionally receive packets on the input path with or without VLAN tags, based on how the card itself is configured. It follows that an inbound callout must exist to remove VLAN tags from packets if the NIC chooses not to.

As we mentioned in Section 3.2, VLAN tags have historically been reserved for communication between switches, and manually inserting them into the outbound Ethernet frames of endpoint machines is very atypical behavior. The entire process of developing SHARP was done on VMs, and so during our initial testing phase we were unsure whether a physical machine's NIC would interact nicely with unexpected VLAN tags. We found that, while the system did not crash upon reading manually inserted VLAN tags, it would write the packet onto the wire without calculating the checksums for the IP and TCP or UDP packet headers. We theorize that this happens because inserting a VLAN tag extends the length of an Ethernet frame header by 4 bytes; when the NIC processes the packet, the network and transport layer headers are not where the NIC expects them to be, and it simply decides to skip checksum calculation. When a host receives a packet that has incorrect or blank checksum fields, standard behavior is to discard the packet. Therefore we had write our own routines for calculating IP and transport layer checksums for outgoing packets. Both IP and the transport layer checksums are formed by performing one's complement addition of 16-bit words of data. However, where the IP checksum only sums the words in the IP header, UDP and TCP checksums involve their respective headers, an IP pseudo-header, and the application-level data they are transporting. Figure 14 shows how the pseudo-header is formed out of pieces of the IP header. Our algorithm for transport layer checksumming borrows heavily from the implementation used in the code base of WinDivert [25].

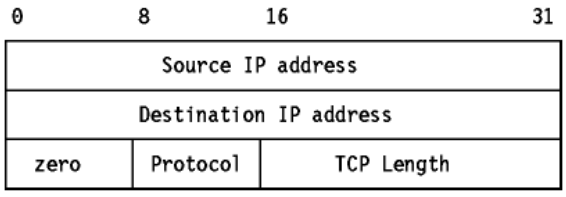


Figure 14: The pseudo-header used in both TCP and UDP layer checksumming is the same. Source: `http://www.danzig.jct.ac.il/tcp-ip-lab/ibm-tutorial/3376c212.html`

3.6.5 Inbound Tag Removal

Since the NIC conditionally removes VLAN tags from incoming packets, an inbound Ethernet-layer callout must exist to remove tags when the network card chooses not to. Fortunately this behavior is much more straightforward that the outbound Ethernet-layer callout's, since behavior is universal across all incoming packets. If the callout sees a packet with a TPID, which indicates the protocol encapuslated in the payload of the frame, corresponding to a tagged packet in the Ethernet frame, the callout removes the tag and resets the packet data to indicate a regular, untagged packet. The packet retrieval process is identical to that of the outbound Ethernet callout; at the end of the procedure, the packet exists in a data buffer. The callout then indexes the packet at a fixed offset, and checks the TPID. If it finds that the packet is tagged, it removes the two-byte tag and the TPID. Finally, the modified data buffer goes through the repackaging process to become a NetBufferList, and that packet is reinjected into the inbound queue. The code snipped below demonstrates the process by which a VLAN tag is removed:

```
UINT32 tagOffset = 12;
RtlMoveMemory(buffer + tagOffset, //Dest
              buffer + tagOffset + tagLength, //Src
              packetLength - 16);
RtlZeroMemory(buffer + (packetLength - tagLength), tagLength);
bufferLength -= tagLength;
```

The four byte chunk which holds the TPID and VLAN tag is removed via a call to RtlMoveMemory, which consequently shortens the packet by four bytes. To ensure that the final four bytes at the end of the buffer are no longer used, they are set to zero. The buffer length is also adjusted to reflect the shorter size of the packet. This does not affect a subsequent call to *ExFreePoolWithTag*, which needs only a pointer to the head of the buffer to free the original buffer which we allocated.

3.7 SHARP Header Management

As demonstrated in the previous section, the process which appends SHARP headers to packets is multifaceted and relatively complex when compared with other operations, such as VLAN tag manipulation. This is because the SHARP header process spans several sections of the code as a whole, using subsystems located in both user space and kernel space. Generation begins in the outbound network filtering layer, where a callout uses the ALE lookup table to recognize when a packet needs a SHARP header. If a packet does indeed need a header, the outbound network chooses not to reinject it, and instead sends it up to user space via WinSightService. WinSightService, which acts as the user space-side portal between user space and kernel space, forwards the packet data to the SHARP daemon. The SHARP daemon receives the packet, assembles a path, and generates a SHARP header. It then appends the SHARP header to the packet, which has now become the new SHARP packet's payload, and sends it back out onto the network. When the network layer callout intercepts this new SHARP packet, it immediately lets it through to the network.

3.7.1 Retrieval of Target Packets

To retrieve packets which need SHARP headers, a callout must exist somewhere along the kernel's network queue to intercept the packets and pass them to user space. Our implementation uses a callout at the outbound network filtering layer to achieve this purpose. Specifically, it sits at the `FWPM_LAYER_OUTBOUND_IPPACKET_V4` layer, which intercepts all IPv4 packets in the outbound queue at the network layer, i.e. after the packet's IP header has been constructed, but before the Ethernet frame's construction or the network interface card. When packets are intercepted at this filtering layer, they are indexed to the start of the IP frame, so the packet does not need to be rewinded to reveal necessary information.

The outbound network callout's logic flow is similar to the other callouts in the kernel module. First, the kernel module extracts the packet's data into a continuous buffer, using the same technique used by the other callouts. This process is described in greater detail in Section 3.4. The buffer is allocated using Windows' memory allocation tools for privileged (non-paged) memory. Once the packet is retrieved, the unified parsing API scans the packet and extracts relevant information from it.

With the packet extracted and parsed, the callout makes a decision on whether it needs to be sent to user space for SHARP routing. To make this decision, the outbound network callout checks if the packet exists as an entry in the ALE lookup table. The ALE lookup table is indexed using a packet's five-tuple, a set of five values which sufficiently categorize a TCP/IP connection. These five values are:

- **Source IP Address**: The source/origin IP address of the packet.
- **Destination IP Address**: The destination/target IP address of the packet.
- **Source Port**: The source port of the packet.
- **Destination Port**: The destination port of the packet.
- **Transport Layer Protocol**: The protocol used in the transport layer of the packet. This will usually be either UDP or TCP.

Luckily, the unified packet parsing API grabs all these values, so key construction only requires that relevant `MqpPacketData` fields are loaded into a `FiveTuple` structure. The packet parsing API is powerful here because of how it simplifies the logic of subsequent callout tasks. Once the key has been generated, a call to `lookupEntryFromTable` searches the ALE lookup table and retrieves the value associated with the key, if it exists in the table. This logic flow is implemented in the code snippet below:

```
UINT32 localAddr =
    FixedValues->incomingValue[FWPS_FIELD_OUTBOUND_IPPACKET_V4_IP_LOCAL_ADDRESS].value.uint32;
  UINT32 remoteAddr =
      FixedValues->incomingValue[FWPS_FIELD_OUTBOUND_IPPACKET_V4_IP_REMOTE_ADDRESS].value.uint32;
  UINT16 localPort = pd->payloadSourcePort;
  UINT16 remotePort = pd->payloadDestinationPort;
  UINT8 protocol = pd->payloadTransportType;

  FiveTuple key;
  INT8 lookup;

  key = buildFiveTuple((ULONG64) localAddr, (ULONG64) remoteAddr,
     localPort, remotePort, protocol);
  lookup = lookupEntryFromTable(key);
```

The callout makes a decision on what to do with the packet based on its presence in the ALE lookup table. Three scenarios are possible: first, the key exists in the table and has an associated set boolean value; second, the key exists in the table and has an associated unset boolean value; and finally, the key does not exist in the table at all. For the purposes of the outbound network callout, the packet is blocked and sent to user space if its key exists in the table with an associated asserted value. Otherwise, the packet is placed back in the network queue without any modification or reinjection. A specialized helper, `MQPKERNEL_PacketToService`, takes the packet data in a buffer and sends it to WinSightService for further processing. A packet can be either blocked or permitted by setting its `ACTION_TYPE` metadata value. A value of `FWP_ACTION_PERMIT` permits the packet, while a value of `FWP_ACTION_BLOCK` blocks the packet.

3.7.2 SHARP Header Generation

After the callout in the outbound network filtering layer sends a packet to WinSightService in a `WINSIGHT_QUERY_DATA` struct with the `MQP_NeedsSharpHeader` flag asserted, a WinSightService worker thread processes the request in the `CollectProcessData()` function. We modified the function as follows to specially process queries that were sent by the SHARP outbound network callout as follows:

```
if (query->RequestFlags.MQP_NeedsSharpHeader == 1) {
      // Copy the packet data from query struct buffer
      int pathLen = query->ProcessPathLength;
      CHAR *MQP_buffer = (char *)malloc(pathLen);
      memset(MQP_buffer, 0x00, pathLen);
      memcpy(MQP_buffer, (void *)query->ProcessPath, pathLen);

      if (MQP_SHARPSocketConnected) {
         int iResult = send(MQP_DaemonSocket, MQP_buffer, pathLen, 0);
         if (iResult != pathLen) {
            AddToMessageLog("Failed to send full packet to the daemon.");
         }
      }
      ...
      query->RequestFlags.IsEmptyPacket = 1;
      return;
   }
```

In this code snippet, we retrieve the packet data stored in the `ProcessPath` buffer of the `WINSIGHT_QUERY_DATA`, and send it to the SHARP daemon through a TCP socket on port 7002 for processing. The daemon has a thread called the `ServiceWorker`, which is dedicated to communication with WinSightService. After performing standard socket setup, the ServiceWorker enters a `while` loop and blocks on a call to `recv(ServiceSocket)`. With this structure, the ServiceWorker can always be prepared for a new message from WinSightService once it finishes processing the previous one.

Any message WinSightService sends to the SHARP daemon will contain a packet. ServiceWorker first checks to see whether the packet starts with an IP header or a SHARP header. If the packet starts with a SHARP header, we know that it was sent by the inbound network callout. This indicates that the only action that must be taken is to reverse the SHARP header and store it in a table that maps connection 5-tuples to SHARP headers. If the *DestinationFlag* is not asserted in the SHARP header of the received packet, the SHARP daemon can conclude that it is executing on a machine that is performing TCP stitching with a non-SHARP destination. If this is the case, to reverse the header the daemon first removes the last IP address and VLAN ID from the header. After this initial check, reversing the SHARP route in the header is simply a matter of reversing the order of the remaining IP addresses and VLAN IDs, regardless of whether the *DestinationFlag* is asserted. Finally, in both cases, the *DestinationFlag* and *SendOrReceive* flags are asserted in the reversed SHARP header.

To determine if the packet starts with an IP header, the SHARP node checks that the first nibble of the packet is equal to `0x4`. If the test is true, the SHARP daemon can conclude that the outbound network callout sent the packet, and that it must append a SHARP header and send the packet along the route specified therein. In a full implementation of SHARP, the daemon would consult with either locally cached rules or the remote controller to determine the SHARP path for each connection. However, due to the time constraints of the project, our system does not include controller interaction. We first check the SHARP header map for any previously-reversed header that matches the outgoing packet's connection 5-tuple. If none is found, a generic SHARP

header is appended and the resulting packet is added to a global semaphore-locked queue for the *ConsumerWorker* to process.

Once the packet is taken off the queue by the *ConsumerWorker* thread, it increments the *Index* field of the SHARP header, and uses it to extract from the header the IP address of the next hop in the chain of SHARP nodes. The *ConsumerWorker* thread then creates a UDP socket to connect to that address. Typically, client sockets are not bound to a particular port; however, we decided to enforce the rule that all client sockets created by *ConsumerWorker* will be bound to UDP port 7001. Since source port binding is an uncommon technique, we have demonstrated its implementation in the code snippet below:

```
// binding on port 7001
struct sockaddr_in sa;
char* localIp;

// connect to Google's public DNS server at 8.8.8.8 to determine local IP address
getLocalIP(localIP, INET_ADDRSTRLEN);

memset(&sa, 0, sizeof(struct sockaddr_in));
sa.sin_family = AF_INET;
sa.sin_port = htons(7001);
inet_pton(AF_INET, localIP, &sa.sin_addr.S_un.S_addr);

result = bind(SendSocket, (struct sockaddr *) &sa, sizeof(struct sockaddr_in));
if (result != 0) {
    printf("bind failed with error: %d\n", result);
    exit(EXIT_FAILURE);
}
```

3.7.3 Retrieval of SHARP Packet in Kernel Space

Once the SHARP daemon has constructed the new SHARP packet, it creates a new UDP connection over SHARP client-reserved ports and sends the packet to the first intermediate SHARP node in the path. SHARP traffic always has a source port of 7001 and a destination port of 7000. SHARP traffic is easily identifiable by comparing source and destination ports to these values. Since the SHARP daemon generates a new connection to send out the SHARP packet, that packet will be seen by both our outbound ALE layer and network callouts. These packets must pass through the callouts with no modification, since they are *already* SHARP packets. To serve this purpose, short-circuit logic exists in both callouts which permits SHARP packets to continue through the kernel queue. Our custom packet-parsing API sets an explicit flag in its output structure to signal that it has parsed a SHARP packet. The code snippet below shows how that flag gets set:

```
if (data->SHARPHeaderTransportType != UDP_PROTOCOL
       || data->SHARPHeaderSourcePort != OUTBOUND_SHARP_PORT
       || data->SHARPHeaderDestinationPort != INBOUND_SHARP_PORT)
  {
     //this is not a SHARP packet

     /* adjust other parameters to reflect a non SHARP packet... */

     data->isASHARPPacket = 0;
     return STATUS_SUCCESS;
  }
  //this is a SHARP packet!
```

```
data->isASHARPPacket = 1;
```

A SHARP packet must be using UDP, have a source port of 7000, and have a destination port of 7001. Once the packet is parsed, both callouts need simply to check whether the SHARP packet flag is set. If set, the callout permits the packet and deallocates the data buffer which held the copied packet before returning. Since SHARP packets are allowed through the higher layer callouts, they can reach the Ethernet layer callout untouched and recieve a VLAN tag.

3.8 Intermediate SHARP Hops

All SHARP packets are processed by the user-space daemon; however, those SHARP packets can come from one of two sources. The daemon can receive packets either from SHARP daemons running on other machines in the network, or from WinSightService. To handle incoming SHARP packets, the SHARP daemon has a third thread, called *SHARPWorker*. The *SHARPWorker* thread operates similarly to *ServiceWorker* in that it creates a UDP socket (bound to port 7000 in this case), enters an infinite `while` loop and blocks on a call to `recv()` therein. Since TCP stitching is performed at the kernel level in our implementation, one invariant is that packets will only arrive at *SHARPWorker*'s socket if the machine on which the SHARP daemon is running is not the last SHARP node in the SHARP packet's route. Because of this, once *SHARPWorker* receives a packet, it simply adds it to the global packet queue and returns to blocking on a `recv()` call. The *ConsumerWorker* then removes the packet from the queue, increments the SHARP header *Index* field, and determines where to send the packet next.

3.9 Handling SHARP Destinations

Once a SHARP packet has completed its route, the payload housed within the packet must be sent to the application layer of the destination node. It is incumbent on kernel space to do this, since the application layer listens on ports for inbound packets coming through the network queue. While it might be feasible to have a user space application recieve SHARP packets, unwrap them, and proxy information to the intended application, such a solution would be slow and unnecessarily complex. It is more sensible to leverage the network queue, and modify packets as they pass toward the application layer. Thus, a WFP callout exists on the inbound path to handle cases where the SHARP path has ended. The callout exists at the inbound IP-v4 network layer, which exposes Layer 3 and Layer 4 information of inbound IP traffic.

As explained in the Section 3.2 of the report, behavior at the end of a SHARP path can vary based on the final destination of the packet. If the final destination is itself a SHARP node, the payload need only be unwrapped from the packet and sent to the application layer. If, on the other hand, the final destination is a non-SHARP node, the final SHARP node in the path must proxy a connection between itself and the final destination. Proxy behavior is described in detail in the next section. The remainder of this section will describe the callout's behavior when the final destination is a SHARP node.

The inbound network callout, like the rest of our WFP callouts, follows the same basic logic flow. Packets are received and converted into a contiguous data buffer. Those data buffers are parsed for relevant information, which informs modifications on those data buffers. Finally, the buffer gets repackaged into a packet, and that packet gets reinjected back into the inbound network queue. The inbound callout first converts and parses incoming packets using the same logic described in the common kernel module functions section of the implementation.

Before describing the behavior of the callout after the packet is parsed, it is important to note a peculiarity of the inbound callouts which must be handled when copying the packet into a fresh buffer. The WFP has to parse each header of the inbound packet to determine the filtering layer it belongs to. For example, the WFP has to scan the Layer 3 header of an incoming packet to confirm that it is

an IP header and send it to the inbound IP-v4 callout. When the WFP scans that header, it moves the index which the head of the packet *beyond* the scanned header, and does not reset that index. This means that inbound callouts receive packets at a layer *above* their filtering layer; in the case of our IP example, the inbound IP-v4 packet would be indexed at the start of the transport header (whether TCP or UDP). Fortunately, the WFP offers an API which makes the IP header available again to the callout. In particular, two functions, called `NdisRetreatNetBufferListDataStart` and `NdisAdvanceNetBufferListDataStart`, move the index pointing to the start of the buffer backward and forward, respectively. For the purposes of the inbound IP-v4 callout, incoming packets are retreated by the length of an IP header to make the packet's IP header accessible. Conversely, the packet must be advanced back to the transport header when letting the packet back into the network queue.

Once the packet has been copied into a buffer and parsed, the callout makes a decision on whether or not the packet's payload is destined for its host. To do so, several parameters of the packet are checked. First, the the callout determines whether the packet belongs to SHARP. The unified parsing API returns a bitfield which is asserted if the scanned packet is a SHARP packet; the callout relies on this field to make its decision. Only SHARP packets can have payloads, so the callout only proceeds if the packet is indeed a SHARP packet. Next, the callout checks whether the SHARP packet has reached the last hop on its path. The SHARP header stores a hop parameter and a length parameter. The index parameter of the packet is incremented every time it passes through the user-space of an intermediate node; thus, by the time the packet reaches its final node, the index parameter is *one less* than the hop parameter of the packet. As a result, the kernel module checks if the current hop parameter is one less than the length of the packet to determine if the packet is at the end of the SHARP path. The following code snippet shows how this condition is checked:

```
if (pd->SHARPHeaderIndex < (pd->SHARPHeaderHops - 1)) {

    //release the buffer and allow the packet through
    WdfSpinLockAcquire(AtomicMemoryManagementLock);
    ExFreePoolWithTag(buffer, BUF_TAG);
    WdfSpinLockRelease(AtomicMemoryManagementLock);

    ClassifyOut->actionType = FWP_ACTION_PERMIT;
    if (Filter->flags & FWPS_FILTER_FLAG_CLEAR_ACTION_RIGHT) {
        ClassifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
    }
    return;
}
```

Finally, the destination flags of the SHARP header are inspected. As explained in the SHARP header section of the implementation, the destination type flag of the packet is asserted when the true destination of the packet is a SHARP node. Thus, if that flag is asserted, the kernel module can absorb the packet's payload back into the application layer. Otherwise, the callout must enter a proxy connection with the true destination, as described in the next section.

To absorb the payload into the application layer, the payload of the SHARP packet is copied over to a freshly allocated buffer, and the original packet buffer is freed from memory. This freshly allocated payload buffer will itself become packaged a new packet which gets reinjected back into the inbound kernel queue. The following code snipped demonstrates this copying operation:

```
UINT payloadStartIdx = pd->payloadTransportType;
     UINT payloadLength = packetLength - payloadStartIdx;

     //create a new buffer which holds the payload of the SHARP packet
     WdfSpinLockAcquire(AtomicMemoryManagementLock);
```

```
payload = ExAllocatePoolWithTag(NonPagedPool, payloadLength, BUF_TAG);
WdfSpinLockRelease(AtomicMemoryManagementLock);

RtlZeroMemory(payload, payloadLength);
RtlCopyMemory(payload, &(buffer[payloadStartIdx]), payloadLength);
```

Once the payload gets extracted from the SHARP packet, both its IP and TCP/UDP checksums must be calculated. Since the payload was wrapped with a SHARP packet before it ever left the true source machine, the network interface card *never* processes the packet. It is the NIC's responsibility to calculate the checksums of outbound packets; the payload, left untouched by the NIC, will never have its checksums calculated. Thus, our system manually calculates the checksums for the payload headers using the same functions we discuss in Section 3.6.4.

3.10 Non-SHARP Destination Proxying

In an ideal world, every client would adopt and implement the underlying SHARP protocol, such that the client could send and receive SHARP packets. Unfortunately, at least in the foreseeable future, clients and servers in the wider network may not be running SHARP; thus, the protocol must adjust to handle connections between those SHARP-less devices. While this is possible, it limits the extent to which packet routes can be controlled. Once a SHARP node sends a packet to a non-SHARP client, all control over that packet's path is lost. Therefore, the SHARP protocol allows for only the *final* node in a chain, i.e. the true destination, to be a non-SHARP node. To implement this, extra logic exists within the kernel which allows the last SHARP node in the chain to spoof a fresh, non-SHARP connection to the final destination. This node is responsible for translating between the incoming SHARP connection and the outgoing non-SHARP connection.

In our case, spoofing a connection is similar to a primitive version of network address translation. A non-SHARP node, known henceforth as the *proxy destination*, does not understand what to do with a SHARP packet. The final SHARP node in the chain, known henceforth as the *proxy source*, must extract the payload from the SHARP packet and send the payload itself. However, the payload holds the IP address of the true source node, as that node generated it. The proxy source must replace the payload's source IP address with its own address, and send it out to the proxy destination. When the proxy destination responds, it sends a non-SHARP packet back to the proxy source. The proxy source must replace the destination IP address of the incoming packet with the true source's address, load the packet as the payload of a new SHARP packet, and send that SHARP packet back through the SHARP network to the true source. The proxy source thus handles translating packets from within the SHARP network and packets from the wider network. The diagram below gives a step-by-step view of the destination proxy logic:

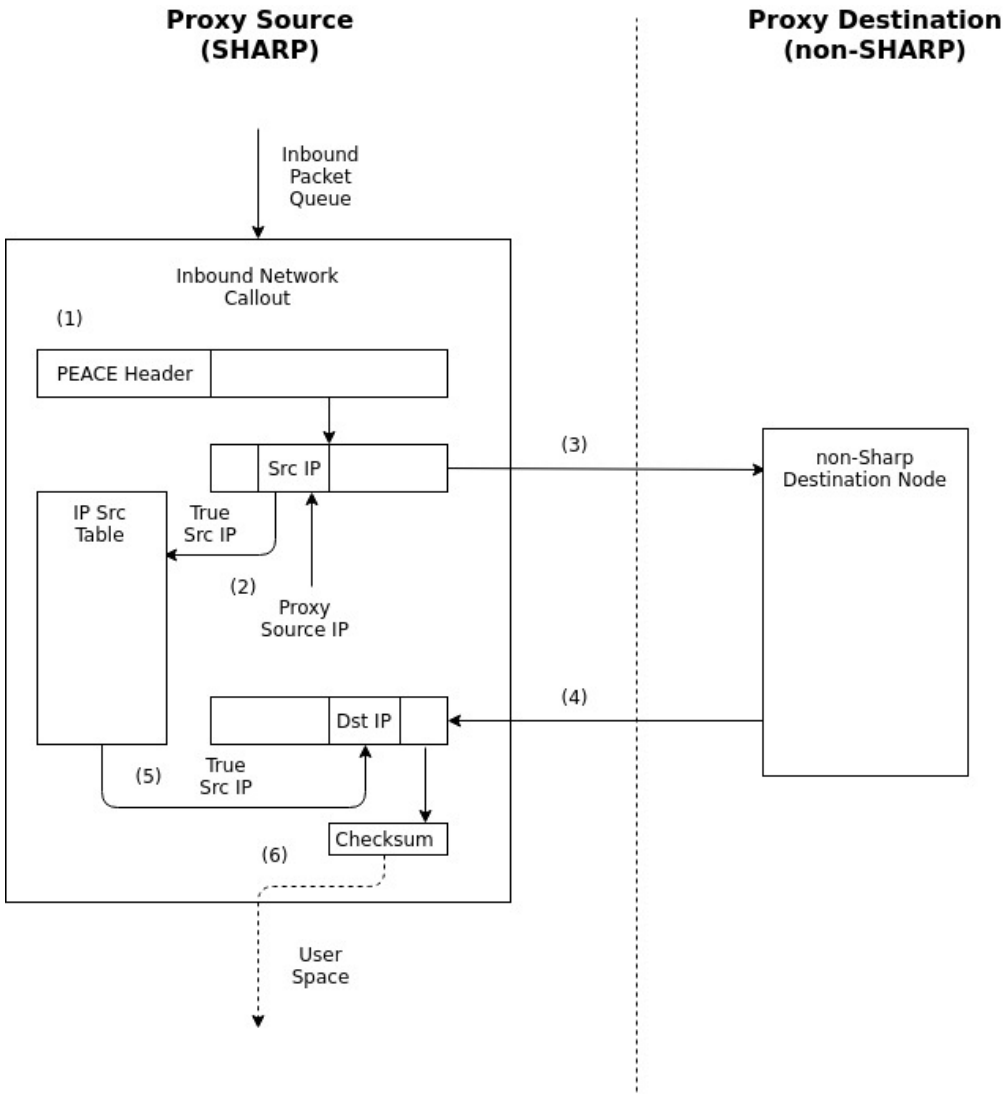


Figure 15: The code logic diagram for destination proxying.

Each step of the control flow is labeled in the diagram, and described in detail below:

1. The packet starts at the last SHARP node in the chain, and is intercepted from within the inbound packet queue. The inbound queue loads the packet into a data buffer and parses it.
2. The callout recognizes that it needs to proxy with a non-SHARP node. It strips away the SHARP header, leaving only the initial packet payload.
3. The callout stores the source IP of the payload, which is the true source's IP address. It replaces it with its own IP address. With the IP address spoofed, the SHARP node can proxy a connection to the non-SHARP node.
4. The SHARP node sends the modified payload to the proxy target. Since the payload enters the network interface card, checksums do not need to be manually calculated.
5. The proxy target sends a return message back to the SHARP node. The packet enters the kernel queue, and is intercepted once again by the callout.

6. The destination IP of the incoming packet, which holds the IP address of the proxy source, is swapped with the true source IP address which was saved in step 2. The packet can now be given a SHARP header and sent back on the reverse path.

7. The IP checksum is manually recalculated for the packet, which will become a new SHARP payload. The payload is then sent to user space to receive a SHARP header and move along the reverse SHARP path.

The proxy logic is implemented at the network filtering layer, since a fully-formed IP packet must exist to be manipulated. The implemented sits on the inbound path by necessity, for two reasons. First, the callout can intercept and toss SHARP packets destined for proxying before user-space unnecessarily sees them. Second, the callout can receive return packets from non-SHARP destinations and prepare them for the SHARP path before user-space sees them. Generally, user space should be unaware that a proxy connection is happening within the kernel modules; all it should do is satisfy requests to create SHARP connections. The rest of this section goes into greater detail on how each segment of the proxy logic is implemented within the inbound network callout.

3.10.1 Preparing a SHARP Packet for the Proxy Connection

The inbound network callout shares the same general implementation pipeline of the other callouts in the kernel. First, an intercepted packet is copied into a data buffer. Next, it is parsed. Finally, some operation happens on the buffer, and either the modified packet is reinjected to sent to user space. However, while the other callouts each implement just one component of SHARP's logic, the inbound network callout implements several features which together handle destination behavior. As a result, the the inbound network callout checks for several conditions in the packet before chosing what behavior to execute. Again, the incoming packets have to be rewound back to the IP header before they are copied into a buffer. They are advanced back to the Layer 4 header if the original packets are ever let back onto the inbound network queue.

Once the packet has been copied over into a buffer, it is parsed using the unified parsing API. If the parsing API fails, the packet is blocked from rejoining the network queue; this is to ensure that SHARP's user space modules do not log malformed packets. Then, the callout checks if the packet is a SHARP packet or not. This distinction is significant because it accurately divides packets which need to begin the proxying process and packets returning from a proxied non-SHARP destination. The callout continues to prepare the proxy connection only if the packet at hand is a SHARP packet. The packet is then checked to determine if it has reached the final SHARP node. If it has not, the packet must be let through to the SHARP user-space modules to send it to the next node on the SHARP path. Since it is the SHARP daemon's responsibility to increment the current hop field in the packet, the kernel module will intercept the packet on the inbound path *before* user space gets a chance to properly increment its value. As a result, the kernel module checks if the current hop parameter is one less than the length of the packet to determine if the packet is at the end of the SHARP path. The following code snippet shows how this condition is checked:

```
if (pd->SHARPHeaderIndex < (pd->SHARPHeaderHops - 1)) {

   //release the buffer and allow the packet through
   WdfSpinLockAcquire(AtomicMemoryManagementLock);
   ExFreePoolWithTag(buffer, BUF_TAG);
   WdfSpinLockRelease(AtomicMemoryManagementLock);

   ClassifyOut->actionType = FWP_ACTION_PERMIT;
   if (Filter->flags & FWPS_FILTER_FLAG_CLEAR_ACTION_RIGHT) {
       ClassifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
   }
```

```
    return;
}
```

In the snippet above, if the packet has not reached the final SHARP node, the kernel module relinquishes control of the packet and allows it to pass to the user-space daemon. If the packet is indeed at the final SHARP node, the callout finally inspects the SHARP header flags. Specifically, it checks the destination type flag as described in the SHARP header section of the implementation. If the flag is asserted, the SHARP node is at the intended final destination and can be returned to the application layer of the node. Otherwise, the packet still needs proxying to a non-SHARP true destination.

If the packet passes all the checks mentioned above, the callout initiates the proxying process on the packet. The callout first strips the SHARP header from the packet, isolating the packet's payload. This task is made easier thanks to the universal parsing API, which marks the the beginning of the payload in the buffer. A new buffer is allocated to hold the isolated payload, and a call to `RtlCopyMemory` places the payload in a new buffer, where it can act as an entirely new packet.

To prepare for proxying, the callout retrieves the host machine's IP address, which is used later to forge the source IP address of the payload. A helper designed for this task, called `MQPKERNEL_LookupStitchSourceAddress`, accomplishes this by finding the reference to the host machine in the SHARP header. If a packet has arrived at a SHARP node, that packet must have the node's IP somewhere in its header. In the case of proxying, that IP will always be the *second to last* address in the header. The code snippet below implements this lookup:

```
UINT MQPKERNEL_LookupStitchSourceAddress(_In_ PCHAR buffer, _In_ UINT packetLength, _In_
    MqpPacketData* data)
{
   UNREFERENCED_PARAMETER(packetLength);
   UINT offset = data->SHARPHeaderStartIndex + 6 + (6 * data->SHARPHeaderHops - 1);
   return *(PUINT)(&buffer[offset]);
}
```

Once the local IP address has been discovered, the module sends the original packet to user space. As explained in the previous section, whenever the SHARP daemon receives a SHARP packet from WinSightService, it prepares a reversed path from the host node back to the original source of the packet. When the proxied packets return, they use the prepared SHARP path to navigate back through the SHARP network to the true source node. The forging process itself requires only a call to *RtlCopyMemory* at a 12 byte offset into the payload; it copies the retrieved local IP address into the buffer location where the source IP address exists.

Next, the original IP of the payload must be indexed in a table for later reference. In fact, the *ALE lookup table* is re-purposed to support this task; the details of how this works can be found in the next section. For now, it is sufficient to understand that a new entry exists in the ALE which contains both the IP address of the proxy destination and the IP address of the original source. Once the lookup table has been updated, the payload can be re-injected back into the outbound network queue, so that it can reach the proxy destination. As usual, the WFP offers an API call to reinject packets into the outbound queue at the network layer: `FwpsInjectNetworkSendAsync0`. This reinjection follows the same logic flow as re-injection in the rest of the callouts, and is described in detail in the Common Kernel Module Functions section of this report. If the packet re-injects successfully, the SendComplete callback function frees all allocated buffers and blocks the original packet. Otherwise, the payload is tossed, and cleanup is handled manually in the callout itself. It should be noted that before the packet gets repackaged and reinjected, its IP checksum and UDP/TCP checksums must be recalculated. Again, the payload itself never touched a network interface card, so the card never automatically calculated those checksums.

3.10.2 Reusing the ALE Lookup Table for Source IP Storage

We originally intended for a separate table to manage the storage and retrieval of source IP addresses during the proxying process. Generally, we wanted to design compartmentalized code which handles single elements of the functionality robustly. Unfortunately, due to time constraints, we had to apply a more pragmatic approach to complete the implementation of SHARP. In the interest of saving development time, we decided to reuse the ALE lookup table to handle source IP address storage. The ALE lookup table is thus populated with two types of entries: those which signal the kernel module to send a packet to user space, and those which keep track of the true source IP of a payload. Curiously enough, these alternate entries in the table also perform the original task of the lookup table; they signal the inbound callout that a returning packet from the proxy destination needs to be sent to user space to recieve a SHARP header.

The keys of the new entries are still five-tuples which represent the returning proxy connection (i.e. the connection from the proxy destination to the source). When the inbound callout begins a proxy connection, it *proactively* populates the table with this key by flipping the values of the outgoing proxy packet. Specifically, the source and destination ports and addresses of the outgoing packet are swapped when generating the key. The code snippet below demonstrates how this is done:

```
//the return connection's source IP is the device we are stitching to
//the return connection's destination IP is us.
UINT32 sourceIP = pd->payloadDestinationIpAddress;
UINT32 destIP = localIpAddress;

//the ports are inverted, since we are expecting a return connection
UINT16 sourcePort = pd->payloadDestinationPort;
UINT16 destPort = pd->payloadSourcePort;
UINT8 protocol = pd->payloadTransportType;

FiveTuple key;
key = buildFiveTuple((ULONG64)sourceIP, (ULONG64)destIP,
      sourcePort, destPort, protocol);
```

In this snippet, the payload destination becomes the source IP, and the proxy source becomes the destination IP. The ports also are swapped in the same way. Once the key has been generated, the entry is populated into the table. However, the value associated with the entry *is the true source IP address, not a simple boolean flag.* This is the modification which separates these alternate entries from their regular counterparts; the values of these entries are used to recall the true source of the connection. Since an IP address can be represented as a positive unsigned integer, the callout will still recognize that the packet associated with the table entry needs to be sent to user space. The inbound callout's behavior based on lookup value is summarized below:

- **lookup equal to 0**: Release the packet, and do nothing with it.
- **lookup equal to 1**: Send the packet to user space, but do not use the lookup value as an IP address.
- **lookup greater than 1**: Send the packet to user space, and also use the lookup value as an IP address.

During the proxying process, once the true source IP is copied out of the payload buffer, it gets loaded as the value of a new entry in the ALE lookup table. Thus, the lookup table both maintains that the return packet must be sent to user space and that the return packet must have its IP address swapped out with the true source's address.

3.10.3 Receiving Packets from the Proxy Destination

Once a packet returns from the proxy destination, the inbound callout will again be the first thing (other than the Ethernet-layer callouts) to recieve that packet. It is the responsibility of the inbound network callout to prepare the packet for the return connection over the SHARP network. The callout will only try to handle a returning proxy packet if that packet does not have a SHARP header. To determine whether the packet truly came from a proxy destination, the ALE lookup table is checked, using the five-tuple of the incoming packet. As explained in the previous section, the reverse parameters of the outgoing packet were used to proactively store the reverse connection in the table. The five-tuple of the return packet will match this proactively-made key. If an entry does indeed exist in the table, and the value associated with that entry is an IP address, the callout knows it is handling a proxy response and can proceed with preparing the packet for the return path across the SHARP nodes.

Luckily, most of the functionality which is necessary to initiate a SHARP connection in user space can be reused to prepare the return path of the proxied connection. The inbound network callout need only replace the destination IP of the incoming packet, which is the address of the local node, with the IP of the true source. The packet can then be sent wholesale to user space, and sent back through the SHARP network. The original packet is then blocked, since a fresh packet with be sent out from user space. Of course, if the incoming packet is not part of a proxied connection, the callout simply allows it to pass through the network queue unaltered.

4 Results

In this section we demonstrate the Layer 2 and 3 routing control offered by the SHARP protocol as described in Section 3. To do this, we create a testbed network with VLANs configured as specified in Section 3.2, and we install our implementation of the SHARP client on machines connected to this network. Because encapsulation as a mechanism for enabling Layer 3 routing control is already in wide use (e.g. GRE, IP-in-IP, IPSec, VPN protocols), we primarily focus on proving that our design allows endpoints to control their packets' paths at the Ethernet layer by choosing from a set of predefined VLAN tag IDs.

4.1 Experiment Overview

We designed a set of two experiments that demonstrates basic SHARP packet encapsulation and header processing, but highlights the Layer 2 routing control offered by SHARP. The network design for the experiment is shown in Figure 16.

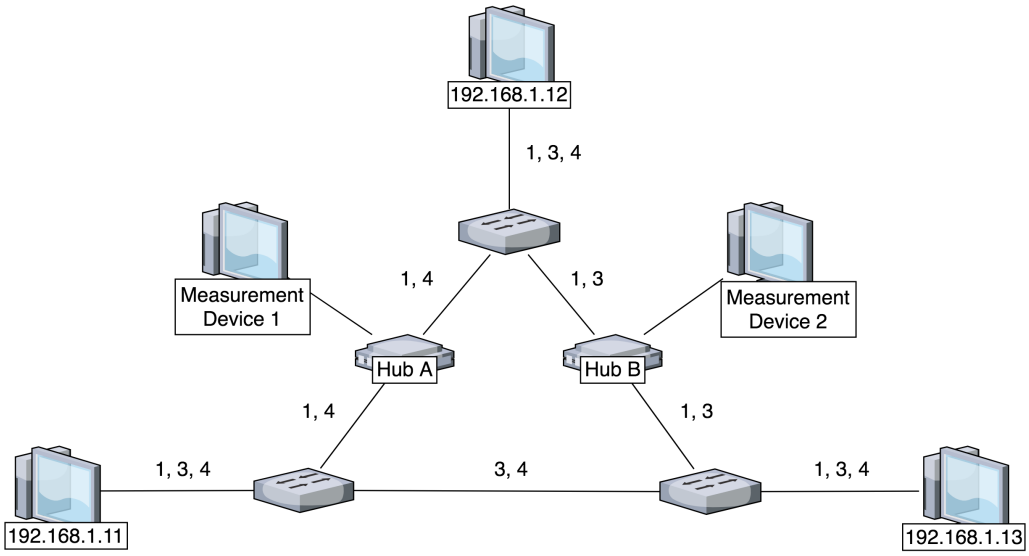


Figure 16: The testbed network design used for conducting Experiments 1 and 2. The network contains three VLANs (1, 3, and 4), which each define a unique spanning tree for the network.

The network consists of three endpoints that are interconnected by a series of 3 switches and 2 hubs. Each switch has trunk ports that support the VLAN IDs shown next to them. For instance, the uppermost switch is connected to the machine with IP address `192.168.1.12` by a trunk port that accepts VLAN tags with IDs 1, 3 or 4. VLANs 1, 3, and 4 each comprise two out of the three links in the inner triangle of switches, thus implementing all unique, valid spanning trees for the network.

The network design includes hubs as a method for monitoring traffic that crosses a particular link. Initially, we expected to be able to run `tcpdump` on each of the switches to watch SHARP packets traverse the network; however, our initial testing showed that `tcpdump` could only capture traffic that was destined for the switch itself. Because the switch's packet forwarding functionality can be performed at the hardware level, software such as `tcpdump` is never able to capture these packets. Thus, in order to observe SHARP packets in transit, we introduced the hubs between two of the switch pairs. When a hub receives bits on one of its ports, it simply broadcasts those bits on all other active ports. To capture traffic passing through a hub, we simply connect a computer

running *Wireshark*. In this experiment, since the switches on either end of the hub will expect traffic with VLAN tags, the recording computer will only be able to observe traffic, and cannot send traffic of its own past either switch. Since the recording computers are not meant to engage with the other machines on the network for the experiment, this is ideal.

As preparation for each experiment, a rule is implemented at the outbound ALE layer of the SHARP client's kernel driver (see Section 3.5) indicating that connections with destination port 80 and destination IP address `192.168.1.13` require a SHARP header. Also, Python3 is installed on the machine at `192.168.1.13` and the command `python3 -m http.server` is executed to start an HTTP server. The experiments are then conducted as follows:

1. Install the SHARP client on each of the endpoints.
2. Start Wireshark on each of the endpoints, as well as on the monitoring devices connected to the hubs.
3. Execute `curl 192.168.1.13:80` on `192.168.1.11` to initiate an TCP SYN request for an HTTP connection.
4. Wait for the SHARP-header encapsulated request to reach `192.168.1.13`.
5. Kill the `curl` command.
6. Stop Wireshark and save the capture to a `.pcapng` file.
7. Uninstall the SHARP client and save log files.

The key (and only) difference between the two experiments is the SHARP header that is appended to the outgoing TCP SYN packets on `192.168.1.11`. The header for each experiment is shown in Figure 17.

SHARP Header - Experiment 1

3	0	1	192.168.1.11	3	192.168.1.12	4	192.168.1.13

SHARP Header - Experiment 2

3	0	1	192.168.1.11	4	192.168.1.12	3	192.168.1.13

Figure 17: The SHARP header appended to the outgoing TCP SYN packet from `192.168.1.11` to `192.168.1.13` in each experiment. Note that the VLAN IDs between each node have been swapped.

In Experiment 1, the SHARP packet will be tagged with a VLAN ID of 3 when leaving `192.168.1.11`, and thus should pass through Hub B (but not Hub A) on its way to `192.168.1.12`. On the other hand, in Experiment 2 the SHARP packet should travel through Hub A instead of Hub B on its way to `192.168.1.12`. If we show that this behavior does occur, then we will have demonstrated that a host-based SDN can offer the same Layer 2 routing control as a switch-based SDN, and it is as simple as changing the VLAN tag on an outgoing packet to change the entire path a packet takes through the LAN.

After capturing traffic for each experiment in `.pcap` files, we will apply the display filter `udp and udp.port == 7000` to ensure that only traffic containing SHARP packets will be shown.

4.1.1 GNS3

Due to limited hardware availability, we initially aimed to conduct our experiments with a virtual testbed, comprised of logically interconnected VMs. We identified Graphical Network Simulator 3 (GNS3) as an application that would potentially suit our needs. GNS3 is a network emulation software which allows users connect switches, routers, and endpoint machines via a drag-and-drop interface [26]. Importantly, in GNS3 users can import VM images to use as endpoints in the virtual

network; in order to install our own software, we needed the testbed application we chose to offer this feature.

Because the PEACE software off of which we implemented SHARP is proprietary, we performed the development and debugging for SHARP on Windows 10 VMs hosted on a WPI Linux server. The Linux server used QEMU and KVM for its hardware virtualization and so our VM images were stored in the `.qcow2` file format, which was one of the formats supported by GNS3. We found quickly that our Windows 10 VM's 70 GB storage was too large for GNS3 to handle, as the application would become unresponsive after we tried to add the VM to the list of available endpoints. By creating a new Windows 10 VM with only 20 GB of hard disk space, we were able to successfully add a VM with SHARP and PEACE installed to a sample network in GNS3. Despite opting for a more lightweight VM, the GNS3 application was still somewhat slow and unwieldy to work with.

What ultimately caused us to abandon the idea of a virtual testbed was the fact that were unable to set up VLAN trunking on the virtual network. Trunk ports are essential to the SHARP design, and GNS3's default switch element does not support VLAN trunking. It is possible to add more advanced, proprietary switch elements to GNS3 by importing images such as the Cisco VIRL IOSvL2. However, obtaining the Cisco VIRL IOSvL2 image required a privileged Cisco account, and we decided it was not worth our time to pursue the virtual testbed option any longer.

4.1.2 Physical Testbed Construction

Having opted to create a physical testbed, we needed to acquire various pieces of hardware to construct the network shown in Figure 16. We borrowed Windows 10 laptops to serve as the SHARP-enabled endpoints from WPI's Academic Technology Center, and although hubs are not often used in production networks anymore, we were able to find professors in the Computer Science department who still had some and graciously loaned them to us. While we were unable to acquire switches for the experiment, we borrowed three TP-LINK Archer C7 routers from the department. By replacing the stock firmware with *OpenWRT*, we gave the routers VLAN support. By disabling DHCP on the routers and using only the LAN ports, they performed essentially the same function as switches. In Appendix A we detail the process of installing *OpenWRT* and `tcpdump` on the routers, and how to create VLANs and assign them to access and trunk ports. Figure 18 shows the final physical network setup.

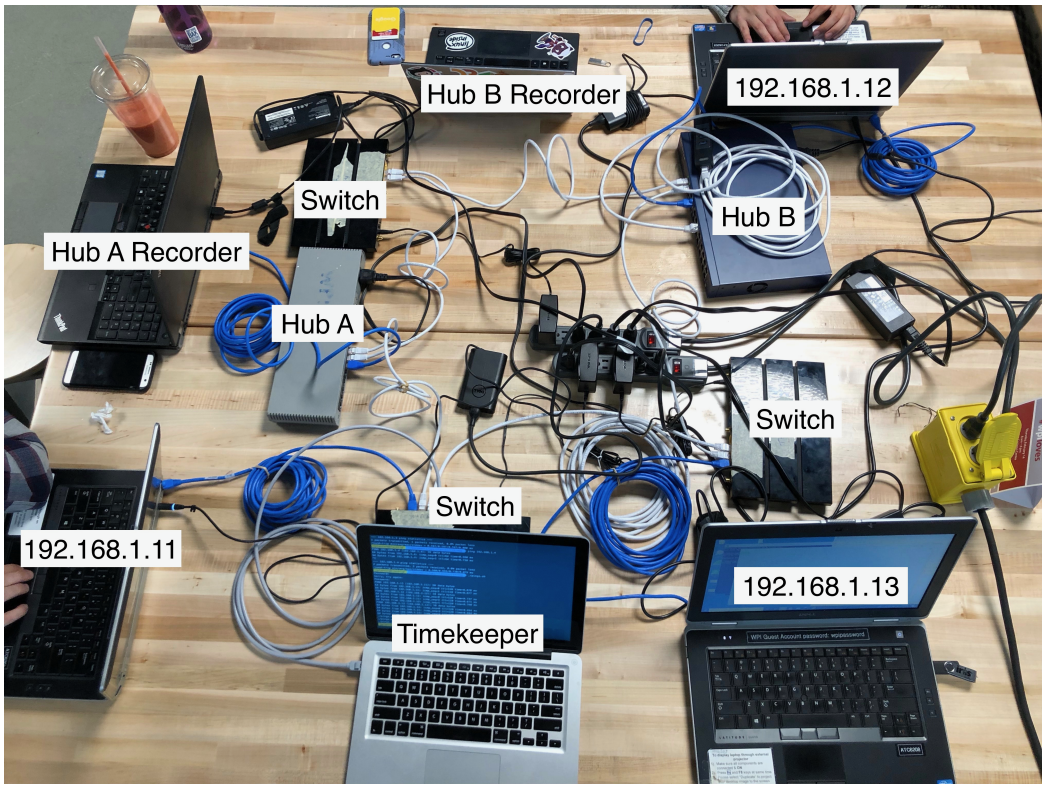


Figure 18: The final physical implementation of the testbed network used for conducting Experiments 1 and 2.

4.1.3 Experiment 1

We now present the results for Experiment 1. Figure 19 shows the SHARP daemon's *ServiceWorker* thread receiving the TCP SYN packet created by the `curl` command. The *ServiceWorker* then checks the SHARP header table for an appropriate header, and finding none, appends the generic SHARP header. The *ConsumerWorker* thread then indicates that it has received the packet, and as specified in the SHARP header, creates a UDP socket in order to send the packet to `192.168.1.12`.

```
C:\PEACE>WinSightEmissary.exe 192.168.1.11
UPDATE: WSAStartup succeeded.
----------------------------------------
[PBA] 52
[Service][Received this] 0x45 0x00 0x00 0x34 0x47 0xb2 0x40 0x00 0x80 0x06 0x2f 0xa9 0xc0 0xa8 0x01 0x0b 0xc0 0xa8 0x01
0x0d 0xc2 0x15 0x00 0x50 0x1e 0xdd 0xce 0x3a 0x00 0x00 0x00 0x00 0x80 0x02 0x20 0x00 0x98 0x9a 0x00 0x00 0x02 0x04 0x05
0xb4 0x01 0x03 0x03 0x08 0x01 0x01 0x04 0x02
[Service] No key found. Appending generic SHARP header.
[PBA] 74
[Consumer][Packet] 0x00 0x03 0xff 0xff 0x00 0x01 0xc0 0xa8 0x01 0x0b 0x00 0x03 0xc0 0xa8 0x01 0x0c 0x00 0x04 0xc0 0xa8 0
x01 0x0d 0x45 0x00 0x00 0x34 0x47 0xb2 0x40 0x00 0x80 0x06 0x2f 0xa9 0xc0 0xa8 0x01 0x0b 0xc0 0xa8 0x01 0x0d 0xc2 0x15 0
x00 0x50 0x1e 0xdd 0xce 0x3a 0x00 0x00 0x00 0x00 0x80 0x02 0x20 0x00 0x98 0x9a 0x00 0x00 0x02 0x04 0x05 0xb4 0x01 0x03 0
x03 0x08 0x01 0x01 0x04 0x02
[Consumer Worker] Sending to IP: 192.168.1.12
----------------------------------------
```

Figure 19: The SHARP daemon running on `192.168.1.11` receives the TCP packet from WinSightService. The destination port for the packet is 80 (A), and the flags in the TCP header indicate that it is a SYN packet (B). After the *ConsumerWorker* thread receives the packet, it prepends a generic SHARP header (C), and sends the encapsulated packet over UDP to `192.168.1.11`.

Figure 20 demonstrates the packet leaving `192.168.1.11` on its way to the SHARP node at `192.168.1.12`. As per the SHARP header, the outbound Ethernet callout on `192.168.1.11` has

inserted a tag for VLAN 3 into the Ethernet header.

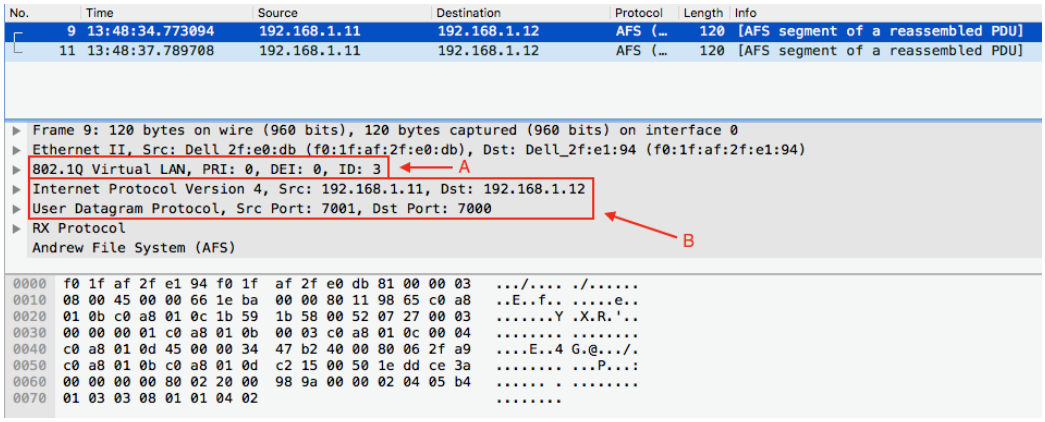


Figure 20: A Wireshark capture shows the SHARP packet travelling from UDP port 7001 on `192.168.1.11` to UDP port 7000 on `192.168.1.12` (B). The packet has a VLAN tag with ID 3 inserted into the Ethernet frame header (A).

As Figure 21 shows, Hub B captures the packet while it travels between `192.168.1.11` and `192.168.1.12`, but has no record of the packet as it travels between `192.168.1.12` and `192.168.1.13`, since the VLAN tag used between those nodes will specify VLAN 4.

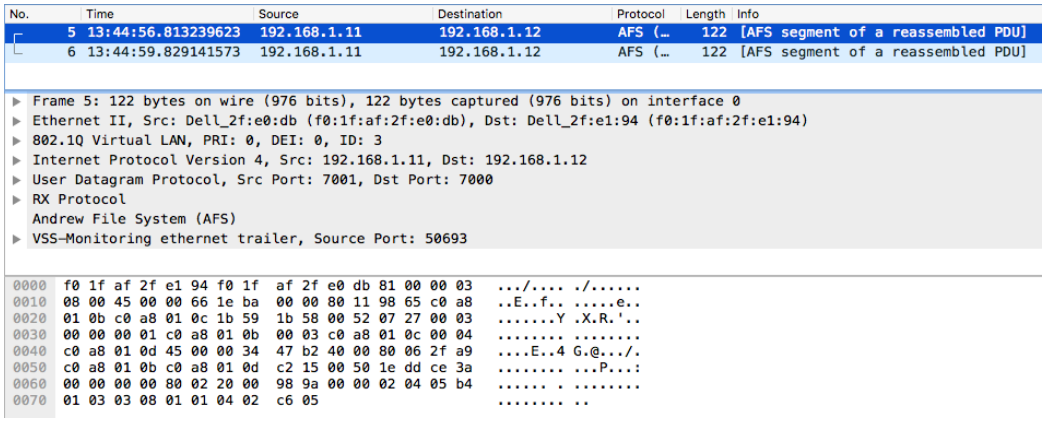


Figure 21: A Wireshark capture shows the SHARP packet travelling across Hub B as it travels from `192.168.1.11` to `192.168.1.12`.

Once the packet reaches `192.168.1.12`, it is received by the *SHARPWorker* thread which was listening on UDP port 7001 for any incoming SHARP packets. The *ConsumerWorker* thread then increments the index field of the SHARP header and sends the packet to `192.168.1.13` in accordance with the path defined by the SHARP header. As the encapsulated packet travels through `192.168.1.12` kernel networking send path, it triggers log prints as shown in Figure 22.

```
1550601134: PEACE Driver: Debug: Payload source port:
1550601134: PEACE Driver: Debug: 49674
1550601134: PEACE Driver: Debug: Payload dest port:
1550601134: PEACE Driver: Debug: 80
1550601134: PEACE Driver: Debug: Outbound Ethernet: SHARP packet detected, appending tag with following VLAN ID:
1550601134: PEACE Driver: Debug: 4
1550601134: PEACE Driver: Debug: Outbound Network: letting SHARP packet through.
```

Figure 22: Log file showing the outbound processing of the SHARP packet on `192.168.1.12`. Due to print buffering, the order of print statements is slightly incorrect. However, the log still displays the outbound network callout permitting a packet that it recognizes as having a SHARP header, and shows the outbound Ethernet callout determining to append a tag for VLAN 4 due to the index value in the SHARP header.

Once `192.168.1.12` writes the SHARP packet onto the wire with a tag for VLAN 4, the network switch connected to it directs the packet through the port connected to Hub A. Figure 23 shows the Wireshark capture for Hub A. Finally, the SHARP packet arrives at `192.168.1.13`, as shown in Figure 24.

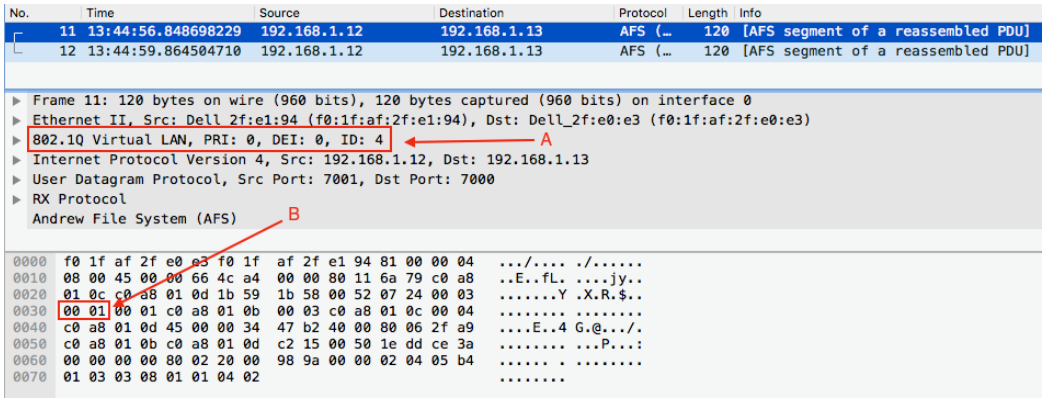


Figure 23: The SHARP packet captured by Hub A on its path between nodes `192.168.1.12` and `192.168.1.13`. The packet has a VLAN tag of 4 (A), and the index value of the SHARP header has been visibly updated from 0 to 1 by `192.168.1.12` (B). Note that Hub A has no record of the packet as it travelled between nodes `192.168.1.11` and `192.168.1.12`.

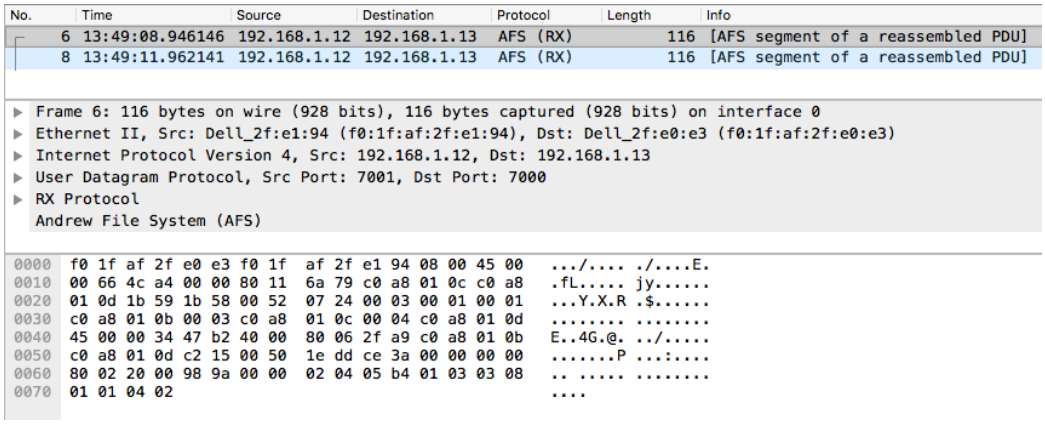


Figure 24: The SHARP packet arrives at its final destination.

4.1.4 Experiment 2

Having shown that packets will not pass through Hubs A and B if they are not appended with tags for VLANs 4 and 3 respectively, we now swap the VLANs specified in the SHARP header from Figure 17. This will demonstrate that we can selectively decide which path a packet will take through the network simply by modifying the VLAN tag inserted into its Ethernet frame header. Again, the experiment begins with `192.168.1.11` attempting to send a TCP SYN packet for a HTTP connection to `192.168.1.13`. Figure 25 shows the kernel-level log file for `192.168.1.11` as the original packet is intercepted, receives a SHARP header, and permitted by the WFP callouts.

```
1550602893: PEACE Driver: Debug: Test ALE: HTTP connection found, adding to ALE Lookup Table with a 1
1550602893: PEACE Driver: Debug: Outbound Network: Packet needs SHARP header, sending to WinSightService.
1550602893: PEACE Driver: Debug: Outbound Network: letting SHARP packet through.
1550602893: PEACE Driver: Debug: Payload source port:
1550602893: PEACE Driver: Debug: Payload dest port:
1550602893: PEACE Driver: Debug: 80
1550602893: PEACE Driver: Debug: Outbound Ethernet: SHARP packet detected, appending tag with following VLAN ID:
1550602893: PEACE Driver: Debug: 4
1550602893: PEACE Driver: Debug: 49675
```

Figure 25: The kernel-level log file for `192.168.1.11` indicates that the ALE callout (created for testing) flagged the packet to receive a SHARP header. The outbound network callout prints that it sent the packet to *WinSightService* and then, after the packet had received a SHARP header from the SHARP daemon, permits it to continue on the network send path.

The SHARP packet has a VLAN ID of 4 and therefore travels through Hub A, as shown in Figure 26. On the other hand, the packet is never seen by Hub B while on its way to `192.168.1.12` (see Figure 27). We have therefore shown that a host can control which path a given packet takes to reach its destination by modifying the value of the VLAN ID in the tag. Note that Hub A has no record of the packet as it travels between `192.168.1.12` and `192.168.1.13`.

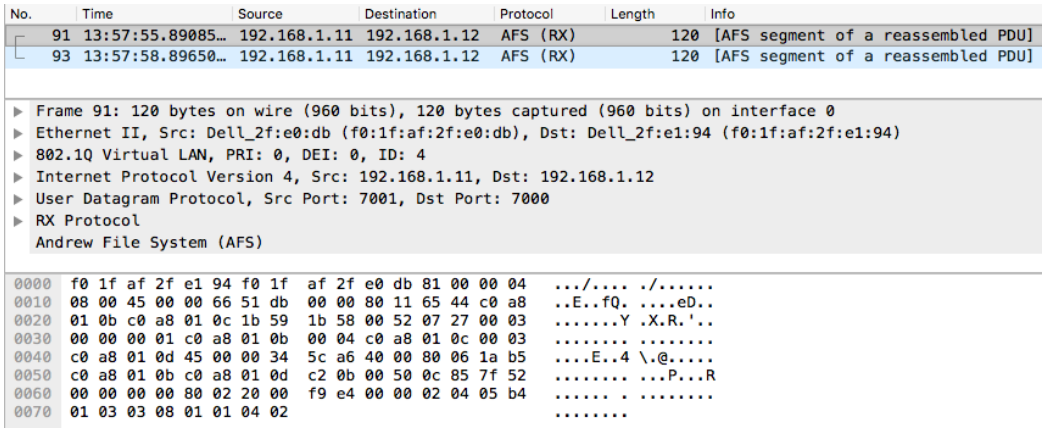


Figure 26: Hub A's capture log shows a SHARP packet destined for `192.168.1.12` with a tag for VLAN 4. Note that the hub has no record of the packet as it travels from `192.168.1.12` to `192.168.1.13` over VLAN 3.

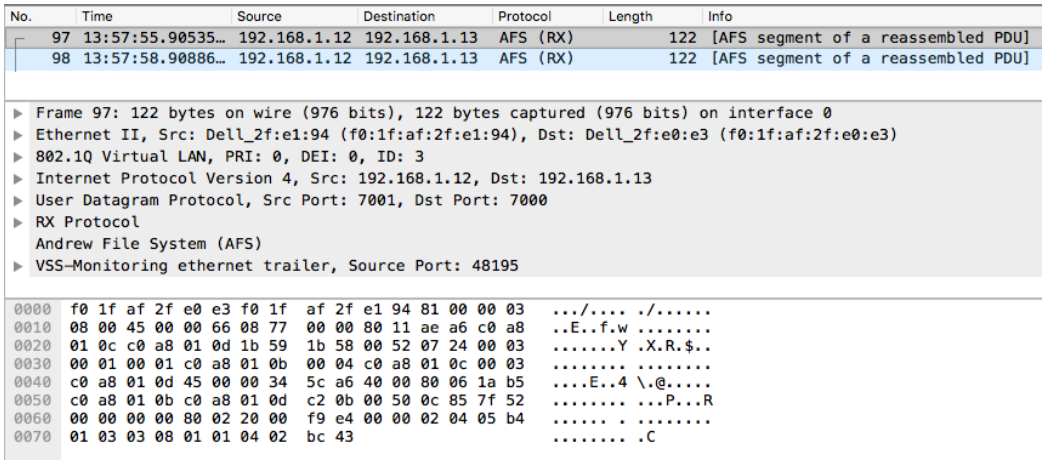


Figure 27: Hub B's capture log shows a SHARP packet destined for `192.168.1.13` with a tag for VLAN 3. Hub B exhibits the same behavior as Hub A in that it never receives the packet between from `192.168.1.11` and `192.168.1.12` over VLAN 4.

First and foremost, the results of Experiments 1 and 2 demonstrate that we can control packet pathing by manipulating VLAN tag IDs. Additionally, the packet captures collected show the selective encapsulation of packets in SHARP headers based on a comparison of flow features to a rule we created. Combined with the proper reception and redirection of SHARP packets by intermediate SHARP nodes, these results prove that we properly implemented all parts of the basic SHARP design except for the proxying behavior and destination packet processing. We discuss the difficulties we encountered in debugging our implementation of these parts of the system in Section 5.

5 Discussion

Here we discuss the limitations of our implementation of SHARP, and potential future additions and improvements to the design.

5.1 SHARP over the Wider Network

The system described in this paper is limited in scope to a single LAN operating under a single SDN controller. The controller maintains a map of the Layer 2 and Layer 3 topology of the LAN it administers, and can thus issue SHARP headers for packets travelling between end hosts within the LAN. However, the controller has no understanding of either the Layer 3 or Layer 2 topology in the wider network, or in a separate LAN; thus, it cannot manipulate the path of a packet once it moves past a gateway router to the wider network. For some institutions, it is permissible for arbitrary routing to be limited to a single LAN, especially for institutions who an afford to locally host an SDN controller. In some cases, however, a smaller institution may wish to have the controller hosted by a separate remote service, and have local clients interface with the controller over the wider network. In these cases, it would be advantageous to carry routing control beyond the local network of the controller.

Ideally, a controller would like to know about the topology of *every* network which has implemented the SHARP SDN, such that it could influence any potential path over the Internet as much as possible. In reality, each controller will only understand the topology of its local network. Controllers could still communicate with each other, in a distributed system, to construct paths which use more than one local network. Such a design would allow controllers to communicate and cache routing decisions between one another. For example, if a controller needed to know VLAN tags and IP addresses to construct hops on a network administered by a different controller, the first controller could retrieve and cache the topology it needs from the second controller. Any controller would have enough information to generate SHARP paths between any two nodes running SHARP. In a sense, the controllers would form a distributed system, since each one holds part of the larger network topology and shares relevant information when necessary.

Another way to go about routing SHARP packets over the wider internet would be to mimic the interdomain/intradomain approach adopted by the BGP. Controllers would be in charge of knowing how to route packets within their own local networks, similar to an intradomain protocol. Controllers would also be aware of the routers in separate networks to which the controller would send SHARP packets bound for those networks, similar to an interdomain protocol. When those edge routers receive packets from other SHARP networks, they could ferry the packets to local controllers to receive routing rules for the local network. This would allow a controller to send packets to SHARP nodes on the wider network without communicating with the controller associated with the remote SHARP node. While such systems are relevant to SHARP, they are beyond the main scope of this research project. A future MQP could focus on solving SHARP's distributed controller problem.

5.2 Securing the SHARP Protocol

The SHARP protocol accomplishes our main objective: it offers host-based routing at both Layer 2 and Layer 3. Unfortunately, the inherent security of the protocol was not a consideration in its design. As a result, the SHARP protocol in its current form has inherent security issues which inhibit its ability to be adopted in a practical environment. A cursory analysis of the protocol indicate that man-in-the-middle (MITM) attacks are feasible both between a SHARP node and the controller and between two SHARP nodes in a SHARP path. Since all nodes in a SHARP path share the burden of directing packets along that path, a malicious node which intercepts SHARP packets could arbitrarily redirect those packets. Clearly, the channels of communication amongst

the SHARP nodes and the controller must be secure for SHARP to realistically ever see widespread adoption.

Fortunately, an entire class of cryptographic systems exist which expressly solve the problem of facilitating communication between two parties over an insecure network. These systems typically rely on a trusted authority (TA) which both parties trust to facilitate communication. Our controller already interacts with each node to dictate routing rules, so it could easily be repurposed to also serve as the TA. For the remainder of this section, the Needham-Schroeder protocol will be examined as a potential system which could be implemented into SHARP; however, it should be noted that other protocols exist, including Kerberos, which are suitable for SHARP's needs. Under the Needham-Schroeder protocol, a SHARP node which would like to forward a SHARP packet would first ping the controller and request a session key for communication between itself and the next node in the path. The node receives a copy of the session key, encrypted with a shared key between itself and the controller; it also receives a copy of the session key encrypted with a shared key between the next node and the controller. The first node would send the latter encrypted session key to its desired node, which would decrypt it to retrieve its copy of the session key. Since each session key was encrypted with a trusted key between the controller and a unique node, both parties can be confident that they were issued a fresh and trustworthy session key. Thus, they can proceed to send nonce-padded messages to each other, encrypted with the session key. It should be noted that the Needham-Schroeder protocol does not fully protect against replay attacks or MITM attacks; thus, Kerberos may be a more suitable protocol to implement into SHARP.

One outstanding issue of using a key transport protocol is that somehow, a trusted key must be established between the controller and each node. Furthermore, the channel between the controller and any given node must also be resilient to MITM attacks and replay attacks. Unfortunately, the protocols listed above do not explicitly describe how trusted keys are issued to communicating parties. In the case of SHARP, that responsibility could be handled manually by a network administrator, or some other system could automatically register legitimate SHARP nodes in the network. The design and implementation of such a protocol into SHARP could be the foundation of another future MQP.

5.3 Analysis of Kernel Module Failures

Our experiment results conclusively demonstrate SHARP's ability to route packets between different host devices. Thus, SHARP packets can be generated and passed between nodes; however, the packets fail to get absorbed back to the application layer once they have reached their final destination. This stems from errors in our inbound network callout, which is responsible for the destination-handling of packets. Unfortunately, the time restrictions of the MQP forced us to prioritize the generation of conclusive results which proved our VLAN/SHARP routing capability, so we were unable to fix the outstanding issues in the inbound network callout. However, we have traced the sources of the errors; they are outlined below.

For cases where a SHARP node is the true destination of a SHARP packet, the inbound callout successfully unwraps and re-injects the packet's payload. The re-injected packet is visible in the kernel queue, and permitted past the inbound network callout. However, the re-injected payload never reaches the application layer. This would suggest that somehow, the payload itself is malformed, and the Windows Filtering Platform is tossing it out. We know that since the payload is never processed by the network interface card, the payload's IP and TCP/UDP checksums are never calculated. We attempt to remedy this by manually calculating both checksums before the payload gets bundled in the outbound network callout. We know for certain that the IP checksum is correct, since we employ it to checksum VLAN-tagged packets. Unfortunately, the TCP/UDP checksum is more complex to calculate, and even harder to validate. We suspect that the TCP checksum is being calculated incorrectly. This concurs with our probing of the WFP, which indicated that packets were getting tossed *before* the TCP/UDP callout. If we had more time to complete this MQP, we would

develop a system where we could replay specific outbound packets through an NIC to ensure that our checksum algorithm is correct.

Unfortunately, the proxying system which allows SHARP packets to arrive at non-SHARP nodes is fundamentally broken. This is because the inbound network callout does not have access to the outbound network queue, despite the fact that the outbound queue re-injection functions are exposed to the inbound callouts. Thus, when the inbound network callout tries to inject a proxy packet into the outbound network queue, the packet instead enters the *inbound* network queue and immediately gets tossed. This means that our proxying code is fundamentally untestable. In this case, limited documentation and code samples led to a fundamental misunderstanding about the Windows Filtering Platform. A redesign of the system, potentially using a user-space service, might be necessary to successfully implement communication to non-SHARP nodes.

5.4 Reflections on the Windows Kernel

Windows driver development comes with unique challenges, from the implementation process itself to the testing and validation pipeline. Those added challenges arise from several sources, including the proprietary nature of the kernel, the design of Windows itself, and the documentation on the Windows filtering platform. In this section of the discussion, we catalogue several the difficulties we dealt with while developing for the Windows kernel.

Windows documentation itself is found on Microsoft's website; it mostly consists of Doxygen-style descriptions of different API calls under the WFP, but does not explain how the WFP is leveraged to perform essential tasks. Furthermore, we found that the WFP often offers several different API calls accomplish the same task. One of the reasons why our inbound network callout was designed incorrectly was because the WFP documentation never clarifies at what callout layer each re-injection function should be used. We also discovered WFP behaviors that were inconsistent with Microsoft's documentation. For example, their Github documentation states that callouts registered at the outbound Ethernet layer intercept packets at their IP header; in reality, those packets are intercepted at the Ethernet header [24]. Collectively, the challenges we encountered while learning about the WFP reduced the speed at which we could develop SHARP.

The only resource available which fully implements callouts for the WFP exists as a Git repository published by Microsoft, appropriately titled the *WFP sampler*. While in-line comments do exist throughout the sampler, which clarify the functionality of code snippets, the code itself does not document the context surrounding how different calls are used. For example, some API calls need specific global variables asserted to function properly; the sampler might achieve this purpose with a `goto` statement which sets several global variables at once, of which only a couple are relevant. The sampler code also has limited relevance to our application because the WFP offers several different methods of accomplishing the same tasks. Often, the sampler would use a valid method or technique which would be irrelevant for our application because we had chosen a different method provided by the WFP to accomplish the same task.

We also incurred delays in our development from our kernel debugging environment. Our testing environment uses several Windows virtual machines. Some machines are reserved for developing SHARP, while others are dedicated to testing SHARP. Thus, to test SHARP, our code had to first be compiled on a development machine, then transferred over to a testing machine. We accomplished this with Git; however, this added inefficiency to our testing pipeline because all object files and binary files had to be pushed and pulled remotely. Once the compiled binaries arrive at a test machine, a script installs our kernel modules. If the code fails, (as it almost invariably will during development), Windows crashes, producing a blue screen and rebooting. This leaves several minutes where the machine is unusable and unrecoverable. The machine will try to reboot and run the broken kernel modules several times before it gives you the option to boot Windows in safe mode to uninstall them. There is no way to determine what went wrong during that time. Once the machine is recovered, the only way to examine the crash is to pass a crash report produced by

Windows to *WinDbg*, a debugging platform which produces register dumps, stack traces, etc. from that crash dump. Unfortunately, these crash dumps often did not indicate the true source of a crash. The Windows kernel is multi-threaded; if the kernel modules do something which causes a separate thread to crash, the crash dump will provide a stack trace to that unrelated thread. Furthermore, Windows overwrites its crash reports every time it blue-screens; since a machine will blue-screen several times in a row before becoming recoverable, only the final (and often irrelevant) blue-screen is available for analysis. This left us with print debugging and code deletion as our primary debugging techniques.

5.5 Future Development Steps

While this report marks the end of our major qualifying project, we suspect that we will be continuing to develop and update SHARP as a protocol. This section describes the steps which we plan to take next as we continue to develop SHARP.

Our immediate goal is to repair our broken inbound network callout. Most importantly, we would like SHARP payloads to be retrievable the application layers of SHARP nodes. As discussed in Section 5.3, we suspect that a miscalculation of the TCP/UDP checksum is causing the WFP to toss payloads as they exit the inbound network callout. Thus, we will probably develop a method to easily test the validity of our checksums, perhaps by replaying the same packets several times with and without a network interface card. The proxy system for non-SHARP destinations in the inbound network callout needs a more general rework. We will write a new callout which encapsulates the proxy service in its entirety. To circumvent the fact that inbound callouts do not have access to the outbound queue, we will probably use another userspace helper to reroute packets out of the machines.

Once we fully implement SHARP's basic functionality, we will have more freedom in how we want to develop our system. One possibility could be to implement the security features mentioned in Section 5.2, to help make SHARP a system which could be used in practice. We could also integrate SHARP into a software-defined network, such as PEACE. To test the SHARP protocol, we were caching manually-generated rules on SHARP nodes. We have yet to implement controller interaction or rule caching with SHARP. This would also be an essential step towards making SHARP usable in real-world settings. On top of this, we could add network resiliency protocols and advanced performance metrics to best gauge the performance hit incurred by using SHARP. As a general rule, we would like to further show that SHARP could be used in practical applications.

6 Related Work

Here we describe early implementations of host-based SDNs in the literature. We also examine works that employ the SDN paradigm to improve network security, works that address the problem of flexible TCP connections, and different implementations and applications of overlay networks.

6.1 Host-based SDN

As we mentioned in Section 1, the scientific community has only marginally explored the potential of a host-based SDN design. In *Scotch*, Wang *et al.* dynamically incorporate hosts running Open vSwitch into a switch-based SDN in order to take advantage of the much higher processing power they possess [2]. Papers by Taylor *et al.* [27] and Shue *et al.* [11] mark some of the first host-based SDN designs, and suggest that hosts can provide the SDN controller with contextual information about each tentative connection to allow for more fine-grained rules. Our work builds directly off of these papers, which allow a controller to grant or deny connection requests, but lack the traffic path control functionality that are natural for switch-based SDNs.

An implementation of host-based Layer 2 routing control does exist in the current literature. With the SPAIN protocol, Mudigonda *et al.* create VLAN spanning trees, and allow endpoints to manipulate traffic pathing by connecting them to trunk ports and inserting VLAN tags into outbound traffic [28]. This is very similar to the final design we chose for SHARP, however SPAIN does not support endpoints delegating routing decisions to an external controller as an SDN would. Additionally, SHARP includes a protocol header which allows for more complicated network routes to be defined, which have intermediate nodes and can travel across multiple VLAN IDs.

6.2 SDN-Based Security

The security applications of SDNs has been readily explored by the community. We designed SHARP in part to give IT technicians more freedom in the placement of IDS middleware within their networks; to achieve the same goal with a traditional switch-based design, Qazi *et al.* developed SIMPLE [29]. SIMPLE is a policy enforcement program that updates flow rule tables in OpenFlow-enabled switches to steer traffic through a logical ordering of middleboxes provided by network administrators. To aid developers in the creation of controller-based security software, Shin *et al.* created the FRESCO framework [30]. Nobakht *et al.* develop a module for the popular Floodlight SDN controller to perform specialized intrusion detection and prevention for IoT devices [31]. Once an IoT device is registered with the system, the SDN redirects all of its traffic through the controller for monitoring. Othman *et al.* demonstrate a controller-based firewall application for traditional switch-based SDNs [32], while the PEACE host-based SDN firewall demonstrates that host-provided connection context can be used to create more powerful firewall rules [10].

6.3 Flexible TCP Connections

In most cases, the design of the SHARP system allows for a connection's assigned SHARP route to be modified during the connection. Ignoring the trivial case of the source and destination node, there is only one situation in which a node must remain static. When the *DestinationFlag* is not asserted, the second to last node in the chain proxies the packet stored in the SHARP header by overwriting the inner packet's source IP address. If the SHARP route changes and a new proxying machine is specified, the destination will receive packets for the same connection from two different IP addresses, and if the inner packet's transport layer protocol is TCP, the connection will be reset. To allow the proxying node for a connection to be changed while the connection is ongoing would require that the destination can receive packets for the same TCP connection from multiple IP addresses.

Some methods have already been developed for allowing multiple IP addresses to share a single TCP connection. Migratory TCP (M_TCP) is an augmentation of the existing TCP protocol that allows the server to specify multiple IP addresses to a client [33]. The client can then connect to one of the provided IP addresses and specify when it wishes to communicate with a different server IP address at any point in the connection using a `MIGRATE_REQUEST`. Multiple systems have been proposed for migrating application state between two clients in addition to migrating the TCP connection itself [34, 35].

Two other protocols which could solve the issue of a single connection being shared by two client IP addresses are Multipath TCP (MPTCP) [36] and SCTP [37]. MPTCP has an implementation for the Linux kernel and is employed in Apple's iOS and OS X operating systems, however it is not supported by the Windows kernel [38, 39]. Likewise, SCTP is not a heavily adopted protocol and is not implemented in the Windows kernel, and thus these options were not available to us for designing SHARP. It should be noted that there is a third-party Windows driver for SCTP called SctpDrv [40], however in order to use the driver for a connection, the destination of the connection would have to support SCTP as well. Unless the destination were also a SHARP node, we would not be able to assume that the destination had SCTP support.

Finally, the most widely-adopted but least desirable solution to this problem is using Network Address Translation (NAT). NAT allows for IP addresses and transport layer ports for incoming or outgoing packets to be modified according to a set of rules. To apply NAT to solve the multiple-source TCP connection problem, when a new proxying node N is specified the SDN controller could interact with the SHARP client on the destination machine and N to map N's IP address to the that of the original proxying node, and vice versa. Since our implementation of SHARP did not include controller interaction, we did not include this feature.

6.4 Overlay Networks

Overlay networks are applicable to a wide range of applications. Many overlay network applications, including Andersen *et al.*'s resilient overlay networks (RONs), involve dynamically selecting the highest-quality packet routes between hosts [19]. In the RON implementation, a special header is appended onto each packet dictating the heuristic for route quality, like throughput, latency, or packet loss. Each node on the network parses the heuristic, or policy, in the header and generates a routing table which directs the next hop for the packet. Gu *et al.* also use overlay networks to guarantee quality of service between nodes [41]. Stone uses overlay networks to dynamically reroute suspicious traffic from edge routers to special tracking routers for inspection. Each edge router, as an overlay node, checks if incoming traffic matches a certain attack signature. If it does, the overlay network uses BGP to update the routing path of the traffic to the tracking router [42]. Other work has been done to build application-agnostic overlay networks. Li and Mohapatra propose using one or more overlay broker (OB) nodes within an autonomous network to implement arbitrary overlay services. The resulting "overlay service" would implicitly guarantee services like network topography discovery, routing path selection, and fault tolerance [43].

7 Conclusion

Host-based SDN has the potential to be an inexpensive, flexible, and powerful alternative to switch-based SDN for enabling routing control and improving security in the LAN. So far, the concept of a host-based SDN with routing control has not been explored by the scientific community, partly because of the conception that it could not offer the same fine-grained traffic engineering capabilities as switch-based SDNs. The goal of our project was to design a host-based SDN that achieved feature parity with switch-based SDNs, and do so using only legacy hardware. Our final design is called SHARP, which we implemented as a user-space daemon and a set of kernel-level callouts for the Windows Filtering Platform. To accomplish routing control at the Ethernet layer, we repurposed VLANs to create a set of unique spanning trees defining routes between nodes on a network. We also attached endpoints to trunk ports so that they could manually insert VLAN tags into outgoing packets to decide which VLAN each packet was a part of, and thus which path the packet would take to reach its destination.

We also introduced Layer 3 routing control to the SHARP system, which local switch-based SDNs do not support. To do this we developed the SHARP header, which encapsulates outgoing packets and defines the set of SHARP-enabled machines the packet must visit before reaching its destination. The SHARP header also defines which VLAN IDs to tag the packet with as it travels between each node in the chain. By sacrificing Layer 2 control in some parts of the SHARP path, our system even allows for public IP addresses on the Internet to be specified as intermediate nodes.

To test our design, we built an isolated physical network out of switches, hubs, and laptops. We were able to prove that, with a correct configuration of VLANs on the switches, we could choose between two paths to reach an intermediate node in a SHARP path simply by changing the ID of the VLAN tag inserted in an outgoing packet's Ethernet frame. We also showed that our implementation correctly parsed the path defined in a SHARP header and could reroute packets to intermediate nodes before successfully delivering them to the original destination machine. With SHARP, we have demonstrated that a host-based SDN can offer routing control equal to and greater than that of a switch-based SDN, and in a manner that is neither costly nor restrictive.

References

[1] H. Abrahamsson and M. Bjorkman, "Robust traffic engineering using l-balanced weight-settings in OSPF/IS-IS," in *2009 Sixth International Conference on Broadband Communications, Networks, and Systems*, pp. 1–8, Sep. 2009.

[2] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen, "Scotch: Elastically scaling up SDN control-plane using vSwitch based overlay," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pp. 403–414, ACM, 2014.

[3] "Understanding the SDN architecture – SDN control plane & SDN data plane." `https://www.sdxcentral.com/sdn/definitions/inside-sdn-architecture/`. Accessed December 28, 2018.

[4] "What is SDN?." `https://www.juniper.net/us/en/solutions/sdn/what-is-sdn/`. Accessed October 8, 2018.

[5] "Sofware-defined networking." `https://www.cisco.com/c/en/us/solutions/software-defined-networking/overview.html`. Accessed October 8, 2018.

[6] "Floodlight REST API." `https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343539/Floodlight+REST+API`. Accessed December 28, 2018.

[7] "What is openflow? definition and how it relates to SDN." `https://www.sdxcentral.com/sdn/definitions/what-is-openflow/`. Accessed October 8, 2018.

[8] "Openflow." `https://whatis.techtarget.com/definition/OpenFlow`. Accessed October 8, 2018.

[9] "Three different SDN models." `https://www.rcrwireless.com/20170811/three-different-sdn-models-tag27-tag99`. Accessed October 8, 2018.

[10] "Contexsure networks." `https://www.contexsure.com/`. Accessed October 2, 2018.

[11] M. E. Najd and C. A. Shue, "Deepcontext: An openflow-compatible, host-based SDN for enterprise networks," in *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, pp. 112–119, Oct 2017.

[12] "OVS-on-Hyper-V design." `http://docs.openvswitch.org/en/latest/topics/windows/`. Accessed January 3, 2019.

[13] "Windows filtering platform." `https://docs.microsoft.com/en-us/windows/desktop/fwp/windows-filtering-platform-start-page`. Accessed January 3, 2019.

[14] "Packet injection functions." `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/packet-injection-functions`. Accessed January 3, 2019.

[15] "ALE layers." `https://docs.microsoft.com/en-us/windows/desktop/fwp/ale-layers`. Accessed January 8, 2019.

[16] "Windows driver framework." `https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/_wdf/`. Accessed January 10, 2019.

[17] "What is encapsulation?." `https://www.computerhope.com/jargon/e/encapsul.htm`. Accessed January 13, 2019.

[18] C. Shue, Y. Shin, M. Gupta, and J. Y. Choi, "Analysis of IPSec overheads for VPN servers," in *Secure Network Protocols, 2005.(NPSec). 1st IEEE ICNP Workshop on*, pp. 25–30, IEEE, 2005.

[19] D. G. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris, *Resilient overlay networks*. PhD thesis, Massachusetts Institute of Technology, 2001.

[20] S. McQuerry, "CCNA self-study (ICND exam): Extending switched networks with virtual LANs," Jan 2016.

[21] "Understanding and configuring spanning tree protocol (STP) on catalyst switches." `https://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/5234-5.html#concepts`. Accessed Februrary 8, 2019.

[22] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[23] "Using layer 2 filtering." `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/using-layer-2-filtering`. Accessed March 12, 2019.

[24] D. McMichael, "Offset of fwp_layer_inbound_mac_frame_ethernet · issue #529 · microsoftdocs/windows-driver-docs."

[25] "Windivert documentation." `https://github.com/basil00/Divert/wiki/WinDivert-Documentation`. Accessed Feburary 8, 2019.

[26] "GSN3." `https://www.gns3.com/`. Accessed February 13, 2019.

[27] C. R. Taylor, D. C. MacFarland, D. R. Smestad, and C. A. Shue, "Contextual, flow-based access control with scalable host-based SDN techniques," in *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pp. 1–9, IEEE, 2016.

[28] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul, "SPAIN: COTS data-center ethernet for multipathing over arbitrary topologies.," in *NSDI*, vol. 10, pp. 18–33, 2010.

[29] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," *SIGCOMM Comput. Commun. Rev.*, vol. 43, pp. 27–38, Aug. 2013.

[30] S. W. Shin, P. Porras, V. Yegneswara, M. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks," in *20th Annual Network & Distributed System Security Symposium*, NDSS, 2013.

[31] M. Nobakht, V. Sivaraman, and R. Boreli, "A host-based intrusion detection and mitigation framework for smart home IoT using openflow," in *Availability, Reliability and Security (ARES), 2016 11th International Conference on*, pp. 147–156, IEEE, 2016.

[32] W. M. Othman, H. Chen, A. Al-Moalmi, and A. N. Hadi, "Implementation and performance analysis of SDN firewall on POX controller," in *Communication Software and Networks (ICCSN), 2017 IEEE 9th International Conference on*, pp. 1461–1466, IEEE, 2017.

[33] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode, "Migratory TCP: Connection migration for service continuity in the internet," in *null*, p. 469, IEEE, 2002.

[34] M. Bernaschi, F. Casadei, and P. Tassotti, "Sockmi: a solution for migrating TCP/IP connections," in *Parallel, Distributed and Network-Based Processing, 2007. PDP'07. 15th EUROMICRO International Conference on*, pp. 221–228, IEEE, 2007.

[35] M. Barisch, J. Kögel, and S. Meier, "A flexible framework for complete session mobility and its implementation," in *Meeting of the European Network of Universities and Companies in Information and Communication Engineering*, pp. 188–198, Springer, 2009.

[36] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP extensions for multipath operation with multiple addresses," tech. rep., 2013.

[37] R. Stewart, "Stream control transmission protocol," tech. rep., 2007.

[38] "Multipath TCP - linux kernel implementation." `https://multipath-tcp.org/pmwiki.php`. Accessed October 9, 2018.

[39] "Apple opens multipath TCP in iOS11." `https://www.tessares.net/highlights-from-advances-in-networking-part-1/`. Accessed October 9, 2018.

[40] "Sctpdrv: an SCTP driver for microsoft windows." `https://web.archive.org/web/20110108073234/http://www.bluestop.org/SctpDrv/`. Accessed October 9, 2018.

[41] X. Gu, K. Nahrstedt, R. N. Chang, and C. Ward, "QoS-assured service composition in managed service overlay networks," in *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pp. 194–201, IEEE, 2003.

[42] R. Stone *et al.*, "Centertrack: An IP overlay network for tracking DoS floods.," in *USENIX Security Symposium*, vol. 21, p. 114, 2000.

[43] Z. Li and P. Mohapatra, "QRON: QoS-aware routing in overlay networks," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 29–40, 2004.

Appendix A Router to Switch Configuration

As we discussed in Section 4, the physical testbed on which we demonstrated our implementation required four VLAN-enabled switches. The WPI CS Department had a large supply of available TP-LINK Archer C7 routers, and so we opted to use those to implement our testbed. Here we discuss the process of converting the routers into switches by installing the open-source firmware OpenWRT. We then explain how to create VLANs and configure each of the router's LAN ports to act as either an access or trunk port. Finally, we explain how to install the `tcpdump` packet capture tool on the routers.

A.1 Installing OpenWRT

The TP-LINK firmware that the routers shipped with did not support VLANs so we installed a distribution of OpenWRT, a free Linux-based operating system for embedded devices. OpenWRT offers many features, including support for VLANs, that vendors only offer in the firmware of their high-end devices.

The TP-LINK v3.13 firmware for the Archer C7 has a straightforward interface for firmware upgrades. However, our initial attempt to install the OpenWRT firmware was rejected by the stock firmware because the signature was not recognized. After some time testing different methods, we discovered a successful (albeit convoluted) way to install OpenWRT:

1. Download DD-WRT, OpenWRT, and TP-Links v2 firmware files. The DD-WRT and TP-Link files can be found at the DD-WRT router database, or the following link: `https://dd-wrt.com/support/router-database/?model=Archer%20C7%20(AC1750)_2.x`. The OpenWRT firmware file can similarly be found on OpenWRT's website at: `https://openwrt.org/toh/views/toh_fwdownload`.

2. Connect a laptop to the router on LAN Port 1.

3. In a browser type in `tplinkwifi.net` in the address bar and log in using the default credentials of user name and password as 'admin'.

4. Click "System Tools", and then click "Firmware Upgrade".

5. Click "Browse" and select the DD-WRT binary file downloaded in Step 1. After clicking "Upgrade", the router will take a few minutes to switch to using the DD-WRT firmware, and the web page will become unresponsive.

6. Connect to 192.168.1.1 via a browser and set a user name and password when prompted. A web page (shown in Figure –) will appear.

7. Click on the tab labelled "Administration", submit credentials if prompted, and click on the "Firmware Upgrade" tab.

8. Select the Archer C7 v2 binary file from Step 1 and click "Upgrade". As in Step 5, after a few minutes the webpage will become unresponsive.

9. Connect to 192.168.0.1 via a browser. The TP-LINK router web interface will once again appear, except the firmware has now been downgraded to v3.13.

10. Navigate to the "Firmware Upgrade" tab and now select the OpenWRT binary file from Step 1 and upgrade. As before, the webpage will eventually become unresponsive.

11. Open a terminal program and run `ssh root@192.168.1.1`. After logging in, the interface will appear as shown in Figure 28.

Figure 28: The initial OpenWRT shell opened with `ssh`.

12. Set a password for the root account using the `passwd` command.

13. Return to the browser and navigate to 192.168.1.1. Use the credentials set in Step 12 to and log in. The OpenWRT web interface will now appear.

A.1.1 Installing `tcpdump`

The `tcpdump` program is a tool for recording the packet data of traffic entering and leaving a machine. To install `tcpdump` on our routers, we used OpenWRT's package manager, the Open Package Management (`opkg`) tool. Instead of registering the routers' MAC addresses with WPI's network, we opted to connect them to the Internet via a mobile hot spot for the purpose of downloading `tcpdump`. Our method for downloading `tcpdump` is as follows:

1. Connect a computer to the router on a LAN port.

2. Access the router's web console by putting its IP address in a browser's search bar.

3. Click on the "Network" tab, and then click the "Wireless" Tab. The webpage should appear as shown in Figure 29.

4. Click the "Scan" button on the `radio1` interface, which will redirect to a page displaying 2.4GHz Wi-Fi networks. Select the Wi-Fi network to be used for downloading the software..

5. Select the correct mode under the "Operating Frequency" setting. For our 2.4Ghz bandwidth mobile hot spot, the "legacy" mode was correct.

6. Once the router is connected to the Internet, open a terminal program and run `ssh root@<router IP address>`.

7. Run `opkg update` and then `opkg install tcpdump`.

A.1.2 Configuring VLANs

To configure VLANs for an OpenWRT router, first connect to the router and open its web interface. Click the "Network" tab, and then the "Switch" tab to arrive at the webpage shown in Figure 30.

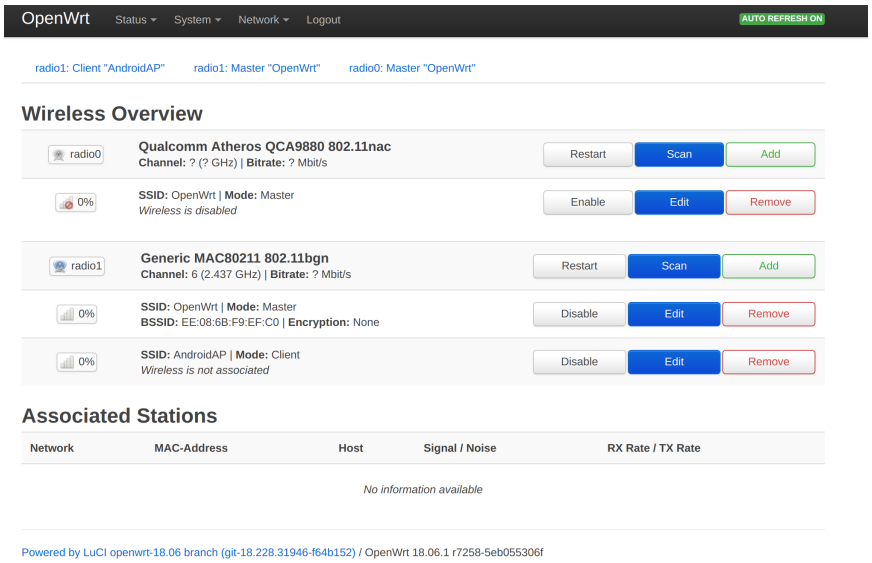


Figure 29: The "Wireless" tab of the OpenWRT web interface.

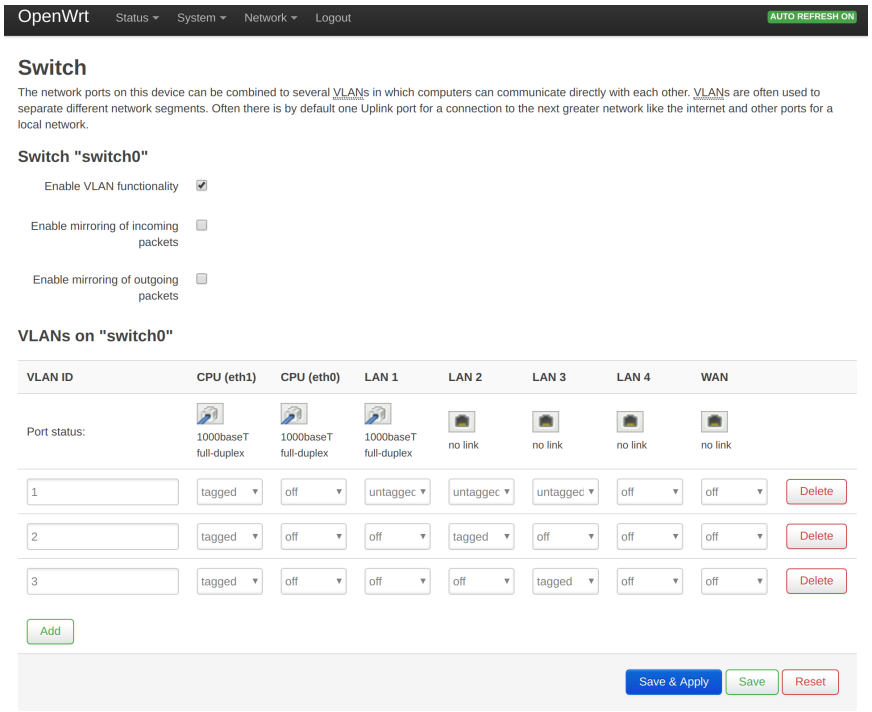


Figure 30: The "Switch" tab of the OpenWRT web interface is used to configure VLANs on various ports and interfaces of the router.

In this screen we can create VLANs and assign them to ports. To create a new VLAN, click the "Add" button in the bottom left corner, and set the ID to be a unique number between 1 and 4096. Each row corresponds to one VLAN, while each column corresponds to a given port or interface. To configure a port as an access port for a given VLAN, set that port to be "untagged" and ensure that no other entries in the column are also set to "untagged." The router does not allow more than one VLAN to be set to "untagged" for a given port, otherwise it would be unable to determine which untagged packets belong to which VLAN. To set a VLAN to be trunked on a given port, set it to be "tagged." Any number of VLANs can be trunked on a given port since trunk ports expect incoming frames to include VLAN tags that indicate which VLAN they belong to.