Inferring Network Infrastructure and Session Information through Network Analysis

By

BRIAN PERRY
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Matt Bishop, Chair

Dipak Ghosal

Karl Levitt

Committee in Charge

2022

This is dedicated to my father for all of the guidance you have provided me through my life, for consistently helping me achieve my goals, the meaning of hard work and the unyielding resolve necessary to succeed. And my late grandfather, you are my greatest supporter and a strong pillar of the family; and with your aspiration you lead me here. Both of you will always be the biggest influence in my life.

I also want to thank all of my instructors especially Dr. Matthew Bishop for being a tremendous advocate throughout my entire time in the program, department members, coworkers, and friends for all of your time and assistance to complete this dissertation.

Contents

List of Figures

List of Tables

1 Abstract

An adversary can be a resourceful entity that can obtain information (i.g. infrastructure design) that is intentionally hidden in order to prevent a vast array of exploitations from occurring. Similarly, an adversary can be seen as a monitoring entity preventing the users from freely accessing the Internet via the network's design. This leads us to our questions in two fold. Does a reconnaissance methodology exists to infer the load balancing algorithm operating within a network's infrastructure, and can network sessions be identified by an adversary (monitoring entity) with and without protocol obfuscation?

Software-Defined Networking (SDN) was used to construct a network with a load balancing algorithm, because it separates the control plane and the data plane. The control plane performs all of the routing decisions for the network through the use of a special device called a controller. This controller is a very critical aspect of the network so we decided to build an SDN network with multiple controllers with a load balancing algorithm to distribute the processing load. We performed a controller side-channel attack to analyze a packet's response time in order to 1) determine the number of controllers operating within an SDN network, 2) inherently determine the load balancing algorithm utilized, and 3) identify the OS scheduler as the environment's bottleneck.

We decided to expand the ability to mask information from an adversary or monitoring entity by studying protocol obfuscation. We focused our efforts on protocol obfuscation over encryption because it can be difficult for a monitoring entity to determine the utilization of protocol obfuscation over a connection's utilization of encryption. To be precise, we obfuscated the SSH protocol through the exchange of PDFs. We examined how a monitoring entity (aka an adversary) can classify a SSH session with and without PDF obfuscation based on the characteristics of the protocol. Throughout our experiments we noticed some inter-packet delay characteristics that constantly appeared within the environments, but the packet size characteristics proved to be a reliable metric for analysis because of the similar patterns seen throughout the environments and network sessions. We argue an obfuscation program has to hide the target data and the target protocol's characteristics so that a user can completely and holistically hide their network session from a monitoring entity.

2 Introduction

An adversary can be a resourceful entity that can obtain information (i.g. infrastructure design) that is intentionally hidden in order to prevent a vast array of exploitations from occurring. Similarly, an adversary can be seen when a monitoring entity constantly scans environments to obtain information about the content the users are reviewing, so that they can prevent the user from benefiting from all the Internet has to offer. Can an adversary perform a reconnaissance methodology in order to infer the load balancing algorithms within a network's infrastructure. This was extended to determine whether an adversary (monitoring entity) can identify a network session with and without protocol obfuscation.

Software-Defined Networking (SDN) is an attractive network architecture for organizations that require a dynamic network with high bandwidth. It separates the control plane and the data plane where the control plane contains a special device called a controller. The controller is responsible for managing all of its switch's forwarding planes; as a result, network devices, such as switches, can be seen as unintelligent. These are the reasons why we decided to use SDN as the network infrastructure that an adversary will attack. Additionally, the nature of an SDN network causes the controller to become a single point of failure. If the controller were to go down, the whole network could shut down because the networking devices will not be able to forward packets with non-existent or expired forwarding rules.

Services rely upon for uninterrupted access become a single point of failure within an infrastructure. In order to increase the service's resilience towards unexpected system crashes or malicious attacks, it is necessary for the administrator to install at least one failover server. The failover server(s) will allow for continued service when these unexpected system downtimes occur. As you can expect, since an SDN network's single point of failure is the control plane, this leads to a recommendation for multiple controllers to operate with a single SDN switch.

The goal of this study is to use the controller side-channel attack to determine the number of controllers operating within an SDN network by analyzing the packets' response times. This side-channel attack also inherently determines the load balancing algorithm utilized within the environment, and it identifies the OS scheduler as the bottleneck of the environment. A malicious entity would benefit from obtaining these pieces of information because it will assist them in successfully performing malicious attacks. These malicious attacks include Distributed Denial of Service (DDoS) attacks, mapping the network's topology, and gaining access to restricted systems.

Furthermore, we expanded the ability to mask information from an adversary or monitoring entity by studying protocol obfuscation. If an adversary were able to eavesdrop on a network connection, they could obtain information by analyzing its data or the session's characteristics. A number of obfuscation methods enable users to hide their data. For example, using cryptography and protocol obfuscation techniques hide the traffic's material. Encryption is fairly obvious — it instantly informs an adversary that the data is worth protecting, which leads to the adversary using various algorithms in an attempt to decrypt the data. Protocol obfuscation can be more difficult to detect because it hides the data in plain sight, which can be good or bad, depending upon the depth of the obfuscation procedure.

Tunneling protocols is an effective method of obfuscation. We explore tunneling SSH connections over an exchange of PDFs. We examine how a monitoring entity (aka an adversary) can classify a network session's protocol based on the characteristics of the protocol. We developed an obfuscation application that uses PDFs as the method of obfuscation, which hides both the packet's data and the packet's protocol characteristics. Specifically, the obfuscation application hides the protocol's inter-packet delay (IPD) and packet size characteristics. We investigated the protocol obfuscation in an isolated, local area, and wide area network in order to evaluate its effectiveness.

Throughout our experiments we noticed some inter-packet delay characteristics that constantly appeared within all of the three environments. The fluctuation range (less than 10^1 but greater than 10^{-6}) and command execution can be identified if the inter-packet delay was around 22.5 microseconds. Furthermore, the packet size characteristics proved to be a useful metric for analysis due to similar patterns seen throughout all of the three environments and all of the network sessions. If a monitoring entity cross-references both the inter-packet delay and packet size characteristics, they can successfully identify command execution as well as the type of command executed within a wide area network; primarily because of the reliability of the packet size characteristics. Our results indicate an obfuscation program will need to mask the target protocol's characteristics in order to completely and holistically hide its network session.

2.1 The Impact of This Problem

In this section, we discuss our motivation for this work.

2.1.1 SDN

In order to increase network resilience, Abdelaziz et al [1] pointed out that multiple controllers must run in an Software-Defined Network (SDN). This leads to the question of how many controllers should be within an software-defined network, and where should the controllers be placed. Heller et al [22] called this problem the "controller placement problem". By comparing the latency of their experimental networks with the latency from Internet Topology Zoo [33] topologies, Heller et al conclude that the number of controllers, as well as where to deploy the controllers, depends on the network administration requirements, choices of metric, and network topology. If an adversary can reverse engineer load balancing algorithms, then they could possibly utilize the same method for various different scenarios. For example, they could use the methodology to identify the number of controllers active in an software-defined network, the number of servers running in a cluster, whether a routing protocol adjusts routing based on the load of a link, and the number of processors within a CPU. This methodology is versatile enough to use timing analysis to discover any and all entities that are providing the same, or similar, service.

If an adversary can learn information about the infrastructure, then they can use that information to disrupt services. For example, the central management nature of software-defined networking may be used to identify the number of controllers active in an software-defined network because all new forwarding requests have to go to the controllers. As a result, an adversary can inject specially crafted packets that force the switch to send a forwarding request to the controller. If the adversary were to flood the switch with these specially crafted packets, then the controller would become congested, thus increasing the packet's response time. In a multiple controller software-defined network there will be noticeable response time patterns that are based on the load balancing algorithm's use of a failover controller. These response time patterns indicate to the adversary that a failover controller exists within the network. We will have more details on detecting a failover controller in a multiple controller software-defined network within the remaining sections of this dissertation.

Additionally, this methodology could be expanded to mapping a Content Delivery Network (CDN) infrastructure or cluster of servers, determining if the load of a link is a metric for the routing protocol, and determining the number of processors within a CPU. Refer to Section 4.6 for more in depth details.

An adversary can map a cluster of servers by establishing numerous connections to have a channel to all of the servers within the cluster. Some of the connections will have random performance improvement or degradation because it is extremely difficult for servers to constantly have the exact same workload. By keeping track of their connections and the individual connections' performance, these random performance improvements or degradations will will indicate to the adversary the number of servers in the cluster.

An adversary can map a CDN by congesting the closest server with multiple connections, which will force the load balancing algorithm to direct new connections to a server that is farther away from any of the old connections. These new connections will take an increased amount of time to traverse the network than the old connections so they will have a higher response time, which will indicate to an adversary that the new and old connections are established with servers at different locations. This can be repeated until the adversary reaches the last CDN server.

Networks today rely on packet switching that allows a network to forward a packet through a new route the moment it becomes available. An adversary can establish multiple connections to congest sections of the route so that a networking device is forced to redirect some packets through a new route. The packets forwarded through the new route will have a better response time (i.e. lower response time), which will indicate to the adversary that a routing protocol adjusts routing based on the load of a link.

CPU today can have multiple cores so that the processing load can be distribute among the CPU's cores. An adversary can use an infinite loop program and a terminating program to determine the number of processors on the CPU. The adversary can execute an infinite loop with a high priority to take over a core, while at the same time execute the terminating program to see if it in fact terminates. If so, then it indicates another core in within the CPU. This can be repeated until the terminating program no longer terminates, by increasing the number of infinite loops executing in parallel.

2.1.2 Protocol Obfuscation

The motivation behind protocol obfuscation is to allow a user to covertly transmit and receive data, even though an adversary (aka monitoring entity) can be auditing the network for a specific set of data. As indicated previously, a monitoring entity can setup an environment so that they can read encrypted traffic. Since encryption reveals to a monitoring entity that the data contains private information, the entity can attempt to decrypt the data in order to determine what was sent or can use rubber-nose decryption. So, a user's next best option would be to obfuscate the data because the idea of obfuscation is to mask the data that a monitoring would be interested in. Unfortunately, obfuscation also allows an entity to exfiltrate sensitive data out of a restricted server and/or network. Therefore, it is necessary to determine how a network administrator can prevent a malicious entity from stealing confidential information.

By utilizing an existing tunnel that crosses the monitoring entity's scope, the user can communicate with services without providing the monitoring entity another server that they can attack. This means the user can access sensitive or restricted servers (aka services). Furthermore depending on the implementation, these users can establish connections with these servers and/or services without fear of being identified by an adversary. Additionally, these same protocol obfuscation methods can be used with cryptographic protocols.

While protocol obfuscation can be used to hide non-malicious network sessions, it can also be used in conjunction with various types of attacks. For example, it can be used as a covert channel or an attack that extracts and corrupts sensitive data. An entity could use protocol obfuscation inside their covert channel so that an Intrusion Detection System (IDS) cannot review and identify their network session as a covert channel. Using protocol obfuscation within a botnet's command-and-control channel, the commands themselves can be obfuscated, such as informing a bot to remain inactive until a specified time. For example, it could be a numerical value within the payload or it could be the size of payload that contains the sleep command.

2.2 SDN

SDN is a network paradigm for dynamic, manageable networks that require high bandwidth. By separating the network control and forwarding decisions from the data plane, SDN allows for central management through a special device called a *controller*. The controller is responsible for managing the forwarding logic on all assigned SDN switches within the network, and is usually a PC running an application specifically designed as a controller for an SDN switch.

The separation of the data plane and control plane is the core difference between SDN and traditional networks. Unlike traditional networks, SDN does not use routing protocols such as the Spanning Tree Protocol (STP) or Routing Information Protocol (RIP) to generate forwarding rules. As mentioned previously, SDN instead uses a special computer system called a controller that makes the routing decisions for a switch, whenever the switch asks the controller for a forwarding rule. When an incoming packet matches an already installed forwarding rule, the switch will use that rule to forward the packet. If the packet does not match any rule, the switch will ask the controller for an appropriate forwarding rule. A mismatched packet is a packet that does not have a corresponding forwarding rule within the switch (Figure 1), and a matched packet is a packet that corresponds to a forwarding rule within the switch (Figure 2). By having the controller perform all the routing decisions, an SDN network is a central management infrastructure; therefore, the controller is a single point of failure. Due to this vulnerability, there have been multiple studies on how to prevent adversaries from performing Denial of Service (DoS) attacks on the controller [6] [5] [34] [11] [45].

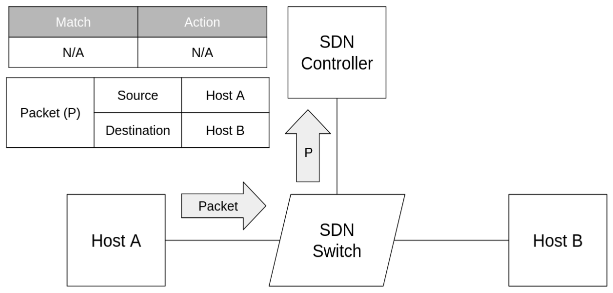


Figure 1: Mismatched Packet

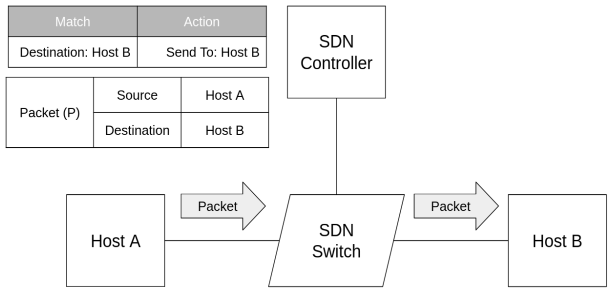


Figure 2: Matched Packet

While SDN is a network paradigm, the Open Networking Foundation's [17] OpenFlow protocol is the most widely used implementation of SDN. Additionally, the OpenFlow protocol's standards state that a switch has one of three options when it comes across a mismatched packet. The switch can either:

- Send a request to a controller asking where to forward the packet;

- Send the packet to a designated default device that is not a controller; or
- Drop the packet.

Implementing the latter two would go against the dynamic nature of SDN, so we can safely assume that an SDN network would most often implement the first option. Also, the OpenFlow protocol's standards state that a controller has one of two options when it receives a request from a switch. The controller can either:

- PACKET_OUT message to tell the switch how to process the packet this one time; or
- Use a Flow Table Modification Message to install a new flow entry onto the switch.

Either way, having the controller process the packet will incur an additional delay when sending the packet to its destination. This additional delay has been utilized in previous works to determine if a network is running an SDN architecture [46], [10] as well as to map the forwarding rules on a switch [48], [47], [36].

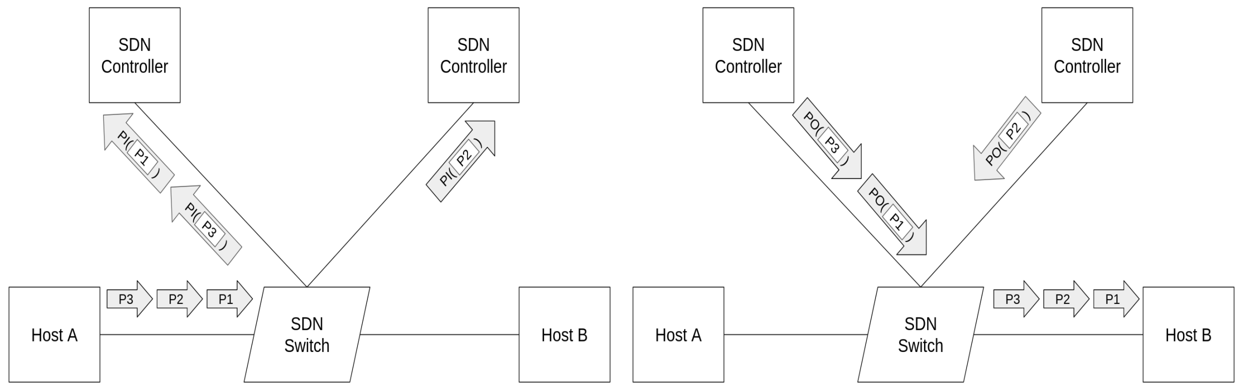
The central management nature of SDN makes the controllers a primary target for attackers because having access to the controller allows an adversary to perform a wide variety of attacks. If an adversary is able to gain access to a controller, then the adversary can install and/or delete forwarding rules on any of its switches; thus, the adversary can reach servers with sensitive information. If for some reason the adversary cannot gain access to the controller itself, then the adversary may be able to distort the controller's view of the network by relocating hosts within the network and/or by establishing new links between switches [25]. It is necessary for an adversary to know how many controllers are within the network, in order for the adversary to successfully perform these attacks. Otherwise, the adversary cannot reliably modify the network because they did not reach the specified controller, or the modifications to one controller may be overridden by another, unmodified controller.

2.2.1 Load Balancing Algorithms

In order to describe the Round Robin and protocol based load balancing algorithm, we will use a multiple controller SDN environment as a reference. The protocol based load balancing algorithm requires at least two controllers.

2.2.1.1 Round Robin Load Balancing Algorithm This load balancing algorithm assigns each controller a numerical value. Additionally, the SDN switches need to support the load balancing algorithm, so they will need to transmit forwarding requests to the next sequentially numbered controller. Since all of the controllers are processing approximately an equal number of forwarding requests, all of the forwarded packets will have approximately the same response time. We want to devise a methodology that can determine the number of controllers operating within the SDN network.

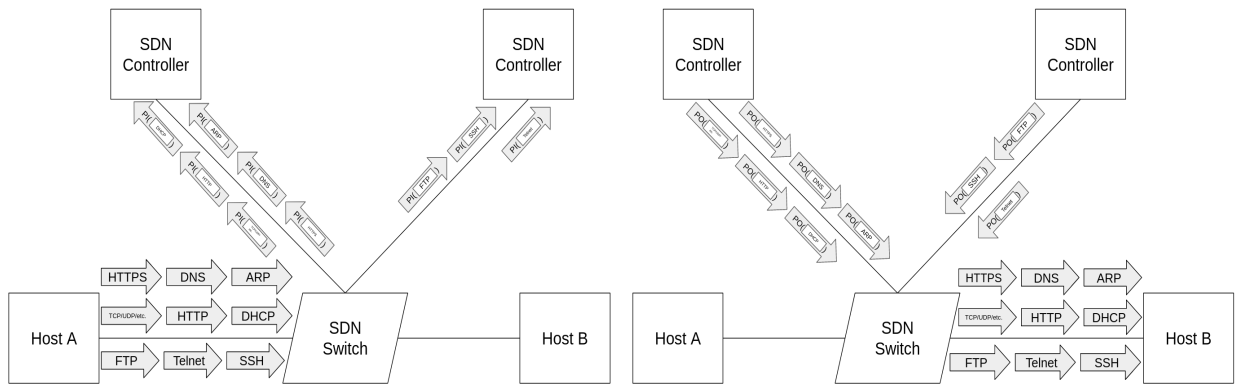
For example, suppose we have two controllers in a multiple controller SDN network. Then, each controller will receive every other forwarding request from each switch. As seen in Figure 3, the forwarding requests are evenly distributed on the control plane, and each controller is able to install forwarding rules without getting congested.



(a) Round-Robin Part 1 (b) Round-Robin Part 2

Figure 3: Round-Robin Load Balancing Algorithm

2.2.1.2 Protocol Based Load Balancing Algorithm This load balancing algorithm assigns each controller a specific protocol or set of protocols. Additionally, the SDN switches need to support this load balancing algorithm, so they will need to transmit forwarding requests to the controller that corresponds to the mismatched packet's protocol. Since each controller's roles are predefined and hard coded, one controller could be processing a large amount of forwarding requests while the other controllers are barely being utilized. We want to determine if a side-channel attack can take advantage of this discrepancy, in order to determine the number of controllers operating within the SDN network.



(a) Protocol Based Part 1 (b) Protocol Based Part 2

Figure 4: Protocol Based Load Balancing Algorithm

For example, consider we have two controllers in a multiple controller SDN network. Then, one controller can be designated as the default controller, while the second controller can generate the forwarding rules for specific protocols. As seen in Figure 4, the network administrator designated the secondary controller to install forwarding rules for the remote access protocols Telnet, SSH, and FTP. Therefore, the default controller is processing the forwarding rules for all other protocols such as the ARP, DHCP, DNS, HTTP/HTTPS, and TCP/UDP protocols.

2.2.2 Problem Statement

For mission critical systems it is important for the system to have backups and redundancy. Unfortunately due to these systems' importance, these systems become a prime target for adversaries. Ideally the backups/redundant systems will prevent an adversary from successfully degrading performance or perform other forms of malicious activity. But an attacker using an Advanced Persistent Threat (APT) will be more thorough in their reconnaissance. An APT attacker will want to determine if a service has any backup servers; to be more precise, the APT attacker will want to determine if backup controllers exist within an SDN network. Moreover, if backup controllers do exist, then the attacker would want to know how many controllers exist and ascertain how failover is initiated. More specifically, the adversary will want to know if the failover is initiated by one of the three following options. Will the backup controllers activate once the main server fails so that only one server is in control at a time? Or will the backup controllers activate once the main server becomes congested so that some of the main server's processing load is alleviated? Or will the backup controllers run in conjunction with the main server, allowing them to be proactive, so that multiple controllers are running at the same time? In other words, the adversary will want to reverse engineer the load balancing algorithm in order to determine the service's backups/redundancy architecture. The main goal of this research is to ascertain whether it is possible for an adversary to determine the number of controllers within an SDN network. This goal also leads to determining the load balancing algorithm that is being utilized and the network's bottleneck. By determining these two subsidiary pieces of information as well, it becomes more important to identify the methodology that can do this.

To get a better understanding of the difference between the failover initiation methods, here is an example of a reactive and a proactive setup. A reactive method is used when the main server takes too long to respond or when the number of connection requests equals some predefined value. This type of load balancing algorithm revolves around the idea that some threshold value, such as response time or number of connections, has to be reached before the failover servers are utilized. A proactive method is when the load is constantly distributed over all of the servers evenly. This type of load balancing algorithm revolves around knowing each servers' load in real time so that all servers are being used and have approximately even performance across all connections. For this type of load balancing algorithm it is easier to think of each server as a bucket, and all buckets are being filled with the same amount of content by the algorithm. A basic example for this type of load balancing algorithm is Round Robin.

The intent of a DDoS attack is to prevent authorized users from accessing a service, DDoS attacks are not necessarily attacks against an individual server. Because of the DDoS attack's intent, the adversary will need to know if the service has any failover servers in order for them to perform a successful DDoS attack. When a service has failover servers, the adversary will have to attack the failover servers as well so that they can successfully accomplish the DDoS attack. When the adversary gains the knowledge of a service's failover servers, this knowledge informs the adversary that they need to increasing the rate of packets they transmit in order to perform a successful DDoS attack.

Furthermore, an adversary would want to know about failover servers because they could use the failover server as their method of attack on a specific target. Sometimes the adversary cannot guarantee that the target will connect to a specific server within the cluster, so the adversary will need to modify all servers providing the service. For example, the adversary may want to upload malware that allows them to eavesdrop on a targeted connection. The eavesdropping malware could be used to obtain metadata, statistical data, or the targeted connection's payload data. For a second example, the adversary may want to use the service to upload some malware to all of the servers because the target views the service as a trusted source. Therefore, the target will be more susceptible to downloading files with zero day attacks. For our last example, the adversary may want the server to have a man-in-the-middle program that reviews and modifies a predefined set of data within the targeted connection. In short, the adversary requires the targeted connection to utilize the server that has been compromised. Consequently, if the adversary only compromises an individual server, then they only have $1/n$ chances of success, where n is the number of servers. Whereas by compromising all n servers, the adversary can guarantee the targeted connection will be attacked.

An SDN network's controllers (aka the servers) could also have a lot of clients (aka the SDN switches). Similarly, the methodology that reverse engineers load balancing algorithms can be very effective for mapping a large corporation's network. The load balancing algorithms used for managing an SDN network's control plane can also be used for managing a corporation's service that has an abundant amount of clients. A corporation who provides a service to a very large set of clients can benefit from a load balancing algorithm. For example, Google uses multiple servers and has all of their clients connected to just one of their many servers. Moreover, Google has each client connect to a server based on the client's region and their load balancing requirements. But what happens when the client's regional servers are congested? Is the client redirected to the next available server closest to the client's region, or sent to some special (aka backup) servers located at Google's main hub? These questions can potentially be answered by utilizing the methodology that reverse engineers load balancing algorithms.

2.3 Protocol Obfuscation

Organizations (and users) have implemented a wide range of security features in order to prevent an unauthorized entity from eavesdropping on their network sessions. In order for users to protect themselves from an unauthorized entity reading their data, various researchers have developed a wide array of methodologies that evade censorship [26]. These methodologies range from hiding the data using CPU intensive cryptographic algorithms to using protocol obfuscation techniques that simply change the data's format.

We decided to study protocol obfuscation because if a user did not want an adversary to instantaneously know that they are transmitting encrypted data, then the user can use protocol obfuscation to hide the encrypted data plus any indication that an encryption protocol is being used. Our inspection encompassed how effective protocol obfuscation can hide the packet's data, as well as the packet's protocol characteristics, from a censoring organization.

There has been notable work within the area of protocol obfuscation. These include an obfuscation application using email servers as the intermediary between two hosts that are covertly communicating with each other [26]. On the other hand, there has been some research that has examined the effectiveness of established obfuscation applications [53]. Actually in the most general sense, protocol obfuscation can be generalized as a form of an overlay network (a tunnel or a peer-to-peer network) because the whole purpose of protocol obfuscation is to use an established network infrastructure, in order to allow two hosts to communicate with each other. This extends the community's research of the area of protocol obfuscation to ingrained protocols like Virtual Private Networks (VPN) [18], The Onion Ring (TOR) [50], and the Invisible Internet Project (I2P) [28].

In this research, we have developed an obfuscation application in order to observe how the network, as well as the host system, behaves when a protocol is being obfuscated. The obfuscation application follows an obfuscation methodology that allows the application to adapt to any transmission and/or reception mechanism. We reviewed a protocol's characteristics with and without operating the obfuscation application to determine how the obfuscation procedure affects the data and its transmission. We implemented specific features within the obfuscation application, so that it can comprehensively hide the protocol. In order to thoroughly determine how this obfuscation application affects its environment, we conducted the same experiments within three different environments.

Our contributions to the field of protocol obfuscation include:

1. Developed an obfuscation application to hide both the packet's data and the packet's protocol characteristics (inter-packet delay and packet size), by inserting the packet's contents within a PDF file before transmitting it onto the network;

2. Inspected the SSH protocol's characteristics specifically, the inter-packet delay and the packet size);
3. Evaluated the SSH protocol's characteristics in order to identify when a command was executed by the user; and
4. Conducted multiple experiments within various environments to perform a comprehensive analysis of the obfuscation application.

2.4 Section Organization

The remainder of this dissertation is broken down in the following sections. Section 3 is the related work on side-channel attacks within SDN networks, CPUs, and virtual machines; additionally, we discuss the related work on protocol obfuscation.

The remaining sections were divided based on the corresponding project, due to the intricacies of the projects. In Section 4, we discuss the work we conducted with the SDN environment. In Section 4.1, we explain the side-channel attack as well as the attack it is based on. In Section 4.2, we detail the protocols and other pertinent information utilized by the side-channel attack. Section 4.3 describes the experimental environment used when conducting these experiments. Section 4.4 is where we review and discuss the experiment's results. And the final Sections (Sections 4.5 and 4.6) contains concluding remarks and various ideas for future research topics.

In Section 5, we discuss the work we conducted with protocol obfuscation. In Section 5.1, we explain the concept of protocol obfuscation. In Section 5.2, we detail our specific implementation of protocol obfuscation by using the transfer of PDF files. Section 5.3 describes the experimental environments used when conducting these experiments. Section 5.4 is where we review and discuss the experiment's results. And the final Sections (Sections 5.5 and 5.6) is were we give the concluding remarks and various ideas for future research topics.

In Section 6, we summarize both project's concluding remarks.

3 Related Work

3.1 Side-Channel Attacks

The literature on reverse engineering load balancing algorithms is sparse. There has been significant work that utilizes side-channel attacks to obtain system or device information. Similarly, the literature on determining the number of servers in a cluster is also sparse.

There has been some research on mapping a CDN infrastructure; to be more specific, researchers were developing methodologies to optimize a CDN hierarchy. Ghabashneh et al [20] and Zhu et al [57] develop methods to improve the performance CDNs for video streaming. Narayanan et al [42] develop methods to improve the performance of CDNs for downloading an entire web page. Ganjam et al [19] and Liu et al [37] researched methods for improving video streaming through the use of a control plane. These researchers were not directly attempting to map a CDN hierarchy, but their research consisted of comparing the current state of a CDN hierarchy's measurements against the CDN hierarchy with their improved methodology. Thus, they indirectly started to research if an adversary can map a CDN infrastructure.

In short, our side-channel attack uses performance variations to determine the number of controllers in an SDN network. Reviewing various research that utilized performance degradation and improvement in the side-channel attack, we found similar side-channel attacks used in SDN networks, CPUs, and Virtual Machines (VM's).

3.1.1 SDN Attacks

There is a popular SDN side-channel attack used to infer network configuration and device information within SDN networks [46] [10] [48] [47] [36] [35]. The SDN side-channel attack takes advantage of the separation of the data plane from the control plane taking advantage of how the switch does not make any routing decisions. When a switch comes across a mismatched packet, the switch will request a new forwarding rule from the controller. As a result, the first packet without a corresponding forwarding rule will have a long delay. But all of the successive packets will be able to match against a forwarding rule; thus, they will have a low response time, because the switch does not need to ask the controller for a forwarding rule. As a result, the communication delay between the switch and controller, as well as the controller's processing delay, has been removed from the packet's response time. The high to low response time difference was used by Shin et al [46] and Bitfulco et al [10] to determine if the networking devices are configured for either an SDN network or a traditional network. Sonchack et al [48] [47] and Liu et al [36] used the high to low response time difference to accurately infer a switch's existing forwarding rules. Leng et al [35] used the high to low response time difference to determine the amount of memory a switch has for forwarding rules. Azzouni et al [8] used the SDN side-channel attack to determine the idle and hard timeout values in forwarding rules, in hopes of determining the controller application.

Various researchers examined misinforming the SDN network configuration from the controller's perspective [25] [4]. Hong et al [25] discovered many common SDN controllers are susceptible to topology management attacks. The controller uses the Host Tracking Service (HTS) protocol to find the location of a host that is within the network. HTS maps a MAC address to a switch's port. So if an adversary sends a spoofed source MAC address of an existing computer within the network (the victim host), then the controller will think the victim host moved locations. To be more precise, the controller will think the victim host is connected to the switch port that is currently connected to the adversary, so the controller will direct the victim host's network traffic to the adversary. Hong et al [25] and Alhari et al [4] also found a vulnerability in the OpenFlow Discovery Protocol (OFDP) Link Discovery Service (LDS). Because the OFDP LDS protocol is used by the controller to detect links between switches, an adversary can maliciously modify and send LDS packets to the controller so that the controller thinks there are network links that do not really exist. Additionally, an adversary could make ghost switches on the network because the LDS service relies upon broadcast packets and assumes all devices on the network will be honest.

Because of the critical importance of the SDN control plane, various researchers examined numerous methods to prevent the control plane from saturating [6] [5] [34] [11] [45]. Ambrosin et al [6] [5] developed a method that protects the control plane from SYN flood Denial of Service (DoS) attacks. Ambrosin's et al method forced the switch to proxy the connections such that the switch completes a TCP handshake, before the switch sends a forwarding request to the controller. Kuerban et al [34] suggested to simply rate limit the number of packets a switch can send to the controller. Chen et al [11] found the bottleneck of the SDN control path are the edge switches, so they recommend network administrators use specialized processing hardware for the edge switches that utilizes a filtering and verification stage. And Shang et al [45] developed a four module protocol-independent method that does not require additional hardware. Whenever a switch comes across a mismatched packet, Shang et al's method has the switch send its forwarding rule request to one of its neighbors so that the switch's neighbor can ask the controller for a new forwarding rule, rather than the switch directly asking the controller for a new forwarding rule. Additionally, the controller will filter out the forwarding rule requests that correspond to an attack, and the controller will install a new forwarding rule that will either monitor the network connection or forward the packets.

As mentioned previously, it is important to use multiple controllers within an SDN network so that the management channel does not become a single point of failure. Placing the controller in an optimal location within the network, called the controller placement problem, depends on the network's requirements and status. If an adversary is able to determine the number of controllers in an SDN network, then the adversary would want to determine the controller application running on the controllers as well. While little work for fingerprinting a controller application has been done, Azzouni et al [8] mapped specific controller applications to its default forwarding rule idle timeouts, hard timeouts, OFDP intervals, and the controller's processing time. Azzouni's et al statistics can be used as a starting point to determine whether the controller application can be ascertained while the attacking host does not have access to the controller.

3.1.2 CPU Attacks

Irazoqui et al [29] utilized a side-channel attack to reverse engineer the slice selection algorithm within Intel processors. First, Irazoqui et al inserted data into a known section of the last level cache; second, they tried to fetch this data after a period of time. When the data was fetched and still resided in the last level cache, the data had a low response time. Otherwise, the data had a high response time because the data was evicted. After the response times were obtained, the response times were inserted into a matrix that determined the slice selection algorithm. Irazoqui's et al side channel attack is similar to the side-channel attack utilized by Varadarajan et al [51], which tried to fetch known data from the cache and look for performance degradation.

Various researchers examined reverse engineering cache replacement policies; more specifically, they examined numerous methods that ascertained whether the implemented policy was either the Least Recently Used (LRU) or the First-In-First-Out (FIFO) policy. Abel et al [2] was able to derive an equation that calculates the probability of data eviction, after n cache misses in LRU caches. Dehghan et al [13] was able to reverse engineer LRU and FIFO policies by calculating the policy's utility function. The policy's utility function is based on the policy's hit probabilities. Reverse engineering cache replacement policies relates to our project's goal because determining the number of controllers in an SDN network requires knowing the probability of a packet hitting the designated controller. The methodologies differ because cache policies and load balancing policies can use different metrics. For example, a load balancing algorithm can be based on LRU. We argue having a load balancing algorithm based on LRU is unlikely because the load balancing algorithm should distribute the load based on the server's current processing load. In contrast, the LRU load balancing algorithm will distribute the load based on the last server used.

3.1.3 Virtual Machine Co-Residency Attacks

Due to the popularity of cloud computing, it is natural for VMs to be embedded into the infrastructure. A co-residency attack (or co-location attack) is a malicious entity attempting to control two or more VMs on the same physical host. The concept of a co-residency attack is for an adversary to (1) control multiple VMs and (2) conduct a procedure that confirms if two VMs are on the same host. Varadarajan et al [51] used a side-channel attack to obtain the confirmation with one VM as a sender and another VM as a receiver. The sender uses atomic memory operations to temporarily lock the memory bus while the receiver periodically measures the bus by sending memory requests that are crafted to be cache-misses. Varadarajan's et al method used performance degradation to discover if more than one VM is on the same host. In other words, the CPU performance degraded because of a cache-miss, so the side-channel attack confirmed multiple VMs are on the same host.

3.2 Protocol Obfuscation

Protocol obfuscation at its core is utilizing a protocol between two hosts in a format that does not accurately represent (i.e. hides) the protocol, where this protocol is forwarded through a network infrastructure that may be monitored. So in this respect, the community's research on this topic can be expanded to various encryption standards such as IPsec [31] and TLS [44], because the target protocol (ex: TCP or HTTP) is hidden in a cyrptographic format. In addition, there are various standards on overlay networks that can be incorporated into the protocol obfuscation research area, such as Virtual Private Networks (VPN) [18], The Onion Ring (TOR) [50], and the Invisible Internet Project (I2P) [28]. A VPN establishes a tunnel between a client and the client's target network. After a client successfully establishes a VPN tunnel, they can remotely access the target network's access, web, and/or file servers; furthermore, the target protocol (ex: SSH, HTTP, or FTP) is encrypted (i.e. obfuscated) while it is forwarded through this tunnel. A similar statement can be made about TOR and I2P because they hide the host's target protocol (HTTP) in a cryptographic format, among other things; except these overlay networks hide the target protocol with multiple layers of encryption in order to provide the client anonymity.

Before discussing established protocol obfuscation research, it is important to review the various research that has already been conducted for the purpose of identifying protocols based on a network's characteristics, because these results propel the need for protocol obfuscation. Hubballi et al [27] identified network sessions by developing an application called BitCoding, which identifies a network session by calculating a bit-level signature from the first bits of the session. Through their experiments, they discovered that the amount of bits required to identify a session's protocol is dependent on the protocol in use. Papapetrou et al [24] utilized the entropy distribution of the size of the DNS packet to identify the protocol that was obfuscated by a DNS tunnel. Their approach achieved a 75% true positive rate for the HTTP and FTP protocols. Additionally, Aiello et al [3] utilized machine learning tools to identify DNS tunnels while keeping the detection scheme robust and able to scale to a large set of data. Crotti and Dusi et al [12] [15] developed an application called Tunnel Hunter that utilized statistical analysis to identify a session as a true HTTP or SSH session or a session obfuscated by an HTTP or SSH tunnel. They showed that the Tunnel Hunter had a 99% accuracy rate for detecting legitimate HTTP and SSH sessions. He et al [21] utilized a convolutional neural network to identify an application obfuscated by an SSH tunnel, and their experimental results had an accuracy of 95.8% for the five applications they analyzed. The researchers' successful identification of un-obfuscated protocols is a primary reason for the continuation of research in the area of protocol obfuscation.

There has been various research into protocol obfuscation. These researchers either built upon already established obfuscation methodologies or developed new obfuscation techniques. For example, the widely referenced obfuscation application StegoTorus [54] is built upon Tor's obfuscation techniques. It was developed to obfuscate a Tor session as a HTTP session, which performed successfully during their experiments. Winter et al [55] essentially built upon Tor's obfsproxy protocol [41] to prevent monitoring entities from identifying Tor's obfs2's and obfs3's traffic. They enhanced the obfsproxy protocol by modifying the packet's IPD and packet sizes, as well as adding another layer of encryption to the data. Zink et al [58] demonstrated how a slight modification in a protocol's implementation can circumvent a monitoring entity's session classification system by obfuscating the handshake procedure for the BitTorrent protocol. While they were able to obfuscate the BitTorrent protocol, their real intent was to examine how the session classification models are highly implementation dependent. They wanted to show this by making a minuet change to avert detection.

Additionally, Houmansadr et al [26] developed a new obfuscation methodology and application called SWEET primarily utilizes publicly available email servers as the tunnel for the client's covert communications. Thus, a SWEET client can be within a monitoring entity's environment but the SWEET server has to be outside of the monitoring entity's environment. Instead of directly transmitting the request to the designated destination, the SWEET client will email the request to a publicly available server; then, the SWEET server will obtain this email, conduct the request on behalf of the client and email back the designated destination's reply. While their methodology was designed for web browsing, the results indicate it can potentially be used for interactive sessions. Mohajeri et al [39] developed a new obfuscation application called SkypeMorph, which masks a client's connection to a Tor bridge (i.e. an entry point) as a Skype video call. Since Skype encrypts all of its traffic, it pairs well with Tor's encrypted traffic; additionally, their application uses the UDP protocol and modifies the session's IPD and packet sizes in order to obfuscate the Tor protocol as a Skype session. Dyer et al [16] developed the Marionette system, which allows the user to programmatically specify by using protocol templates and stateful connections.

Researchers also conducted methods on breaking through the obfuscation to determine the target data or type of data. Wang et al [53] analyzed the reliability of various obfuscation applications by analyzing the obfuscated traffic with semantic, entropy, and machine learning tests. And interestingly, they determined entropy and machine learning tests can successfully detect specific obfuscated traffic, but the semantic tests can produce false positives based on whether or not the designated server's software developer followed the standards. Anderson et al [7] developed a multi-session model to accurately fingerprint an OS, which in itself is interesting; however, they also determined this multi-session model is still effective against OS fingerprint obfuscation methodologies that do not holistically obfuscate the OS. Hjelmvik et al [23] determined an obfuscated session's statistics could be used to identify the protocol; and, their network analysis application generated better results when it was optimized to identify a wide range of protocols instead of combining the individual protocol's optimized settings, for which they also recommend their analysis application be optimized for the operating environment and applications. These papers disclose how the field of obfuscation is growing, how monitoring entities can break through established obfuscation techniques, and that the current methodologies need to be improved upon.

Some researchers surveyed the topic of obfuscation in the context of general obfuscation and identification methodologies. Dixon et al [14] concluded that the previous work does not completely show whether if the researcher's obfuscation methodologies are viable in a user's live environment or not, which is primarily due to the researcher's limited to non-existent access to the monitoring entity's or their own network's dataset. Therefore, most of the methodologies will be implementation dependent and will be developed for a specific target's operating environment. This prevents the obfuscation methodologies from being utilized throughout the community as a whole, and makes it difficult for future researchers to build upon them. Verma et al [52] analyzed whether adversarial machine learning techniques can identify a network session's protocol, even when its packet sizes and inter-packet delays are being obfuscated by the CarliniWagner L2 algorithm. They concluded the algorithm can be used for obfuscation; but more importantly, the wired and wireless networks have several critical differences to characterizing a network session, so further research is required to determine if transmitting the data wirelessly can inherently obfuscate the session. So, there is still a lot of work to be done within the field of obfuscation.

4 SDN

4.1 Concepts

The goal of this work is to determine if there are multiple controllers operating on an SDN network, by using the reconnaissance attack's techniques. The reconnaissance attack is the side-channel attack that maps the flow tables within an SDN switch. Consider a multiple controller SDN network containing a secondary controller. An adversary can force an SDN switch to send requests to the secondary controller by saturating the primary controller with consecutive forwarding requests. Since the secondary controller is not congested, it will have a faster response time than the primary controller; thus, this faster response time will imply to the adversary that there are multiple controllers operating on the network.

4.1.1 Base Reconnaissance Attack Concept

In order for an adversary to successfully perform the reconnaissance attack [46] [10] [48] [47] [36] [35], it is crucial for the reconnaissance attacker to know the Round-Trip Time (RTT) of a mismatched and a matched packet. Ideally, the adversary is directly connected to the SDN switch they are attacking (the victim switch) because it provides an accurate RTT value for the adversary. The reconnaissance attack sends a request packet to another host computer (the target host or victim host) as the primary method used to obtain the RTT value; furthermore, the target host is located within the SDN network. However, an ideal target host is a host that is directly connected to the victim switch. This provides an accurate measurement of the reply packet's delay because it reduces the chance of a delay coming from a congested intermediate switch. A close target host will also ensure that the reconnaissance attack will detect the switch's forwarding rule installation delay. On the other hand, when the distance between the target host and the adversary is too large, the installation delay could be masked by the network transmission time. In other words, the network transmission time might make it difficult for the reconnaissance attack to detect the controller installing a new forwarding rule, which means it will be difficult for the controller side-channel attack to detect the network utilizing the secondary controller.

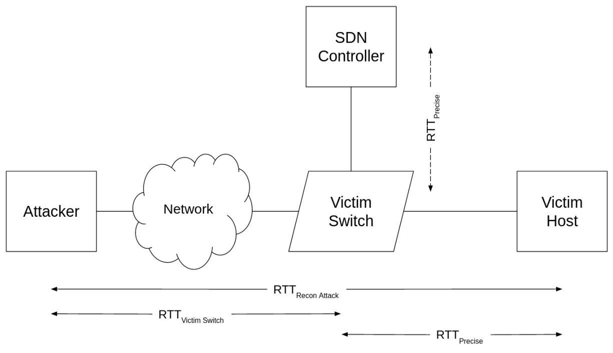


Figure 5: Non-Ideal Victim Switch

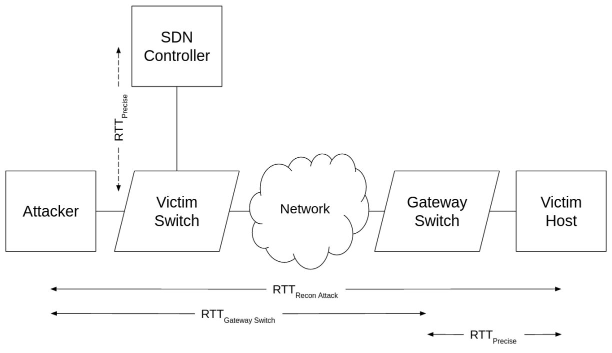


Figure 6: Non-Ideal Target Host

If the adversary is not directly connected to the victim switch, then the adversary can obtain an RTT time between themselves and the victim switch while conducting the reconnaissance attack (Figure 5). The adversary will want to remove as much network delay as possible, so that they obtain a precise RTT response time. By calculating the difference between the RTT of the reconnaissance attack and the RTT of the victim switch, the adversary will obtain an RTT response time as if they were directly connected to the switch ($RTT_{Precise} = RTT_{ReconAttack} - RTT_{VictimSwitch}$). Similarly, the adversary can obtain a refined RTT response time when the target host is not directly connected to the victim switch (Figure 6). The adversary will have to obtain an RTT response time between themselves and the switch that the target host is directly connected to (gateway switch, Figure 6); moreover, the RTT of the target host's gateway switch needs to be determined only for the data plane. Then, by calculating the difference between the RTT of the reconnaissance attack and the RTT of the target host's gateway switch, the adversary will obtain a refined RTT response time ($RTT_{Precise} = RTT_{ReconAttack} - RTT_{GatewaySwitch}$). Additionally, if an ideal target host cannot be found, an adversary could slightly divert from the reconnaissance attack methodology by having the target host be the host the adversary is utilizing. Since the reconnaissance attack methodology relies on the default OS response from the target host, this additional processing load will not cause the attacking host, which is the same as target host, to be the bottleneck. When the adversary is also on the target host, it is beneficial for the adversary to have a valid second set of addresses (MAC, IP, etc) that the controller can use in the forwarding rules. As long as the adversary uses another set of addresses, the controller will view the adversary and target host as separate hosts.

After obtaining the base case RTT for a matched packet, the adversary's next step is to obtain the base case RTT for a mismatched packet by transmitting a specially crafted packet. This specially crafted packet is guaranteed to be processed by the controller, before it reaches the target host, because this packet will contain an address that is:

- A value that does not identify the adversary conducting the attack; and
- A value that does not exist on the network (non-existent value); and
- A value can either be an address, protocol port number, or both.

These specially crafted packets are the main artifacts that the controller side-channel attack will use for identifying the number of controllers running within an SDN network.

As mentioned in Section 2.2, when a switch receives a packet, it will either have a forwarding rule in the flow table and transmit the packet based on the rule (matched packet, Figure 2), or it will not have a forwarding rule for the packet and ask the controller for a new forwarding rule (mismatched packet, Figure 1). This means their response time for matched packets should be faster than mismatched packets. A mismatched packet's increase in response time comes from the additional transmissions between the switch and controller, the controller processing a new forwarding rule, the switch installing the new forwarding rule, and the switch forwarding the packet with the new forwarding rule. To summarize, matched packets will have a network delay time of only t_{lookup}, where

$$t_{lookup} = \text{the switch's lookup time}$$

But mismatched packets will have an additional network delay time of $t_{controller}$, where

$$t_{controller} = \text{the communication time between the switch and controller} + \text{the controller's processing time}$$
$$+ \text{the rule's installation time on the switch} + \text{the switch's forwarding time}$$

So the total networking delay for mismatched packets is

$$\text{total network delay} = t_{lookup} + t_{controller}$$

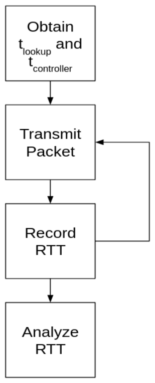


Figure 7: Reconnaissance Attack Methodology Overview

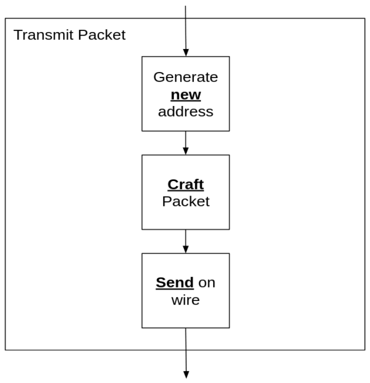


Figure 8: Reconnaissance Attack Methodology Transmit Step

It is impossible to obtain these individual network delay times unless the attacking host is also the target host, because the packet's response time will consist of the network delay time of both the request and reply packet's t_{lookup} and $t_{controller}$ times. This means the attacking host's mismatched packet RTT will consist of $t_{lookup} + t_{controller}$ when the packet reaches the victim host. The victim host's reply packet will also be a mismatched packet because it will be the first time the SDN network sees the victim host replying to the mismatched request packet, so the reply packet will also consist of $t_{lookup} + t_{controller}$ from the time the victim host transmits the reply packet to the attacking host's reception of the reply packet. Therefore, a packet's response time will have a total network delay of $2 * (t_{lookup} + t_{controller})$ starting from the moment the attacking host transmits the request packet to the time the attacking host receives the reply packet. On the other hand, when the attacking host is also the target host, the packet's response time will consist of the network delay time of only the request packet's t_{lookup} and $t_{controller}$ times. As the attacker is also the target, when the request packet reaches the target, the attacker (being the target) knows exactly when it arrived and can readily compute the response time without transmitting a reply packet.

In either case, the delay times become variable and automatically conform to the SDN network that is being attacked. Since the reconnaissance attack has a predefined target host, every packet will consist of the same network delays that form each packet's response time. In conclusion, the reconnaissance attack will see all packets with a total network delay of $2 * (t_{lookup})$ as t_{lookup} and $2 * (t_{lookup} + t_{controller})$ as $t_{lookup} + t_{controller}$ instead.

By pairing the knowledge of how an SDN switch reacts to mismatched packets and utilizing specially crafted packets that take advantage of OpenFlow features, an adversary can perform a controller saturation attack that strategically and constantly forces the switch to send forwarding requests to the controller. The reconnaissance attack's methodology is basically for an adversary to obtain the t_{lookup} and $t_{controller}$ times, transmit packets, and review their RTT times. To be more precise, the adversary will look for the difference between a high RTT ($t_{lookup} + t_{controller}$) and low RTT ($t_{lookup}$). If a transmitted packet:

1. Has a high RTT that is greater than or equal to $t_{lookup} + t_{controller}$,

then the reconnaissance attack can infer that

1. The controller installed a new forwarding rule onto the switch for the packet.

But if the transmitted packet:

1. Has a low RTT of t_{lookup}.

Then the reconnaissance attack can infer that:

1. A forwarding rule already exists on the switch for the packet.

This method is displayed in Figure 7, the details of constructing the specially crafted packets is displayed in Figure 8, and the steps for determining if a forwarding rule exist is displayed in Figure 9.

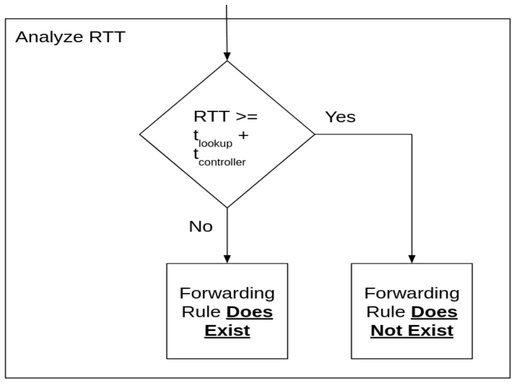

Analyze RTT
RTT >= t_lookup + t_controller
Yes
No
Forwarding Rule Does Exist
Forwarding Rule Does Not Exist

Figure 9: Reconnaissance Attack Methodology Analyze RTT Step

While the reconnaissance attack can be effective, unfortunately there is no guarantee that a packet's high RTT is due to the $t_{controller}$ delay. An adversary does not know if an unforeseen circumstance, such as the switch getting congested, occurred during the reconnaissance attack. In order to ensure a packet's high RTT is caused by $t_{controller}$ and not by an unforeseen circumstance, the reconnaissance attack should be performed at different times of the day so that the adversary can obtain a large sample size. This large sample size allows an adversary to generate a base case by comparing the RTTs.

4.1.2 Computing the Number of Controllers

4.1.2.1 Overview The controller side-channel attack methodology will determine the number of controllers operating within an SDN network. The methodology will analyze the packets' response times based on the load balancing algorithm. As mentioned in Section 2, the controller side-channel attack methodology will also inherently determine the load balancing algorithm itself because the operating load balancing algorithm influences patterns in the packets' response times.

Similar to the reconnaissance attack, the controller side-channel attack's initial and simplest methodology is for an adversary to obtain the t_{lookup} and $t_{controller}$ times, continuously transmit packets, and review their RTT times (Figures 7 and 8). While the controller side-channel attack inherits concepts from the reconnaissance attack, it generates packets that have the sole purpose of performing a controller saturation attack, and ideally it won't have to transmit a packet that utilizes one of the switch's forwarding rules. So the results will differ because all of the RTTs should be greater than or equal to $t_{lookup} + t_{controller}$. As displayed in Figure 10, the RTT patterns will indicate the number of controllers and the load balancing algorithm used in the management plane.

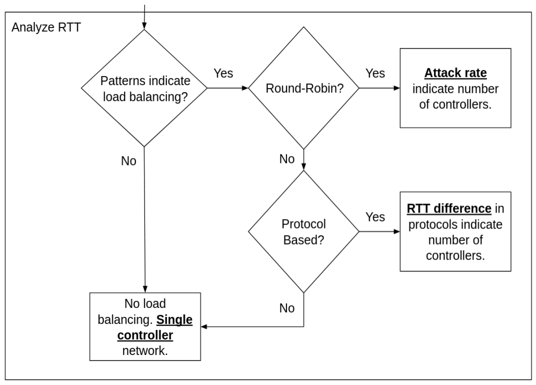

Analyze RTT
Patterns indicate load balancing?
Yes
Round-Robin?
Yes
Attack rate indicate number of controllers.
No
No
Protocol Based?
Yes
RTT difference in protocols indicate number of controllers.
No
No load balancing. Single controller network.

Figure 10: Controller Side-Channel Attack Methodology Analyze RTT Step

4.1.2.2 Round Robin Load Balancing Algorithm As mentioned in Section 2.2.1.1, we want to devise a methodology that can determine the number of controllers operating within the SDN network. This methodology is based on the controller side-channel attack methodology, but this methodology will be conducting an aggressive controller side-channel attack, on the control plane. By using the response times of a single controller SDN network as the control experiment, an adversary can divide the controller side-channel attack's results with the control experiment results in order to determine the number of controllers operating within the network. So if:

1. The controller side-channel attack's response times are an order of magnitude(s) lower than the control

experiment's response times.

Then the controller side-channel attack has identified:

1. It is a multiple controller SDN network;
2. The Round Robin load balancing algorithm is utilized; and
3. The number of controllers that are operating within the network, because the number of controllers equals to the performance increase of the controller saturation attack.

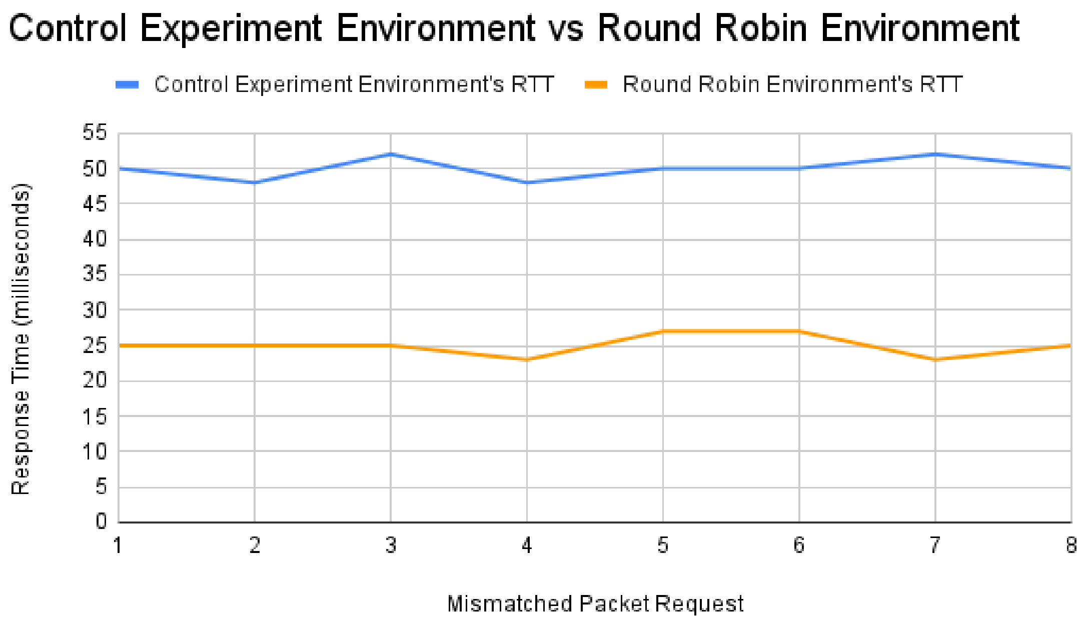


Figure 11: Round Robin Example. An adversary can divide the control experiment environment's average response time of 50 milliseconds by the Round Robin environment's average response time of 25 milliseconds to determine the average response time difference is two orders of magnitude different. Therefore, they have determined there are two controllers operating within the SDN network.

For example, let's say an SDN network has two controllers operating within the control plane while using the Round Robin load balancing algorithm. Additionally, let's say the control experiment's average response time was 50 milliseconds and the controller side-channel attack's average response time for the Round Robin environment is 25 milliseconds (Figure 11). An adversary can divide 50 by 25 to determine two orders of magnitude difference in the average response times. Thus, they have determined there are two controllers within the SDN network and the Round Robin load balancing algorithm is operating within the network.

4.1.2.3 Protocol Based Load Balancing Algorithm As mentioned in Section 2.2.1.2, the controller side-channel attack will take advantage of this load balancing algorithm's side effect in order to determine the number of controllers operating within the SDN network.

In order for the packets to be processed by different controllers, the controller side-channel attack will simultaneously transmit packets that contain different protocols. The goal is for the controller side-channel attack to congest the target controller(s) by flooding the network with packets that contain a certain protocol or set of protocols. In addition, the controller side-channel attack will periodically transmit packets that contain another protocol, where these packets are guaranteed to be processed by a non-congested controller. Since the network contains a mix of congested and non-congested controllers, the packets from the non-congested controller will have a lower response time than the packets from the congested controller(s). The minimum number of protocols needed for the attack is equal to the number of controllers that are operating within the SDN network.

For example, consider we have two controllers in a multiple controller SDN network. Then, the controller side-channel attack requires the use of at least two different protocols. The attacker targets the first controller, and floods the network with packets of the first protocol. The attacker also targets the second controller, but only periodically transmit packets of the second protocol. Since the second controller has only a few packets to process, those packets will have a lower response time than the packets processed by the first controller.

To summarize how the controller side-channel attack determines the number of controllers that are operating on the SDN network. The controller side-channel attack will transmit packets of varying protocols; then, it will compare the RTTs of the packets that have contrasting protocols. It will check if there exists packets:

1. That were transmitted at the same time;
2. Contain different protocols; and
3. Have a noticeable difference in RTT time.

If these conditions are met, then the controller side-channel attack has identified:

1. The protocol based load balancing algorithm is being utilized;
2. The specific protocol(s) that each controller is designated to handle; and
3. The number of controllers that are operating within the network, because the number of controllers equals to the number of protocols that have contrasting RTT's.

4.1.3 Defense

As mentioned before, the controller (and reconnaissance) side-channel attacks have to determine the RTT of a mismatched and matched packet. An administrator could configure the SDN network's switches on the data plane to always transmit all of its packets to the controller so that the controller can use PACKET_OUT messages to forward the packets throughout the network. The adversary would not be able to determine the RTT of a matched packet because none of the packets will be processed by just a switch. So, all of the packets' RTT will be $t_{lookup} + t_{controller}$.

This type of configuration requires every single packet to be a mismatched packet. However, this would go against the dynamic nature of SDN, specifically the purpose of a network's use of a control plane, and the sole purpose of the data plane. This configuration would overload the controller and negate the purpose of switches because the controller will have to forward all of the network's packets throughout all of its hops across the network, which is practically the same as having all of the endhosts connected to the controller instead of a switch. We decided to configure the SDN network's controller (aka the control plane) so that it is difficult for an adversary to determine the RTT of a mismatched packet.

Instead of preventing an adversary from determining the RTT of a matched packet, we decided to only allow an adversary to determine the RTT of a matched packet. In other words, we are going to try to prevent an adversary from obtaining the $t_{lookup} + t_{controller}$ time and just allow the adversary to obtain the t_{lookup} time. This can be achieved by simply preventing the controller from forwarding the mismatched packet for the switch with an OpenFlow PACKET_OUT message. Though, the controller will still be configured to install a new forwarding rule onto the switch for the mismatched packet. This configuration means that the mismatched packet will be dropped by the network and the packet will need to be re-transmitted for the destination to receive the packet.

A packet's RTT will only be the t_{lookup} time because the adversary will get a reply for packets that are only processed by the switch (aka the data plane). This will prevent the adversary from determining the control plane's processing load, because the adversary will not be able to obtain a packet that has been processed by the controller. In other words, the $t_{lookup} + t_{controller}$ time will no longer be a factor in the RTT of a packet.

The OpenFlow protocol allows an SDN switch to buffer a mismatched packet and transmit a pre-defined subset of the packet's bytes plus a buffer ID to the controller. This buffering configuration can be used instead of transmitting the entire mismatched packet to the controller. In order for the defense to support this buffer configuration, the controller will take two actions on the switch for each mismatched packet. The controller's first procedure is to install a new forwarding rule onto the switch for the mismatched packet. The second procedure is for the switch to transmit a PACKET_OUT message telling the switch to drop the the mismatched packet. This second procedure will increase the number of packets transmitted within the control plane, which could lead to faster congestion, but it allows the defense to follow the OpenFlow standards and support to various environments.

4.2 Experimental Design

4.2.1 Overview

The controller side-channel attack utilized one or two protocols depending on the load balancing algorithm because these experiments conducted layer 2 forwarding (MAC learning). If the control experiment environment or the Round Robin load balancing algorithm was implemented, then the controller side-channel attack utilized just the ARP protocol. But if the protocol based load balancing algorithm was implemented, then the controller side-channel attack utilized the ARP and ICMP protocols. Refer to Section 4.2.2 for more details about the protocols utilized with each corresponding load balancing algorithm. For these experiments:

1. We will be taking on the role of the adversary;
2. The number of addresses (t) will be equal to the number of forwarding rules that the switch's cache space can contain; and
3. The controller side-channel attack's saturation of the control plane will primarily be due to the request packets.

The Address Resolution Protocol (ARP) is a layer 2 protocol used for devices to obtain the IP address associated with a MAC address, the controller side-channel attack used this widely acceptable protocol to its advantage. These ARP packets contained a spoofed source MAC address and the destination IP address of the target host. More specifically, the MAC header used the spoofed MAC address as the source address and the MAC broadcast address as the destination address; plus, the ARP header used the same spoofed MAC address as the Sender Hardware Address, a spoofed IP address as the Sender Protocol Address, the MAC network address for the Target Hardware Address, and the target host's IP address for the Target Protocol Address.

The Internet Control Message Protocol (ICMP), specifically the ping message, is a common layer 3 protocol used to determine if a device either exists, is up, or is reachable on the network. The controller side-channel attack took advantage of the ICMP protocol by sending ping requests to the target host. The ping messages contained a spoofed source MAC address and the destination MAC address of the target host. More specifically, the MAC header used the spoofed MAC address as the source address and the target host's MAC address as the destination address; plus, the IP header used a spoofed IP address as the source address and the target host's IP address for destination address.

The target host was configured with a default route of the SDN network, refer to Section 4.3. Thus, the spoofed source IP address was an unused IP address that was outside or within the subnet of the adversary's valid IP address, because the default route guaranteed the ping reply was forwarded to the adversary. Additionally, the controller conducted layer 2 forwarding and did not use the IP address within the forwarding rule's match field; so again, the controller side-channel attack was not limited to using IP addresses within the adversary's valid IP address subnet.

In either case, these crafted packets were all mismatched packets because each packet had a different spoofed addresses, which forced the switch to inform the controller of the ingress physical port that an unseen source MAC address came in on. So, the controller knew how to forward packets that are destined to these spoofed MAC addresses. A new spoofed IP address was generated for each spoofed MAC address in order for the controller(s) and all of the devices to map a single IP address to a single MAC address.

The first ARP or ICMP reply was a mismatched packet because the switch did not have forwarding rules initially. However, all of the remaining ARP and ICMP reply packets were matched packets because of the forwarding rule's match field, refer to Section 4.3. This means the first packet's response time was $(t_{lookup} + t_{controller}) + (t_{lookup} + t_{controller})$ and the remaining (second, third, etc) packets' response times were $(t_{lookup} + t_{controller}) + (t_{lookup})$. This is acceptable behavior because the request packets always inferred the control plane, which ensured the response times were all greater than a match packet's response time of $(t_{lookup}) + (t_{lookup})$.

The adversary's first step would be to find unused MAC and/or IP addresses. While out of scope for these experiments, there are a variety of methods an adversary can obtain spoofed addresses. For obtaining spoofed MAC addresses, if the network is using OSI layer 2, we can safely assume that the adversary is on the SDN network. Consequently, the adversary can obtain these MAC addresses by monitoring ARP packets. If an adversary records the MAC addresses of valid ARP requests, they can easily determine the MAC addresses that do not exist on the SDN network. In short, the adversary just has to determine the addresses that are not used by valid ARP requests, where a valid ARP request is a request that is not sent maliciously. Since the ARP requests are broadcasted and the adversary is on the SDN network, we can safely assume that all ARP requests will reach the adversary.

For obtaining spoofed IP addresses, if the adversary is located within the SDN network, they can monitor the network for DHCP discover, ARP requests, and if applicable DHCP request packets. Since these types of request packets are broadcast, we can safely assume that the adversary will obtain these packets. The adversary can argue all devices that utilize DHCP will lease an IP address with the same amount of time as the attacker's DHCP lease. The adversary can also compare and contrast a current ARP request header's Sender Hardware Address against the Sender Protocol Address so that they can confirm if a device maintained their IP address, after a DHCP renewal. If the adversary is not within the SDN network, they can determine if an IP address is unused by sending a ping request to the potential address. If the ping does not get a reply, then the adversary can safely assume that the address is not in use or not routable or not answering pings.

4.2.2 Load Balancing Algorithms

4.2.2.1 Round Robin Load Balancing Algorithm The controller side-channel attack used just the ARP protocol against the control experiment environment (i.e. single controller SDN network) and Round Robin load balancing algorithm because the control experiment environment and Round Robin load balancing algorithm treated all mismatched packets equally. Additionally, the control plane was not configured to process mismatched packets differently based on its protocol. Essentially for these environments, the controller side-channel attack performed a controller saturation attack. This method allowed the controller side-channel attack to determine the number of controllers when all of the controllers are executing the same or different applications due to the comparison against the control experiment.

4.2.2.2 Protocol Based Load Balancing Algorithm The controller side-channel attack used the ARP and ICMP protocols against the protocol based load balancing algorithm. Unlike the Round Robin load balancing algorithm, this algorithm was configured to treat mismatched packets containing the ICMP protocol differently, as specified in Section 4.3. The ARP packets were processed by the default controller, and the ICMP packets were processed by the secondary controller. More specifically, the ARP packets were flooded onto the network in order to saturate the default controller; but, the ICMP packets were periodically transmitted to the secondary controller.

The controller side-channel attack was configured to transmit the ICMP protocol for every tenth mismatched packet; whereas the other mismatched packets contained the ARP protocol. So, consider the controller side-channel attack is told to transmit ten mismatched packets for the protocol based load balancing algorithm, then the ten mismatched packet's protocols will be in the following sequence: ARP ARP ARP ARP ARP ARP ARP ARP ARP ICMP. However, if the adversary informed the controller side-channel attack that it would be transmitting a single packet for the protocol based load balancing algorithm, then the controller side-channel attack would transmitted two packets, where one of the packets contained the ARP protocol and the other contained the ICMP protocol. This was necessary because the methodology required a mismatched packet's response time from both the default and secondary controllers.

The controller side-channel attack provided the ICMP packets with its own spoofed MAC and IP address. It was crucial for the ICMP packets to have its own spoofed MAC address, and not re-use a spoofed MAC address from the previous spoofed ARP mismatched packet because the experimental environment consisted of layer 2 forwarding. But technically, the ICMP packets could have used the adversary's valid IP address because the experimental environment did not consider it when forwarding packets. We decided against this in order for all IP addresses to correspond to a single MAC address. Whether the controller side-channel attack was used in an experiment for the Round Robin or protocol based load balancing algorithm, it was optimized to first craft all of the packets used within the attack before sending any of them onto the network.

4.2.3 Defense

We will now detail the process for establishing a connection when an SDN network decides to utilize this defense. When an adversary sends a mismatch packet request onto the network, with the hopes of obtaining an RTT for infrastructure reconnaissance, the adversary will not obtain a response because the packet was dropped. However, their mismatched packet was able to cause the controller to install a forwarding rule onto the switch.

When the adversary re-sends this packet request onto the network for a second time, the packet becomes a matched packet and gets forwarded to the victim host because the switch has a forwarding rule for it. The victim host, whom is unaware of the attack, responds to the adversary's packet. The victim host's reply packet could be a matched or mismatched packet, depending on the environment's configuration. Let us considering the environment stated within Section 4.3; in other words, the controller does not install a forwarding rule for the victim host's reply packet at the same time it installed a forwarding rule for the adversary's mismatched request packet. This means the victim host's reply packet is a mismatched packet and is dropped by the network, but a new forwarding rule is installed onto the switch for the victim host's future reply packets.

The adversary will need to re-send this packet request onto the network for a third time in order to finally obtain a response time. The adversary's request packet obtains a response because the request packet is a match packet, due to the first transmission of this packet. Additionally, the victim host's reply packet is also a matched packet because of the second transmission of the request packet. All of this allowed the third transmission of the adversary's request packet to successfully traverse to and from the victim host. However, the RTT of the third transmitted packet will only be crafted by the t_{lookup} time because the request and reply packets were both matched packets, so the adversary will be unable to easily use a mismatched packet's RTT for infrastructure reconnaissance.

If an adversary was persistent and still tried to perform a timing based infrastructure reconnaissance attack while this defense is active, then the adversary would need to dump their packets onto the network multiple times in order to get any semblance of information. To be more precise, an adversary would need to loop through their mismatched packets numerous times in order to obtain the reply packets as soon as possible. The loop is very important because the adversary does not know how long it will take for the controller to process a session's mismatched packets so that a single request packet can obtain a reply; in other words, the adversary does not know the $t_{controller}$ time. If the adversary could somehow still be able to infer the network's infrastructure, which uses this defense, through a timing based reconnaissance attack, we argue that it would take the adversary more resources to obtain the same or less information about the infrastructure because of this defense.

The controller side-channel attack will only subtly change from the details provided in the above section. As mentioned above, the experimental environment will dictate the protocol(s) that the controller side-channel attack will utilize. On the other hand, the controller side-channel attack will sequentially flood the network with the same mismatched packets multiple times instead of sending a packet only once. The attacking program will sequentially loop through the packets to transmit them because an adversary will not know how long it takes to install a packet's corresponding forwarding rule onto the switch.

4.3 Experimental Environment

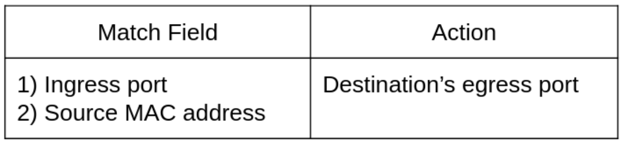
4.3.1 Overview

These experiments were conducted using OvS and the Python SDN controller framework Ryu version 4.34 with Python version 3.6.9. The default implementation of OvS does not have a load balancing mechanism for a multiple controller setup. So we had to modify and install OvS from the source code, for the experiments that required OvS to support the load balancing algorithms. The Python SDN framework Ryu comes pre-packaged with various types of SDN controllers; however, these experiments utilized the pre-packaged simple layer 2 controller. Detailed instructions on how to set up these experiments are located in the Appendix.

OvS is an implementation of a virtual SDN switch; therefore, the experimental environment was virtual and was contained within a single host. The host machine that operated the virtual environment consisted of the following hardware specifications:

1. Two Intel Xeon E5-2637 v3 3.5Ghz processors;
2. 32GB of RAM that operated at 2133MHz; and
3. 931GB of disk space that operated up to 6Gbps.

This host had a disk write speed of 125 MBps and read speed of 1.6 GBps [9]; additionally, it utilized the Ubuntu 18.04.6 64bit OS with kernel version 4.15.0-162-generic. As for the guest OS'es (attacking and victim hosts), this environment used virtual machine instances of Mininet [38] due to its quick and easy setup. The Mininet OS instance for the victim host was additionally configured with a default route that would transmit packets to the SDN network. This allowed the control plane to perform any and all routing decisions for all devices within the network. The OvS switch ("ovs-vswitchd") and base application ("ovs-vsctl") version was version 2.9.8.

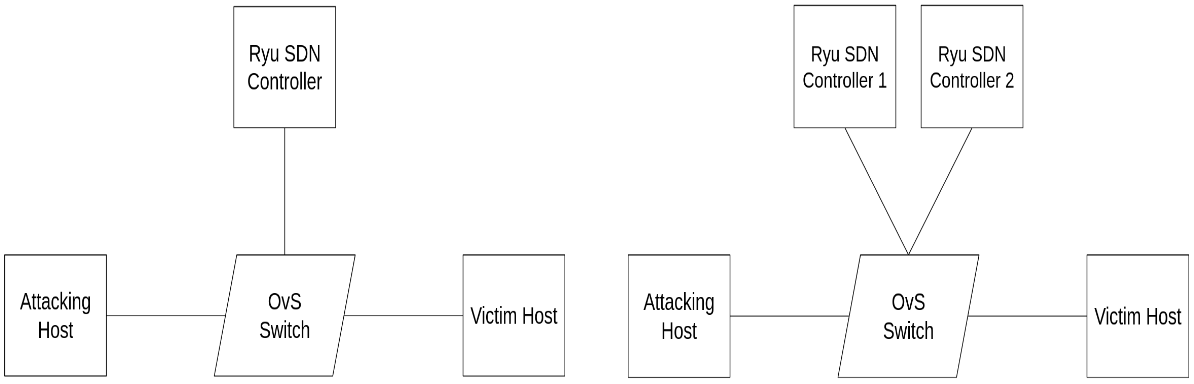


Match Field	Action
1) Ingress port 2) Source MAC address	Destination's egress port

Figure 12: Forwarding Rule Structure

As mentioned earlier, the controller application utilized was a pre-packed simple layer 2 Ryu controller. This application was chosen so that we can replicate the experimental environment detailed by Sonchack et al [48] [47]. This layer 2 Ryu controller in particular would have a forwarding rule match against the packet's ingress port and source MAC address. So essentially, the controller is routing packets based on where the packet is coming from. The controller application would keep a record of the point of origin for each packet's address, in order to determine how to route a packet through the network. And if the controller obtained a packet destined for an address that was not seen by the controller, then the controller would broadcast the packet. For these reasons, the switch's forwarding rules had to be cleared before each experiment; and, the controller application(s) had to be restarted for each experiment. It is important to specify that the Ryu controller did not install a forwarding rule for the broadcast address. This is a crucial specification because this made all of the ARP request mismatched packets; in other words, all of the ARP requests were treated as mismatched packets.

4.3.2 Topology



(a) Single Controller SDN Network (b) Multiple Controller SDN Network

Figure 13: Experimental Environments

The objective of these experiments were to determine the number of controllers that were operating within the SDN network, rather than determining the efficiency of the network's size or the scalability of the software. So the experimental network for the single controller SDN network consisted of a single OvS switch, one SDN controller, and two host endpoints; whereas, the experimental network for the multiple controller SDN network consisted of a single OvS switch, two SDN controllers, and two host endpoints (Figure 13). For the multiple controller SDN network, we utilized two controllers (Figure 13b), where each controller operated as if it was the only controller within the network. And since the controllers were unaware of each other, the load balancing algorithm was implemented within the OvS switch. Two controllers were used within the multiple controller SDN network because it is within Heller et al's [22] recommended number of controllers, that should be operating on a multiple controller SDN network. It is also worth mentioning the control experiment environment (i.e. single controller SDN network) treated all mismatched packets (i.e. the ARP protocol) equally.

The experimental environment contained two host endpoints because each host has a specific task to perform. The attacking host conducted the controller side-channel attack. The target host was unaware of the attack that was performed. Therefore, the target host transmitted the default OS's response to the attacking host's spoofed packets. These experiments were conducted within a virtual environment, so it is important to ensure that the attacker and third virtual machine's bridge adapter have promiscuous mode enabled. Otherwise, these hosts would not have been able to send and receive spoofed packets.

4.3.3 Multiple Controllers with the Default OvS Switch

The OvS switch by default does not support a load balancing algorithm for the control plane, as previously mentioned, even though the default implementation of the OvS switch allows an administrator to configure a multiple controller setup. A multiple controller SDN network with the default implementation of the OvS switch is designed to transmit a broadcast PACKET_IN message to all connected controllers when the switch comes across a mismatched packet. Therefore, when the SDN network is a multiple controller SDN network, the OvS switch will send duplicate forwarding requests to all connected controllers, and each controller will respond to its PACKET_IN message by transmitting a new forwarding rule and respective PACKET_OUT message to the switch. Even though all controllers will send forwarding rules to the switch, the switch's cache will only have a single forwarding rule for the mismatched packet, which could be due to the duplicate forwarding rules overwriting each other because they contain the same information. But since a single mismatched packet has duplicate PACKET_OUT messages, the destination host will receive and respond to duplicate packets (Figure 14). The number of duplicated packets will equal the number of controllers that are connected to the OvS switch. This vulnerability can be used as the basis for various attacks, including:

1. Determining if the default OvS switch is operating within the SDN network;
2. Determining the number of controllers that are operating within the SDN network; and
3. Performing an intentional or unintentional amplified DoS attack.

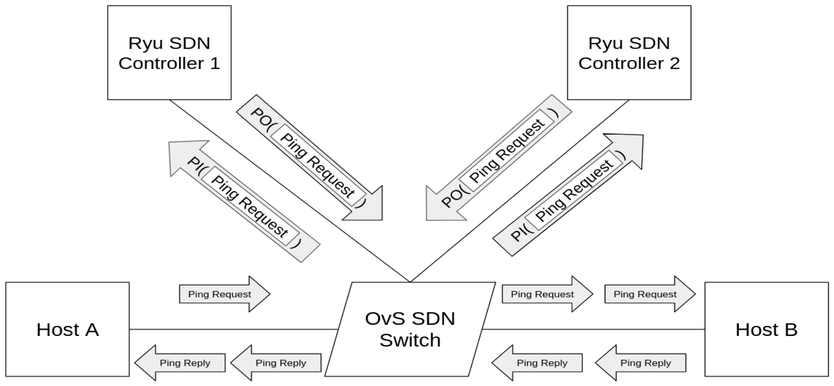


Figure 14: Magnified DoS Attack

4.3.3.1 Multiple Controllers with the OvS Switch Supporting Load Balancing Algorithms

In order to avoid unnecessary controller congestion, and have the SDN network resemble a network infrastructure that utilizes a redundant system, the OvS switch needs to be modified so that it sends a single forwarding request to an individual controller. Therefore, we implemented the Round Robin and protocol based load balancing algorithms (Figures 4 and 3), that are described in Section 2.2.1, in the OvS switch's source code.

For the Round Robin load balancing algorithm, we implemented what was detailed in Section 2.2.1.1 so all mismatched packets (i.e. the ARP protocol) were treated equally. As for the protocol based load balancing algorithm, the secondary controller was designated to install forwarding rules for the ICMP protocol, and the default controller was designated to install forwarding rules for all of the other protocols (such as ARP, TCP, UDP, and SSH). In other words, the secondary controller would obtain mismatched packets with the ICMP protocol, and the default controller obtained mismatched packets for all of the other protocols. As mentioned in Section 4.2.2, the secondary controller obtained one tenth of the amount of request packets transmitted onto the network except when only two request packets were sent onto the network; in this specific case, the secondary and default controller both obtained a single request packet.

4.3.4 Defense

The experimental environments will only subtly change from the details we provided above. The only change to the environments is the network dropping mismatched packets after its corresponding forwarding rule is installed onto the switch. The controller will comply with and support the OpenFlow protocol's buffer option for PACKET_IN messages. Therefore, the controller is configured to send a PACKET_OUT message to tell the switch to drop the mismatched packet instead of forwarding it to the victim host. Section 4.1.3 details and displays this change.

4.4 Results

4.4.1 Overview

We conducted 51 experiments for each environment (single controller, two controllers with the Round Robin load balancing algorithm, and two controllers with the protocol based load balancing algorithm) to obtain three different datasets. Each experiment had the controller side-channel attack transmit between 1 to 1000 (1, 20, 40, ... , 1000 with increments of 20) request packets. For example, the first experiment had the the controller side-channel attack transmit 1 request packet onto the network, the second experiment had the controller side-channel attack transmit 20 request packets onto the network, the third experiment had the controller side-channel attack transmit 40 request packets onto the network, until we got the the fifty first experiment that had the controller side-channel attack transmit 1000 request packets onto the network. We use the terms such as "1000 request experiment" or "experiment 460" within this dissertation to reference the response times of an experiment; in this case, "1000 request experiment" is referencing the experiment that had the controller side-channel attack transmit 1000 request packets onto the network and "experiment 460" is referencing the experiment that had the controller side-channel attack transmit 460 request packets onto the network.

The control plane was the only difference between the three environments. The first environment consisted of the default implementation of the OvS switch, which contained a single controller (Figure 13a), in order to get the control experiment's response times for the controller side-channel attack. The second environment consisted of two controllers within the control plane (Figure 13b), where the OvS switch supported the Round Robin load balancing algorithm. This environment allowed us to determine if multiple controllers within the control plane would increase the control plane's efficiency; and if the efficiency is increased, this environment allowed us to determine by how much. Furthermore, this efficiency increase states how many controllers are within the control plane. The third environment also consisted of two controllers within the control plane (Figure 13b), where the OvS switch supported the protocol based load balancing algorithm. Similar to the second environment, this environment allowed us to determine if the control plane's efficiency would increase; and if so, then by how much. Although different from the second environment, the third environment contained a control plane with controllers that handled pre-defined dataset(s) (i.e. protocols), which means each controller is configured to increase the efficiency for specific protocols, where this targeted efficiency increase in effect states how many controllers are within the control plane.

We determined it is imperative for the controller side-channel attack to have the NIC (i.e. kernel) state each packet's timestamp in order to obtain accurate results [32]. Otherwise, if the attacking program manually obtains a timestamp before and after a packet's transmission and reception (i.e. within the OS's user space), an accurate timestamp cannot be guaranteed because the OS's scheduler might not allow the immediate sequential execution of both the timestamp's procurement and packet's transmission or reception, as our results will show the implication of the OS scheduler's influence. Additionally, this allows the attacking program to use the packet's timestamps from the network's perspective by using the NIC's timestamp.

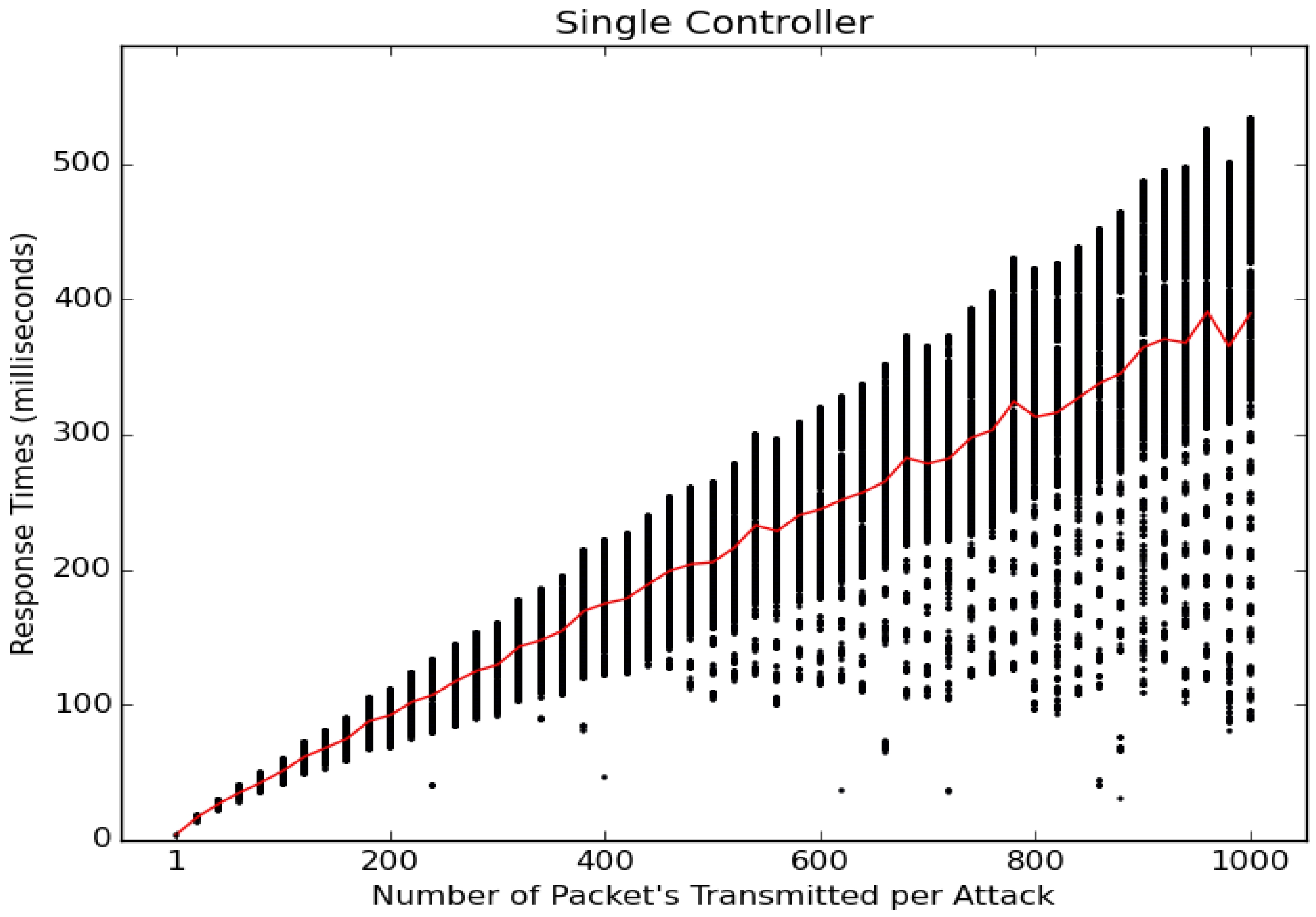


Figure 15: Response Times for a Single Controller SDN Network (Control Experiment Results). The packets' response times increased in direct relation to the intensity of the controller side-channel attack, which is analogous to a controller congestion attack.

Figure 15's x-axis states how many packets the controller side-channel attack dumped onto the network for each experiment in the single controller environment (ex: 1000 request packets were sent by the attacking host). Figure 15's y-axis represents the response times in milliseconds. The individual dots for each experiment represent the individual request's response time, so we can review how long it took each request to get a reply. Lastly, the red line shows the average response time for all of the requests transmitted during each conducted experiment.

As with most First-In-First-Out (FIFO) services, the more objects a FIFO resource has to process, the more time it takes the resource to fully gather and process the final object. This is no different for an SDN network, where the FIFO resource is the controller(s). As seen in Figure 15, the packets' response times increase based on the severity of the controller side-channel attack. The more packet's the controller side-channel attack outputted onto the network corresponded to an increased strain on the controller, so it took the controller more time to process the final request packet because the controller is a FIFO resource. This successfully completes the goal of the first experiment because the first experiment's purposes was to determine how a bare-bones SDN network would react to the controller side-channel attack. So to conclude, at an SDN networks core, a higher packet response time is obtained when more mismatched packets are dumped onto the network.

Interestingly, there were some packets with a response time noticeably lower than the average response time. For example, the 1000 ARP request experiment had some packets with a response time of approximately 103 milliseconds, when the average response time was 389.8 milliseconds. However, the packets with a low response time were only a small subset of packets that had a low response time because the overall average response time kept increasing linearly with each experiment. The reason these packets had a lower response time could be due to either these packets were the first packets sent during the attack and therefore were processed faster because the controller was not congested, or the OS scheduler happened to give the controller application execution time immediately after these packets were transmitted. The reason the average response time kept increasing with each experiment is due to each experiment's increase in number of requests transmitted by the controller side-channel attack. This in affect gave the controller (i.e. the FIFO resource) more objects to process, so it took more time to obtain a response time for the trailing mismatched packets, which increased the overall average response time as a direct effect. As we will discuss further during the other environments' results, this figure indicates a solution to the underlining principle of whether the hardware or the OS scheduler is the bottleneck for the control plane's efficiency.

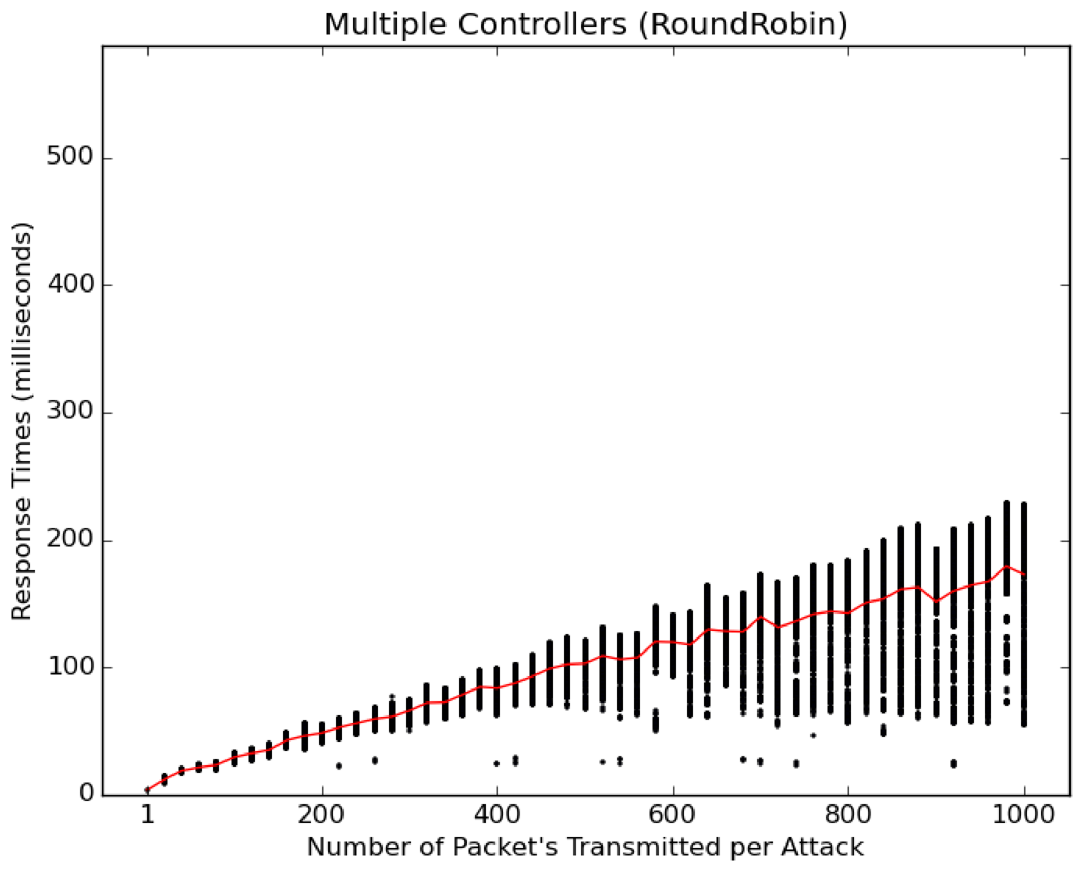
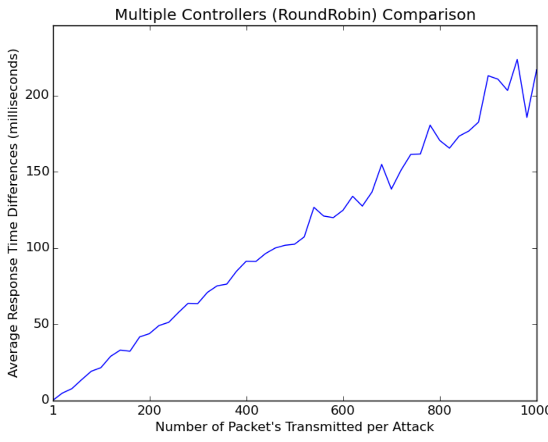


Figure 16: Response Times for a Multiple Controller SDN Network with the Round Robin Load Balancing Algorithm. The average packets' response times are approximately half the amount of the control experiment, which can be used to state the control plane contains two controllers.

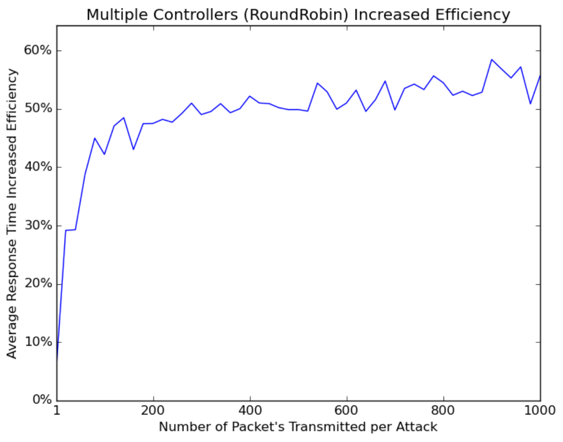
4.4.2 Round Robin Load Balancing Algorithm

Similar to the control experiment's results, the packets' response times linearly increase based on the severity of the controller side-channel attack (Figure 16). There were also packets with a lower response time than the average response time. These lower response times were the primary reason the average response time decreased. We can use Figure 16 to see that a relative portion of response times were lower than the average response time. The reason for these lower response times could be due to either these packets being processed when the control plane was not congested, or the OS scheduler gave the control plane a relative amount of execution time such that these packets were processed in a timely manner.

This multiple controller environment's results supports our earlier statement about the FIFO resource (i.e. the control plane) and an SDN's core architecture design produces higher response times when there are multiple mismatched packets processed in quick succession. This environment's results clearly shows the Round Robin load balancing algorithm increase the control plane's efficiency. A vast majority of the individual, and practically all of average, response times are noticeably lower in the multiple controller environment, as seen when comparing the response times in Figures 15 and 16. We can see the average response times were reduced by about half of the control experiment's average response times for practically all of the experiments. Now that the control plane's improved efficiency has been established, it is important to determine the amount of improvement that corresponds to the utilization of the Round Robin load balancing algorithm.



(a) Control Plane's Improved Response Time. The difference in average packets' response times per experiment linearly increases with the increased severity of the controller side-channel attack. Interestingly, this linear increase almost follows the equation $f(x) = 1/4 * x$, where x is the number of packets transmitted during the attack.



(b) Control Plane's Improved Efficiency. The efficiency in average packets' response times per experiment exponentially increases and then quickly plateaus as the controller side-channel attack's severity increased. This plateau clearly states the control plane increased its efficiency by approximately 50% by using the Round Robin load balancing algorithm.

Figure 17: Control Experiment vs Round Robin Average Response Times and Efficiency Improvement Comparisons. Subfigure 17a displays the control plane's improved response time, where the improved response time is calculated as the $(AverageControlExperimentRTT - AverageRoundRobinRTT)$ per experiment. Subfigure 17b displays the control plane's improved efficiency by using the Round Robin load balancing algorithm, where the improvement is the $((AverageControlExperimentRTT - AverageRoundRobinRTT)/AverageControlExperimentRTT) * 100$ per experiment.

We compared the single controller environment's response times against the Round Robin environment's response times in order to determine the number of controllers operating within the SDN network. When comparing the control experiment (i.e. single controller environment) response time to the response times obtained within the Round Robin environment, we can see how the addition of the second controller and the Round Robin load balancing algorithm increased the control plane's efficiency (Figure 17). A simple difference evaluation of the control experiment's and Round Robin's average response time per experiment $(AverageControlExperimentRTT - AverageRoundRobinRTT)$ clearly displays a linear differential increase between the two environments (Figure 17a). The difference in average response times consistently grew along with the severity of the controller side-channel attack, where severity corresponds to the number of requests transmitted onto the network. This type of difference is the desired result of implementing a load balancing algorithm in a pivotal service such as the control plane; and, Figure 17a shows how the control plane can benefit from these load balancing techniques, especially when it has to process an over-abundant amount of mismatched packets. Interestingly, the linear increase within Figure 17a almost follows the equation $f(x) = 1/4 * x$, where x is the number of packets transmitted during an experiment.

More important than calculating the response time difference is determining the percentage of efficiency improvement for each experiment (Figure 17b). Figure 17b's y-axis is different than the three previous figures. Figure 17b's y-axis represents each experiment's average response time efficiency improvement in terms of percentage. Each experiment's improved efficiency was calculated as follows $((AverageControlExperimentRTT - AverageRoundRobinRTT)/AverageControlExperimentRTT) * 100$. For example, we took the single controller environment's average response time when the controller side-channel attack transmitted 1 request packet, subtracted that number by the Round Robin environment's average response time when the controller side-channel attack transmitted 1 request packet, then took that difference to divide by the single controller environment's average response time when the controller side-channel attack transmitted 1 request packet, and multiplied by 100 in order to get a value in the form of a percent. Furthermore, we took the single controller environment's average response time when the controller side-channel attack transmitted 20 request packets, subtracted that number by the Round Robin environment's average response time when the controller side-channel attack transmitted 20 request packets, then took that difference to divide by the single controller environment's average response time when the controller side-channel attack transmitted 20 request packets, and multiplied by 100 in order to get a value in the form of a percent. This calculation was performed with each experiment's (1, 20, 40, ..., 1000) corresponding average response time to obtain Figure 17b.

These results (Figure 17b) show a sharp exponential increase of improvement for the control plane's efficiency; and then, the improvement of the control plane's efficiency instantaneously plateau at about 50%. More specifically, it took 120 mismatched packets to obtain an efficiency improvement of 47%. This is a desired amount of improved efficiency when implementing two controllers and the Round Robin load balancing algorithm in the control plane.

The first couple of experiments display how the control experiment's and Round Robin's efficiency were close to identical due to the single controller being un-congested and its ability to process the mismatched packets at a rate similar to the duel controller environment. In other words, the control plane won't benefit from a two controllers setup until a certain mismatched packet threshold is meet, the threshold being 120 mismatched packets for this environment. It did not take long for the single controller to get congested and process the mismatched packets at a slower rate than the duel controller environment. The remainder of the experiments display how the control experiment's and Round Robin's efficiency were significantly different due to the congestion of the single controller and the single controller's limited execution time provided by the OS scheduler. The SDN network benefited from the duel controller environment for these experiments because each controller only processed half of all of the mismatched packets; whereas, the single controller environment had the one controller process all of the mismatched packets. In other words, the control plane in the duel controller environment was not as congested as the control plane in the single controller environment. The control plane's efficiency doubled by utilizing two controllers and the Round Robin load balancing algorithm in the control plane as Figure 17b states by displaying the control plane's improved efficiency percentage of 50%.

To conclude, the load balancing technique within the control plane produced the desired result of reducing the control plane's congestion. The results obtained within the Round Robin environment had similar characteristics to the results obtained within the single controller environment, which makes the analysis of the average response time important. The Round Robin environment's response times are noticeably lower than the control experiment response times, which can be used by an adversary to determine the number of controllers within an SDN network. It does not take an over-abundant amount of mismatched packets to determine the number of controllers within an SDN network; to be precise, it takes about 120 mismatched packets in our virtual environment. And finally, the controller side-channel attack is successful because the results show:

1. The SDN network was a multiple controller SDN network by detecting a lower average response time within the Round Robin's environment than the control experiment's environment (Figure 16);

2. The Round Robin load balancing algorithm has been utilized by detecting a linear differential increase between the two environments (Figure 17a); and

3. Two controllers were operating within the SDN network by detecting an efficiency improvement of 50% (Figure 17b).

4.4.3 Protocol Based Load Balancing Algorithm

As mentioned in Section 4.2.2, both the ARP and ICMP protocols were used in experiments for the protocol based load balancing algorithm. Similar to the control experiment's results, there was a linear increase in the packets' response times, which were processed by the default controller, based on the number of packets transmitted by controller side-channel attack (Figure 18). Furthermore, we can use the default and secondary controllers' average response times, which are displayed in Figure 18, to build upon our earlier statement that the control plane is a FIFO resource and an SDN network produces higher response times when there are multiple mismatched packets processed in quick succession. Figure 18 shows us that the control plane as a whole is not a single FIFO resource; in other words, the concept of a single control plane does not mean multiple controllers within the control plane are treated as one controller. Rather, each controller within the control plane is its own FIFO resource; moreover, each controller is sharing execution time for the hardware (CPU, memory, etc.).

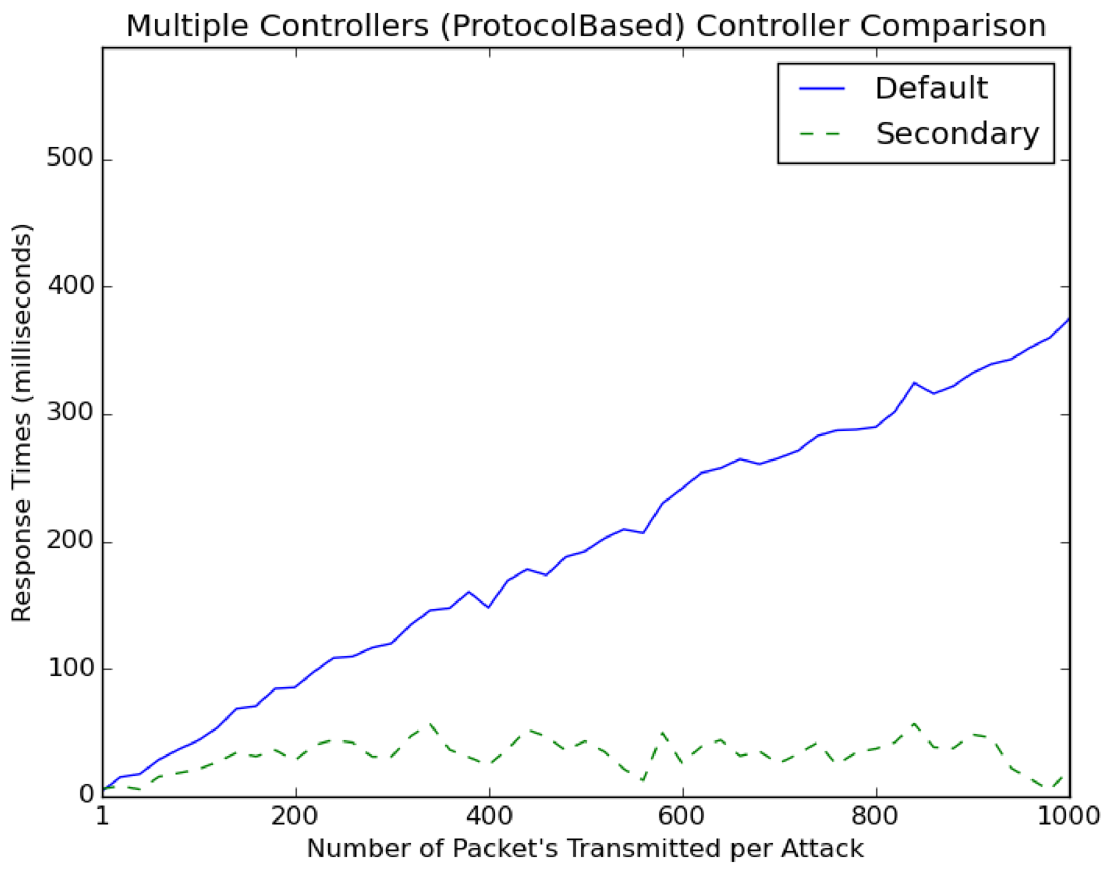


Figure 18: Response Times for a Multiple Controller SDN Network with the Protocol Based Load Balancing Algorithm. The average packets' response times for ARP packets are significantly higher than the ICMP packets, which can be used to state the control plane contains two controllers.

Figure 18's x-axis and y-axis represent the same metrics as our control experiment's figure (Figure 15). Although, this figure's focus is displaying each individual controller's congestion by showing both response time averages for the two controllers. This environment required the controller side-channel attack to simultaneously transmit packets with two different protocols, so this figure shows the average response time for the two protocols utilized. The blue line shows the average response time for all of the ARP requests processed by the default controller during each conducted experiment. The green dashed line shows the average response time for all of the ICMP requests processed by the secondary controller during each conducted experiment.

Similar to our two previous results, the average response time increased linearly based on the severity of the controller side-channel attack for the congested (i.e. default) controller. But the un-congested (i.e. secondary) controller had a consistently low average response time (Figure 18). The SDN network benefited from the protocol based load balancing algorithm because it was able to prioritize targeted connections with the un-congested controller. In other words, the single controller environment's control plane cannot prioritize important connections so these connections are limited and influenced by the congestion of the controller. Comparing the average response time from our control experiment (Figure 15) to the average response time from the protocol based environment's congested controller shows they relatively have the same values. The reason is that the protocol based environment's congested controller processed practically the same amount (90%) of mismatched packets as the control experiment's controller, which indicates the controller application had approximately the same amount of execution time in both environments.

Interestingly, the protocol based environment's un-congested controller's average response time varied between 4 and 57 milliseconds for all of the experiments. And comparing this un-congested controller's average response time against the control experiment's average response time reveals the secondary controller is almost always within range of the control experiment's average response times which had a response time between 4 and 52 milliseconds, when 1 to 100 mismatched packets were dumped onto the network. This occurs because the un-congested controller only processed 10% (i.e. between 1 to 100 mismatched packets) of all the mismatched packets emitted by the controller side-channel attack for each experiment. The reason the secondary controller's average response time varied is because the controller side-channel attack did not dump all of the ICMP packets onto the network in quick succession. So, this allowed the OS scheduler to manage each component's (VMs, controller application, and OvS switch) execution time in such way that the mismatched packet either was immediately processed by each component (i.e. instant response), which lead to an average response time near 0 seconds, or was eventually processed by each component, which lead to an average response time near 50 milliseconds.

These results also supports our earlier statement about the FIFO resource, where each individual controller within the control plane is its own resource; and, an SDN network will produce higher response times when a multitude of sequential mismatched packets are transmitted within the network. This environment's results clearly shows the protocol based load balancing algorithm increased the control plane's efficiency for certain protocols. More specifically, a popularly used protocol does not have to limit the response time of non-popular protocols because of a congested single controller. As seen within Figure 18, the default controller's (i.e. the ARP protocol's) average response time is higher than the secondary controller's (i.e. the ICMP protocol's) average response time for just about every experiment. Additionally, this average response time difference drastically increased as the controller side-channel attack's severity heavily increased against the default controller while at the same time only slightly increased against the secondary controller, where severity corresponds to the number of requests transmitted onto the network.

This difference in average response times is the desired result of implementing this type of load balancing algorithm, and shows its benefits for the control plane. This difference in average response times with two different protocols dictates the number of controllers operating within the SDN network because one protocol is congesting a controller whereas the second protocol is lightly using the second controller. Figure 18 shows us the number of mismatched packets needed to determine the number of controllers within the SDN network. For example, the controller side-channel attack could use 1000 request packets (900 ARP plus 100 ICMP requests) to determine the number of controllers, because the two protocols had an average response time difference of 354 milliseconds. The controller side-channel attack could use as low as 20 request packets (18 ARP plus 2 ICMP) in order to notice the secondary controller's proficiency over the default controller's (7 millisecond difference). However, we recommend using 80 request packets (72 ARP plus 8 ICMP) or more for comparing the control plane's efficiency because the results will show an un-questionable amount of difference between the two average response times (18 millisecond difference or more with at least an efficiency improvement of 50%).

These experiments show how an excessive amount of mismatched packets paired with the execution time of the OS scheduler lead to the detection of the default controller's congestion, as well as the secondary controller's non-congestion. In other words, the secondary controller's execution time paired with only a few mismatched packets lead to detecting its non-congested state. The controller side-channel attack's flood of mismatched ARP packet requests provided the default controller with a large and successive processing load. The secondary controller was only provided a small and noncontinuous processing load by the controller side-channel attack. The default and secondary controllers were only able to work on their processing load when the OS scheduler provided them execution time. Furthermore, Figure 18 display of the difference in average response times shows how the OS scheduler did not provide the default controller ample execution time to complete its large processing load, when compared to the execution time provided to the secondary controller. If the OS scheduler provided the default controller additional and un-interrupted processing time, then the default controller would have been able to process its mismatched packets faster, which would lead to de-congesting the controller and the default controller's mismatched packets' response times would be similar to the secondary controller's mismatched packets' response times. The secondary controller only had to process a mismatched packet intermittently, so it did not have a substantial processing load like the default controller. Thus, the OS scheduler provided the secondary controller an abundant amount of execution time for the controller to process its mismatched packets. We further discuss the OS scheduler within Section 4.4.4.

To conclude, the utilization of the load balancing algorithm technique within the control plane produced the desired result of reducing the control plane's congestion for specific protocols. Interestingly, the default controller's average response time follows the control experiment's average response time, and the secondary controller's average response time is almost always within range of the control experiment's average response time. This severe difference in the average response times can be used by an adversary to determine the number of controllers within an SDN network due to the noticeable difference between the default and secondary controller's average response time. And similar to the Round Robin environment, it does not take an over-abundant amount of mismatched packets (about 80 mismatched packets) to determine the number of controllers within an SDN network. In conclusion, the controller side-channel attack is successful because the results (Figure 18) show a significant proficiency difference within the control plane, and this difference was established due to the protocols utilized within the packets. More specifically, the difference in the ARP and ICMP average response times state:

1. The protocol based load balancing algorithm was utilized;
2. One controller was designated to handle the ARP protocol while the other controller handled the ICMP protocol; and
3. Two controllers were operating within the SDN network.

4.4.4 The Bottleneck: OS Scheduler vs Hardware

Our experimental results clearly state how the Round Robin and protocol based load balancing algorithms can substantially increase the control plane's efficiency. Just as it was important to determine the amount of improvement these load balancing algorithms had within the control plane, it is important to determine what is the limiting factor that is crafting a mismatched packet's response time. In other words, what is the control plane's bottleneck? Is the bottleneck the system's hardware or is it the OS's scheduler? After analyzing the experiment's results, we have determined the control plane's bottleneck is the OS scheduler. We use the term "control plane execution time" within this dissertation to refer to the amount of execution time with the CPU, memory, motherboard bus, etc. for the controller application(s).

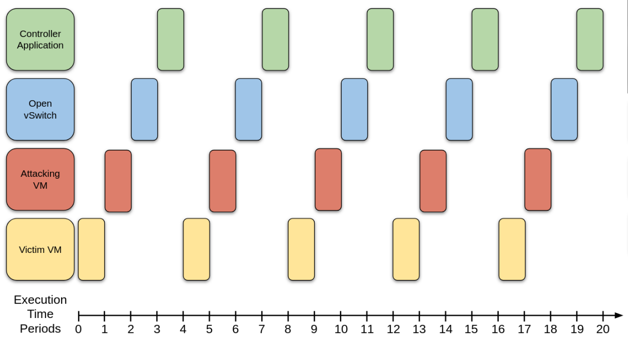
First of all, we determined the control plane was the SDN network's bottleneck for the mismatched packets not the VMs or Open vSwitch. We argue the attacking VM that executed the controller side-channel attack and tcpdump applications, as well as the victim VM that transmitted the OS's default response to an ARP and ICMP request, were handling packets faster than the controller application. While the controller side-channel attack was operating within the OS's user space, it would craft all of the packets used during the attack before transmitting the first packet. This allowed the attacking program to obtain the next sequential packet from a list of crafted packets and then send it. The tcpdump application was specified to save the packets directly to disk in order to optimize its performance. The victim VM just responded to the ARP and ICMP requests with the OS's default response, so this was performed within the kernel space and therefore faster than the controller application. The controller application was operating within the user space that had to process the different OpenFlow messages, alert the right procedure when an OpenFlow flow modification message was obtained, and complete the routing algorithm in order to determine how to forward the packet. All of these requirements are the reasons we argue the controller application was the limiting factor (i.e. bottleneck) for packet handling. These arguments are substantiated by the results obtained within both multiple controller environments because the significant difference between the single and multiple controller environments was the addition of the second controller and utilization of the load balancing algorithms.

We were able to focus on these two factors (OS scheduler and hardware) as the candidates for the bottleneck because practically the same environment was utilized for each experiment. Both of the multiple controller environments show the OS's scheduler is the bottleneck instead of the hardware (i.e. CPU, motherboard bus, memory, etc.), because these environments show a lower average response time. The first hint obtained for the OS scheduler as the bottleneck was the necessity to use the NIC card's timestamp because the OS scheduler sometimes did not sequentially execute the timestamp's procurement and packet emission or reception procedures, which provided inaccurate results with an increased packet's response time. The control experiment's results (i.e. the single controller environment) hinted at the OS scheduler as the bottleneck because of the various independent packet response times lower than the average response time; plus, some experiment's average response time was lower than the previous sequential experiment's average response time. For example, the experiment that sent 980 mismatched packets had a lower average response time than the experiment that sent 960 packets. The independent packet response times do not indicate the hardware as the bottleneck because if the hardware was the bottleneck, we would not expect to obtain such noticeable increases between the individual packet response times, which is predominately seen within experiments 460 to 1000 (ex: experiment 880). If the limiting factor was the hardware, then we would expect experiments 460 to 1000 to have all of the individual response times near each other with a steady increase, just as it is for almost all of the experiments from experiment 1 to 460 (ex: experiment 200). Thus, these increases between the individual packet response times indicate the OS scheduler gave the VMs, controller application, and Open vSwitch an un-optimized amount of execution time or the OS scheduler gave some other component(s) execution time.

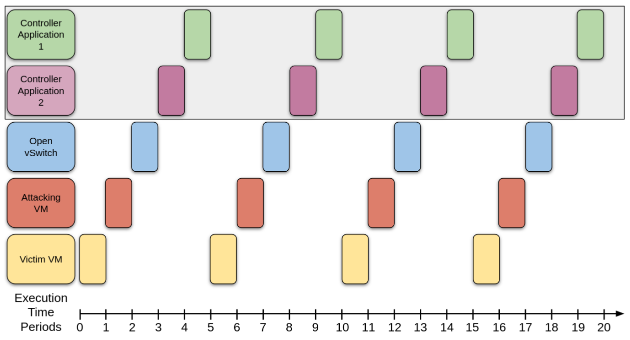
The average response time also does not indicate the hardware as the bottleneck because if it did, then the average response time should not decrease as it did from experiments 960 to 980. In other words, we would expect the average response time to have a steady increase without any noticeable decreases (ex: the increase with experiments 1 to 540) if the limiting factor was the hardware. Therefore, these decreases in the average response time indicate the OS scheduler gave the VMs, controller application, and Open vSwitch components more execution time or an optimized amount of execution time. However, the control experiment's results are only an indicator of the OS scheduler as the bottleneck and are not decisive because an un-congested controller can process mismatched packets faster than a congested controller, which explains why some packets had a response time lower than the average response time. Additionally, these few and subtle decreases within the average response time are a solid indicator of the OS scheduler as the bottleneck, but not a decisive indicator because these decreases could be due to hardware optimizations such as the CPU utilizing another core, the CPU temporarily increasing its clock frequency, or the mismatched packet had un-interrupted access of the motherboard's bus and/or components.

The analysis of the Round Robin multiple controller environment tells us that the inclusion of a second controller and the Round Robin load balancing algorithm improved the SDN network's performance. This performance improvement is a direct result of the OS scheduler providing the control plane more execution time, which allowed more mismatched packets to be processed per control plane execution time period. Therefore, this improvement shows the hardware is not the bottleneck, because if it was the bottleneck then the utilization of the second controller and Round Robin load balancing algorithm would not improve the network's performance. If the hardware was the bottleneck, then the Round Robin environment's response times would either match the control experiment's response times or their response times would be higher.

When analyzing the control plane's execution time, consider only the components utilized during the experiment for sharing the hardware's capabilities: the VMs, controller application(s), and Open vSwitch. Also, consider the OS scheduler provides each component the same amount of execution time per period. So, the single controller environment's execution time per component would be 1/4 of the total execution time, where the four components that all shared the hardware's capabilities and each got 25% of the total execution time were the attacking VM, the victim VM, the one controller application, and the Open vSwitch. Whereas, the Round Robin environment's execution time per component would be 1/5 of the total execution time because of the additional controller application, in other words the five components were the attacking VM, the victim VM, the first controller application, the second controller application, and the Open vSwitch. The control plane's execution time within the single environment was $1/4 = 25\%$ but the Round Robin environment had $1/5 + 1/5 = 40\%$ because the two controller applications each obtained 1/5 of the total execution time, which allowed the execution involving the control plane to almost double.



(a) OS Scheduler Single Controller Environment Example. The control plane (aka the controller application) had a total execution time of 5 execution time periods ($20 * 25\% = 20 * 1/4 = 5$), which is one additional execution time period than the multiple controller environment.



(b) OS Scheduler Multiple Controller Environment Example. Controller application 1 and 2 was both allotted 4 execution time periods, which is one execution time period less than the single controller environment. However, when combining the execution time periods of controller 1 and 2, the control plane had a total execution time of 8 execution time periods ($20 * 40\% = (20 * 1/5) + (20 * 1/5) = 8$).

Figure 19: OS Scheduler Example. Subfigures 19a and 19b displays the total amount of execution time for each component (VMs, controller application(s), and Open vSwitch). The multiple controller environment's control plane, as a whole, had a greater amount of total execution time that shows how the OS scheduler is a limitation for the control plane.

Let's further build upon our example and say it takes 20 execution time periods to complete an experiment. As seen within Figure 19a, the control plane (aka the controller application) for the single controller environment had a total execution time of 5 execution time periods ($20 * 25\% = 20 * 1/4 = 5$), because the hardware was shared between all four components of the experiment. As seen within Figure 19b, the control plane (aka controller application 1 and 2) for the multiple controller environment had a total execution time of 8 execution time periods ($20 * 40\% = (20 * 1/5) + (20 * 1/5) = 8$), because the hardware was shared between all five components of the experiment. When comparing the controller application execution times individually, the controller application for the single controller environment had an additional execution time period when compared to the allotted execution time for either controller applications within the multiple controller environment. However, the control plane for the multiple controller environment as a whole (i.e. considering both execution time periods for controllers 1 and 2) was allotted a higher amount of execution time of 8 execution time periods, where the control plane for the single controller environment was allotted only 5 execution time periods. This is to show how the OS scheduler is a limitation for the control plane and the addition of a second controller application can improve the performance of the control plane.

Additionally, this increase in the control plane's execution time allowed more packets to be processed per time period. When analyzing the amount of mismatched packets processed by the control plane per execution time period, consider a controller application can process a total of two mismatched packets per time period, and the control plane received four mismatched packets at the same time. The single controller environment only has one controller application, so it would be able to process two out of the four mismatched packets within an execution time period and queue the remaining two mismatched packets for the next execution time period. The Round Robin environment has two controller applications, so it would distribute two mismatched packets to both controller applications; therefore, the control plane would be able to process all four mismatched packets per execution time period without queuing any packets for the next execution time period because the first controller would process its two mismatched packets and then the second controller would process its two mismatched packets. This means the multiple controller environment's control plane is able to process more mismatched packets in the same time.

The above key points proving the OS scheduler as the bottleneck is supported by and can really be seen in Figure 18. It shows how the hardware is not the bottleneck because if the hardware was the bottleneck, then the ARP and ICMP average response times would overlap (i.e. equal) each other; because essentially, we would expect the hardware to be backed up with mismatched packet requests such that it does not matter if the mismatched packet is an ARP or ICMP protocol packet. If the hardware was the bottleneck, the mismatched packet would struggle to get processed by the hardware in either protocol's case. Additionally, both the ARP and ICMP average response times would equal or be higher than the control experiment's average response time because both of the environments' control plane processed the same amount of packets. This figure also shows how the OS scheduler is the bottleneck because the ICMP packet's response times are significantly lower than the ARP packet's response times. The OS scheduler as the bottleneck also explains why the ICMP average response time fluctuates. The execution time period per component was not optimized for processing the ICMP packets with every conducted experiment.

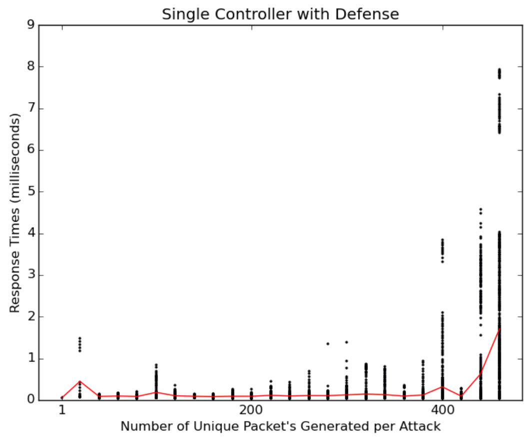
We can see how congesting the default controller reduces its performance. But the secondary controller maintains a steady performance rate. In other words, the ARP packets have a significantly higher response time than the ICMP packets, because the default controller is congested and is not given enough time by the OS scheduler to process all of the ARP requests obtained. Figure 18 states the Open vSwitch can quickly process all of network's packets because there is a difference in a packet's response time solely due to the packet's protocol. Since all of the forwarding rules had the same priority, this difference in average response time was not caused by the Open vSwitch getting congested; in other words, the Open vSwitch is not the limiting factor. And as mentioned above, the VMs were also not the limiting factor because the difference in average response time is in direct relation to the congestion of the mismatched packet's controller application. The difference in average response time is based on the packet's protocol, which means the difference is a direct result of the controller application's individual performance (i.e. total execution time and processing rate per execution time); meaning, the controller application(s) are the limiting factor in the SDN network.

Interestingly, the default controller's average response time is similar to the control experiment's average response time, plus the secondary controller's average response time is almost always within bounds of the control experiment's average response time when referencing experiments 1 to 100, which indicates the hardware's limitations paired with the OS scheduler have been reached.

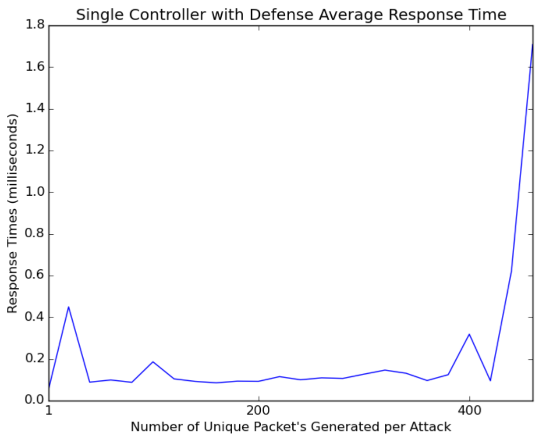
The protocol based load balancing algorithm tailors the control plane's execution time for specified protocols, which is exactly why the ICMP packets' response times are lower than the ARP packets' response times. When analyzing the control plane's execution time and processing capabilities per execution time period, we will use the details mentioned above. This multiple controller environment's control plane had more execution time than the single controller environment, because this environment increased the control plane's execution time from $1/4 = 0.25$ to $1/5 + 1/5 = 0.4$, just like the Round Robin environment. This almost double amount of execution time for the control plane explains why the ICMP packets have a lower response time, but it does not explain why the average response time between the default and secondary controller differs. The protocol based environment differs from the Round Robin environment, because in respect to the packet's protocol, the controller applications had less average execution time than the single controller environment. More specifically, the default controller had $1/5 = 0.2$ of the total execution time to process the mismatched packets with the ARP protocol; whereas, the single controller environment had $1/4 = 0.25$ of the total execution time to process a similar amount of ARP packets. So, the protocol based environment's default controller application (the default controller) had $1/4 - 1/5 = 0.05$ less execution time than the single controller environment to process the ARP mismatched packets, even though as a whole the control plane's execution time increased. This is only a slight decrease in execution time to process the ARP packets, which explains why the default controller's average response time follows the single controller environment's average response time; plus, the protocol based environment's default controller processed 10% less mismatched packets per experiment than the single controller environment's controller application per experiment.

The secondary controller had $1/5 = 0.2$ of the total execution time to process the mismatched packets with the ICMP protocol, which was almost the same amount of execution time the control plane had within the single controller environment. Additionally, the secondary controller only had to process 10% of the total mismatched packets transmitted per experiment; in other words, the amount of mismatched packets processed by the secondary controller per experiment was 10% of the mismatched packets processed by the single controller environment's controller application per experiment. This means the secondary controller had more execution time than the default controller to process its mismatched packets; therefore, it took the secondary controller less time to process the few mismatched packets it received per experiment than the default controller, as seen with the secondary controller's average response time in Figure 18. More specifically, it took fewer execution time periods for the secondary controller to process the mismatched packets when compared to the default controller, where the default controller required numerous execution time periods in order to process the over abundant amount of mismatched packets it received. For example, experiment 1000 had the default controller process 900 mismatched packets while the secondary controller processed 100 mismatched packets. Using the details above, where a controller application can process 2 mismatched packets per execution time period, the secondary controller took 50 execution time periods to process all of its mismatched packets, while the default controller required 480 execution time periods. So, the secondary controller was idle while the default controller was processing an abundant amount of mismatched packets. This explains why the average response time between the default and secondary controller differs and supports our analysis that the OS scheduler is the bottleneck.

4.4.5 Defense



(a) Individual and Average Response Times. The individual and average response times remained consistent and steady when less than 400 unique request packets were utilized. When 400 or more unique request packets were utilized the individual and average response times started to severely increase, except for experiment 420.



(b) Average Response Times. The average packets' response times per experiment was relatively constantly low when less than 400 unique request packets were utilized by the controller side-channel attack. This changed when the controller side-channel attack used more than 400 unique request packets (except for experiment 420), because the average response time was linearly increasing.

Figure 20: Individual and Average Response Times with Defensive Technique. Subfigure 20a displays the individual and average response times per experiment. Subfigure 20b displays just the average response time per experiment in order to easily analyze the results.

Figure 20's x-axis represents the total number of requests packets generated per experiment with the single controller environment. To be more precise, these figure's x-axis is different from the previous figures' x-axis because the adversary had to dump the total number of request packets multiple times per experiment. Figure 20's y-axis represents the response times in milliseconds when a request was successfully able to obtain a reply. The individual dots for each experiment represent the individual request's response time, so we can review how long it took each request to get a reply; whereas, the red line states the average response time for all of the requests transmitted during each conducted experiment (Figure 20a). Figure 20b is a graph of just the average response time, which is within Figure 20a, so that we can easily visually analyze the average response time.

As seen in Figure 20, the packets' response times remained low until the controller side-channel attack was utilizing 400 or more unique packet requests. These low response times display the successful operation of the defense (Section 4.2.3) because the low response times show just the switch's lookup time (t_{lookup}). When the controller side-channel attack attempt to utilize 480 or more unique request packets, it took a significant amount of time to perform the attack and filter the results in order to determine that the attack was not able to obtain 480 or more unique reply packets. These attempts prove that the defense has completed its goal by making an SDN network more resistant to this type of attack (reconnaissance based attack), because it took the adversary more resources to obtain information similar to or less than our control experiment's attack on the SDN environment.

As seen within Figure 20a, we can visually see how the individual packet response times were within range of each other, for the most part. Some of the experiment's packet response times were higher than the other experiment's packet response times (e.g. the experiment with 20 unique request packets). This is especially seen within experiment 460 and further supports our earlier conclusion that the OS scheduler is the SDN network's bottleneck and the control plane is a FIFO resource.

This defensive technique drastically changed the individual and average response times when compared to our control experiment (Figure 15). When the controller side-channel attack was flooding the network with less than 400 unique request packets, there were consistent and steady response times. The response times started to increase when the controller side-channel attack was using 400 or more unique request packets, which is also when the average response time started to linearly increase (except for experiment 420). Even though the average response time began to indicate that the control plane was becoming congested, the controller side-channel attack was also starting to reach a limit of resource utilization in terms of optimized information gathering.

Furthermore, when comparing Figures 20a and 15, we can see how the individual and average response times are significantly lower while the SDN network is operating the defense. The highest individual response time was when the controller side-channel attack was utilizing 460 unique request packets, which had a response time near 8 milliseconds. On the other hand, our control experiment's highest individual response time when the controller side-channel attack was utilizing 460 unique request packets was well above 200 milliseconds. Similarly, the defense's highest average response time was a little less than 1.8 milliseconds;

whereas, the control experiment's average response time at experiment 460 was somewhat less than 200 milliseconds. All of this supports our earlier statement that this defensive technique is successful at deterring an adversary from obtain information about the SDN network's infrastructure.

4.5 Conclusion

The results show an adversary can use the controller side-channel attack to determine the number of controllers within an SDN network. The controller side-channel attack ascertained the control plane's within the multiple controller SDN networks had two controller applications. Two controller applications were determined for the Round Robin environment because the control plane's efficiency improved about 50% when compared to the control experiment. Two controller applications were determined for the protocol based environment because the control plane had a significant difference in processing time depending on the packet's protocol.

Additionally, the controller side-channel attack can be used to determine the load balancing algorithm utilized within the control plane by performing a simple differential analysis with the control experiment's results. If there is a large and linear increasing difference between the multiple controller and control experiment environment's average response times, the Round Robin load balancing algorithm was utilized. But if there are two significantly different average response times due to the packet's protocol or there is a minimal difference between the average response times of a controller within the multiple controller environment and the controller within the control experiment environment (i.e. the default controller's average response time equals the single controller environment's average response time), the protocol based load balancing algorithm was utilized. We have determine it takes as few as 120 mismatched packets for the Round Robin load balancing algorithm and 80 mismatched packets for the protocol based load balancing algorithm in order to determine the above information, by analyzing the difference between the various average response times. And while the results show it can take as few as 20 mismatched packets to determine the number of controllers with the protocol based load balancing algorithm, we recommend using 80 mismatched packets in order to notice a substantial difference in the two controller application's average response time.

The controller side-channel attack also determined the OS scheduler is the control plane's (i.e. SDN network's) bottleneck instead of the system's hardware. Throughout these experiments practically the same environment was utilized, except the addition of the second controller and load balancing algorithms. Plus, the SDN network's performance improved in both multiple controller environments as seen with their lower average response times. These two details paired together identify the OS scheduler as the bottleneck.

Interestingly, the single controller environment's and protocol based environment's average response times indicate the hardware's limitations paired with the OS scheduler have been reached.

An effective defense against the controller side-channel attack, or any reconnaissance attack type, is to essentially hide the $t_{controller}$ time by preventing the controller from forwarding mismatched packets for the SDN network. To be precise, after the controller installs a new corresponding forwarding rule for a mismatched packet onto the switch, the mismatched packet is discarded instead of being forwarded to its destination by a PACKET_OUT message. Our results showed how the packets' response times were only crafted with the switch's lookup time (t_{lookup}) because the average response time was continuously low for the most part. The packets' response times started to increase linearly when the controller side-channel attack utilized 400 or more unique request packets; moreover, this linear increase in response times indicated that the control plane was becoming congested. However, the defensive technique is successful because it is ensuring an adversary has to use more resources in order to obtain the same or less information. For example, the adversary had to flood the environment with the request packets multiple times before they can obtain a reply, and the control experiment's environment had a high response time well above 200 milliseconds; whereas, the environment with the defensive technique had a high response time near 8 milliseconds.

4.6 Future Work

There are various ways to build upon this research. First and foremost, there is the research topic of developing defenses against the controller side-channel attack. There is also the topic of developing run-time algorithms for optimizing virtual SDN networks in order to improve its performance. Additionally, there is the research topic of performing and potentially expanding this side-channel attack methodology against environments other than an SDN network, such as mapping a Content Delivery Network (CDN) infrastructure, determine the number of servers in a cluster, determine if a routing protocol adjusts based on the load of a link, and determine the number of processors in a CPU.

Potential defenses against this attack include intentionally decreasing or increasing the control plane's performance. This can be done by configuring the Open vSwitch to intentionally increase a mismatched packet's response time so that the response time equals the control experiment's response time (i.e. a single controller environment). However, intentionally increasing all mismatched packet's response times would negate the benefits of a multiple controller environment, so the network would need to detect and apply this potential defense against only malicious packets in order to benefit from a multiple controller environment. Another potential defense is intentionally decreasing a mismatched packet's response time so that the results state the multiple controller environment has more controllers than there really are. The OS could increase the controller application(s) priority in order to give the control plane earlier and/or more execution time periods, or the OS could dedicate the controller application(s) to a CPU core by changing the affinity. Doing so might allow the control plane to generate response times of a multiple controller environment that has more controller applications than the real environment. Further research is required in order to determine if this is possible, and if so, what is the performance impact?

The Open vSwitch can also be configured to randomly decrease and increase its mismatched packets' response times such that the average response time widely fluctuates. This can potentially prevent the controller side-channel attack from determining the number of controllers because the response times would be inconsistent even when compared against the control experiment. Unfortunately, this defense could significantly reduce individual connections' performance as a side effect.

There are additional methods for improving the performance of virtual SDN networks other than a multiple controller environment. A virtual network's performance is also based on how well the OS scheduler manages the components' execution time. So, the modification of the OS scheduler has the potential to increase a virtual SDN network's efficiency for the purpose of decreasing a packet's response time. These modification can be static such as increasing the priority of critical components like the controller application. Or dynamic for a more complex approach, the OS scheduler can potentially use advance features to prioritize components while its executing based on the run-time context. This could allow the OS scheduler to essentially follow connections so that each component is provided an appropriate amount of time to process the individual aspects of the connection. For SDN networks transmitting more than one connection, the dynamic OS scheduler will need to know which connections to prioritize in order to know the components to prioritize. Or, the network's performance could potentially improve if the OS implements an entirely different OS scheduler. Various OS schedulers will need to be experimented upon in order to determine which one pairs best with the virtual network.

Additionally, this methodology could be expanded for various different scenarios. For example, this methodology could be expanded to determine the number of servers running in a cluster, determine the number of servers in a Content Delivery Network (CDN) infrastructure or map each CDN server's relative tier in the CDN infrastructure, determine whether a routing protocol adjusts routing based on the load of a link, and identify the number of processors within a CPU. Services that have a high number of clients, such as search engines, benefit from being distributed over a cluster of servers, with the workload balanced between all of the servers within the cluster. The service could also benefit from being physically distributed across multiple locations so that it is in close proximity to the clients. The CDN service's total load will be distributed based on the region so that the root server's load is alleviated, thus decreasing the service's response time. A cluster of servers and a CDN both have to use a load balancing algorithm in order to successfully distribute their processing load, just like with an SDN network's control plane. These three types of services have an ever-changing number of sessions and number of clients, and the clients can be physically close or far from the service.

In order to expand the methodology to map a cluster of servers, an adversary can establish numerous connections so that they have a channel to all of the servers within the cluster. Since it is extremely difficult to constantly have all servers with the same workload, some of the adversary's connections will have random performance improvement or degradation. These connection improvements or degradations will produce noticeable response time patterns just like an SDN network's control plane response time patterns for detecting a failover controller. This random performance change is due to the server releasing resources from other client connections that have ended; then, the server allocates the recently freed resources to the adversary's connections. Likewise, the exact opposite occurrence could happen, where the server shares the adversary's congested resources with new client connections. In order for the adversary to obtain reliable results, the adversary should establish a generous amount of connections with the cluster so that they can have more control over the cluster's resources.

In order for the methodology to map a CDN, an adversary can establish multiple connections with the service so that they congest the closest server. Mapping a CDN with this methodology would be most effective during peak operational hours because that allows the adversary to not have to establish an overabundant amount of connections. Eventually the closest server will become over congested, and the load balancing algorithm will initiate the failover to a server that is farther away from any of the new connections. Because the failover server is further away from the client, the packets will take an increased amount of time when they are forwarded through the network. This increase in the new connections' response times will be seen by the adversary, and the increase will indicate that the new connections are established with a different server than the old connections. The adversary can continue establishing multiple connections until they reach the last failover server. A CDN's over congestion to the use of a failover server with an increased response time is comparable to an SDN network's control plane congestion thus increasing its packet's response time. Moreover, there will be a pattern in the packet response time due the congestion in both a CDN's failover server and an SDN network's control plane.

Furthermore, this methodology could be expanded to determine whether a routing protocol adjusts routing based on the load of a link. Currently, our network infrastructure relies on packet switching rather than circuit switching. Therefore, networking devices view incoming data as individual objects rather than viewing incoming data as part of an existing connection. Packet switching allows packets to instantaneously take a new route because a networking device has determined a new and better route, at that specific time. An adversary can take advantage of packet switching's nature by establishing multiple connections that follow the same route. When the adversary establishes multiple connections, sections of the route will become congested, so a networking device will be forced to redirect some packets through a new route. Redirecting the packets will ideally decrease a packet's response time, and the decrease can indicate to the adversary that the routing algorithm considers network congestion as a variable.

Similarly, the methodology could potentially be used outside of an SDN network to determine the number of processors within a CPU. Modern computer architecture allows multiple cores to be within a single CPU, thus allowing the operating system to distribute a computer's processing load among the CPU's cores. An adversary could run a high priority program with an infinite loop to guarantee a single core within the CPU will constantly be utilized. While the infinite loop is running, the adversary executes a terminating program with a normal priority. The terminating program infers the number of cores within the CPU because the terminating program can only execute when there is another core within the CPU. CPUs with multiple cores also means an adversary could use the same method with parallel processing techniques like multithreading or multiprocessing. A process or thread would perform the infinite loop so that the computer is forced to utilize the other cores. While the infinite loop is running, another process or thread will use the terminating program to determine if another core exists. This is repeated with the addition of executing another high priority infinite loop program in parallel until the terminating program no longer terminates, where the number of infinite loop programs executing identifies the number of processors within the CPU.

5 Protocol Obfuscation

5.1 Concept

5.1.1 Overview

In general, in order for two hosts to obfuscate a protocol, they need to fulfill two requirements. First, both hosts need to execute the same or a similar protocol obfuscation program. And second, both hosts need to have all of the dependency packages that are required to execute this obfuscation program (ex: APIs). These requirements exist because both endpoints need to be aware that a protocol is being obfuscated, and they need to know how to read and write the obfuscated data. While these requirements prevent an entity from obfuscating protocols with all available servers on the Internet, one of the obfuscating endpoints can be used as a pivot point in order to access all available servers on the Internet. Even though it is out of scope of this project, the installation of the obfuscation program and its dependency packages could have been due to a social engineering attack, zero-day attack, insider attack, or malware.

However, in order for protocol obfuscation to work as intended, two assumption must be made. First, if the monitoring entity performs deep packet inspection, then the monitoring entity does not know how to classify the network session that is being hidden by the obfuscation procedure. The second assumption is that the network will forward packets that contain obfuscated data within the payload. The monitoring entity does not identify a network session as an obfuscated session, thus the obfuscated protocol remains hidden. The network itself could drop the network session's packets because they are in an un-supported format. For example, the network could require IPv4 addresses in order to forward the packets, so the obfuscation program cannot use the IPv6 header for the Network OSI layer. More specifically, this example requires the obfuscation procedure to support by the IPv4 header.

Based on the two necessary requirements, the hosts performing protocol obfuscation are tunnel entry and/or exit points. Both endpoints feed the target protocol session's data (target data) into the obfuscation program so that the target data can be obfuscated (obfuscated data). As shown in Figure 21, the obfuscation program will determine if the ingress packet is coming from the target protocol session or the obfuscation tunnel. If the packet is coming from the obfuscation tunnel, then the program will de-obfuscate the obfuscated data so that it can forward the target data through the target protocol session. Otherwise if the packet is coming from the target protocol session, then the program obfuscates the target data so that it can forward the obfuscated data through the obfuscation tunnel.

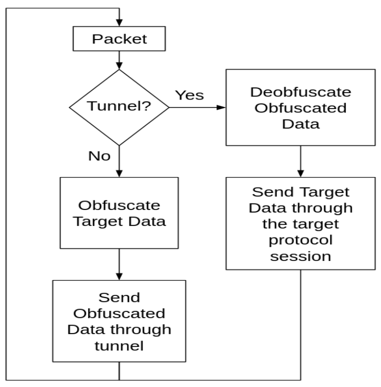
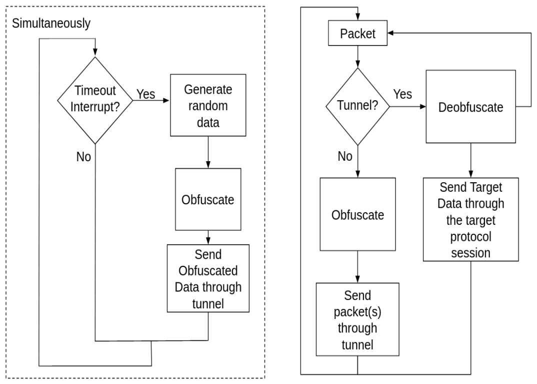

Packet
Tunnel?
Yes
No
Deobfuscate Obfuscated Data
Obfuscate Target Data
Send Target Data through the target protocol session
Send Obfuscated Data through tunnel

Figure 21: A block diagram of the basic obfuscation and deobfuscation steps.

The obfuscation program designed and implemented in this study did not manage, maintain, or modify the target protocol session. Instead, it was responsible for establishing and maintaining connections with the other host that was performing protocol obfuscation. Since the obfuscation program essentially forwards the target protocol session's data, it is possible for this program to obfuscate different types of protocols. Thus, the obfuscation program is similar to a pass-through switch because it just forwards a protocol session's data.

5.1.2 Hide Protocol Characteristics

While protocol obfuscation hides the target protocol session's data and the transport layer's port numbers, it does not hide other protocol characteristics such as inter-packet delay (IPD) and packet size. This implies that if a monitoring entity compares a network session's characteristics to some base case of the target protocol characteristics, then it is possible for the monitoring entity to determine the presence of an obfuscated protocol without looking directly at the target data. This can be a limitation of the obfuscation program, because it is not hiding the protocol session well enough to avoid detection. So in order to better obfuscate protocols, it is necessary for the obfuscation program to periodically transmit random data through the tunnel so that the IPD of the target protocol does not match the IPD of the obfuscated protocol session. Additionally, the obfuscation program will need to ensure the obfuscated and target protocol sessions' packet sizes do not match. This can be performed in a variety of ways including transmitting the packets with a constant packet size. Achieving both of these characteristics mean performing additional steps in the obfuscation and deobfuscation procedure (Figure 22).



Simultaneously
Timeout Interrupt?
Yes
Generate random data
No
Obfuscate
Send Obfuscated Data through tunnel
Packet
Tunnel?
Yes
Deobfuscate
No
Obfuscate
Send Target Data through the target protocol session
Send packet(s) through tunnel

Figure 22: Extending the obfuscation and deobfuscation steps of Figure 21 to hide the (a) IPD and (b) packet size distribution within the obfuscated protocol.

5.1.2.1 IPD Characteristics

In order for the obfuscation program to hide the target protocol session's IPD characteristics, it needs to use a timer to determine when to send random data through the obfuscation tunnel. In Figure 22, you can add (a) and (b) and for this part refer to Figure 22a, while utilizing the obfuscation tunnel, the program is constantly waiting for a timeout interrupt. If the timeout interrupt activates, then the obfuscation program generates some random data that is sent only through the obfuscation tunnel. This feature increases the obfuscation tunnel's transmission rate. If the timeout value is too small, the target protocol session will not remain responsive. Therefore, the obfuscation program should not greatly affect the transmission rate of the target protocol session.

If the obfuscation program is running within a general purpose Operating System (ex: Linux or Windows) instead of a Real-Time Operating System (RTOS), then the obfuscation program will need to prioritize processing the target data or obfuscated data even when the timeout interrupt activates. Since the timeout interrupt could activate when the obfuscation program is processing either the target data or the obfuscated data, the obfuscation program could either prioritize transmitting random data through the obfuscation tunnel, or it could prioritize processing the target or obfuscated data. We decided to prioritize the target or obfuscated data because we want to maximize both network session's goodput, where the random data that does not correspond to the target protocol session is seen as non-useful data. On the other hand, if the obfuscation program running within a RTOS, then the obfuscation program might not have this conflict, depending on the scheduler.

By prioritizing the processing of target or obfuscated data, the obfuscation program will not be able to transmit data through the obfuscation tunnel exactly within the timeout value. We cannot know for certain the exact step that will be interrupted by the timeout. For example, the timeout interrupt could activate at the very beginning, the middle, or the very end of the deobfuscation substep (Figure 23b). However, if the obfuscation substep is interrupted (Figure 23a), then the obfuscation program does not have to worry about ignoring the timeout, because the obfuscation program is already planning on sending data through the tunnel. Since this feature's purpose is to keep the obfuscation tunnel's IPD small, the obfuscation program can achieve this by transmitting either obfuscated data or random data through the obfuscation tunnel.

There is a second method that can be used to control a network session's IPD. Instead of using a timeout interrupt to increase the transmission rate for the obfuscation tunnel, the obfuscation program can use a timeout interrupt to stabilize or reduce the transmission rate. For example, after generating obfuscated data, it is stored in a buffer instead of being instantly transmitted through the obfuscation tunnel. The obfuscation program will transmit the buffered obfuscated data through the obfuscation tunnel when the timeout interrupt activates. Additionally, this feature has the same OS prioritization issue as the increased transmission rate feature, when the buffer is empty. Otherwise, the obfuscation program will prioritize transmitting the buffered obfuscated data over randomly generated data. While dependant on the timeout values, this stabilization/reduction transmission rate feature can be paired with the previously described increased transmission rate feature. By using both transmission rate features in conjunction with each other, the obfuscation program can maintain the obfuscation tunnel's IPD within a specified range or mimic a specified protocol's IPD.

Unfortunately, both of these features can lead to various problems with certain protocols or features. For example, the stabilization/reduction transmission rate feature might not work with time sensitive protocols since the obfuscation tunnel intentionally delays data transmission. Furthermore, depending on the timeout value and constant packet size, it might not be possible to use the increased transmission rate feature with the constant packet size feature, as indicated in Section 5.4.

5.1.2.2 Packet Size Characteristics

In order for the obfuscation program to support transmitting packets at a constant size, the obfuscation procedure (Figure 23a) needs to check if the obfuscated data is larger than the predefined packet size. If so, then the program will need to split the target data before sending it through the tunnel. But if the obfuscated data is smaller than the predefined packet size, then the program will need to append random data to the payload. Additionally, the deobfuscation procedure needs to be modified as well (Figure 23b). Just like with the Basic Obfuscation program (Figure 21), the obfuscation program will need to check if an entire obfuscated data set is obtained before deobfuscating it and then forwarding the target data through the target protocol session. This means if part of the obfuscated data set is missing, then the program has to wait until the remaining part of the obfuscated data set is obtained from the next packet(s). Moreover, after obtaining the full obfuscated data set, the program has to ensure that the target data isn't the random data sent to reduce the Inter-Packet Delay.

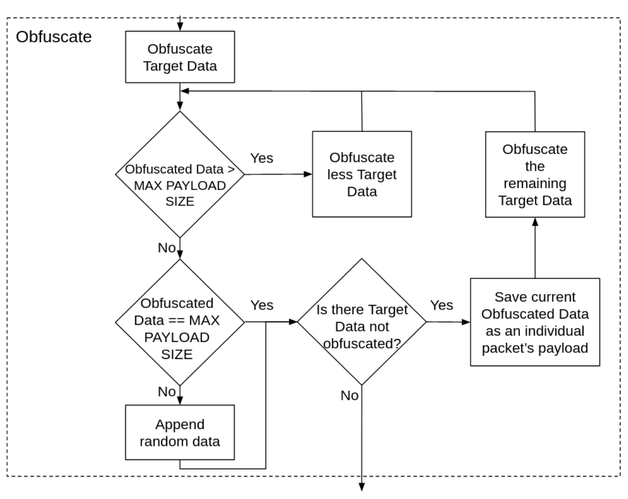
The obfuscation program has multiple options for hiding the packet size characteristics, other than ensuring all of the packets have a constant packet size. The obfuscation program can transmit packets in a wide variety of sizes so that it mimics the packet size characteristics of a specified protocol, which is not the same as the target protocol session. On the other hand, the obfuscation program could transmit packets that follow a specified distribution, where this distribution could equal the network's packet size distribution of the obfuscated data set. Details for these and other obfuscation options can be found within Section 5.5.

5.2 Experimental Design: Protocol Obfuscation with PDFs

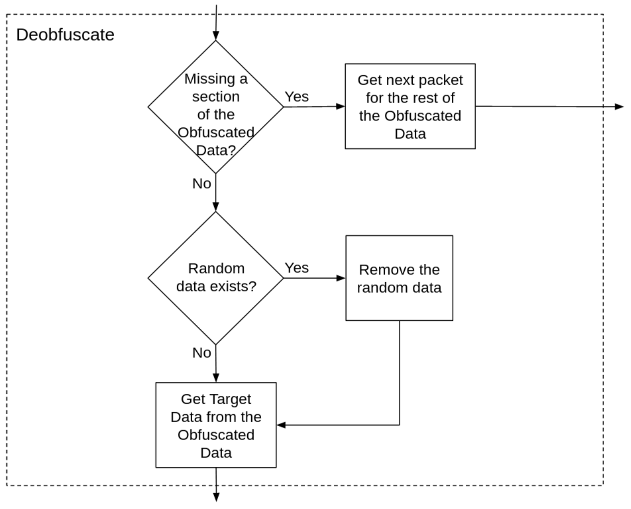
For our experiments, we decided to use PDFs as the cover for obfuscating the SSH protocol. Protocol obfuscation through PDFs (PDF obfuscation) hides a protocol session from a monitoring entity by inserting the target protocol session's data within a PDF file, before transmitting the protocol session onto the network. The target protocol (aka target data) is the SSH protocol and its data, and the cover data is the PDF file.

The obfuscation program utilized in this study obfuscate protocols that use the TCP/IP stack, which means the obfuscation program establishes a TCP session with the client and server service. The primary reason the obfuscation program uses the TCP/IP stack is because TCP sockets do not require administrator privileges. While the obfuscation program can use protocols that are not based on the TCP/IP stack, it would require administrator privileges because the program would need access to raw sockets [43]. Additionally, it would need to know the structure of the protocol that the tunnel is built upon.

As for the target data that is hidden by the obfuscation program, the experiments used the SSH protocol, because its entire network session requires a single connection, it contains a control plane, and it requires


Obfuscate
Obfuscate Target Data
Obfuscated Data > MAX PAYLOAD SIZE
Yes
Obfuscate less Target Data
Obfuscate the remaining Target Data
No
Obfuscated Data == MAX PAYLOAD SIZE
Yes
Is there Target Data not obfuscated?
Yes
Save current Obfuscated Data as an individual packet's payload
No
Append random data
No

(a) Obfuscate Substep


Deobfuscate
Missing a section of the Obfuscated Data?
Yes
Get next packet for the rest of the Obfuscated Data
No
Random data exists?
Yes
Remove the random data
No
Get Target Data from the Obfuscated Data

(b) Deobfuscate Substep

Figure 23: The obfuscate (a) and deobfuscate (b) substeps. The obfuscate substep (a) would first obfuscate the target data, then it ensures the obfuscated data payload is equal to the constant packet size. If the payload's obfuscated data size is less than the constant packet size, then random data is appended to the payload; or, if the payload's obfuscated data size is greater than the constant packet size, then the program obfuscates less of the target data and loops back to make sure the new payload's obfuscated data size is equal to the constant packet size. It could be possible for the target data to be sent over multiple packets. The deobfuscate substep (b) waits to obtain enough payloads whom sum equals the constant packet size. After the sum of payloads reach the constant packet size, the program removes any random data before deobfuscating and forwarding the target data.

the user's input. The SSH protocol has the added benefit of encrypting its data, so we can analyze how the obfuscation program performs when it obfuscates encrypted data. Figure 24 provides the information regarding the TCP ports used in the experiments. Specifically, we used ports above 1023 so that the obfuscation program can be executed without administrator privileges.

Figure 24 also provides a visual overview of the procedure the obfuscation program executes for every packet. The TCP network sessions are established in the following manner: the SSH client initializes and establishes a TCP session with the client obfuscation program, the client obfuscation program initializes and establishes a TCP session with the server obfuscation program, and finally, the server obfuscation program initializes and establishes a TCP session with the SSH server. The SSH data flows in the following manner:

(1) The SSH client transmits its target data to the client obfuscation program because it thinks the client obfuscation program is the SSH server;

(2–3) The client obfuscation program inserts the target data into a PDF file to obfuscate the data;

(4) The client obfuscation program transmits the PDF file to the server obfuscation program;

(5–6) The server obfuscation program extracts the target data that is within the PDF file;

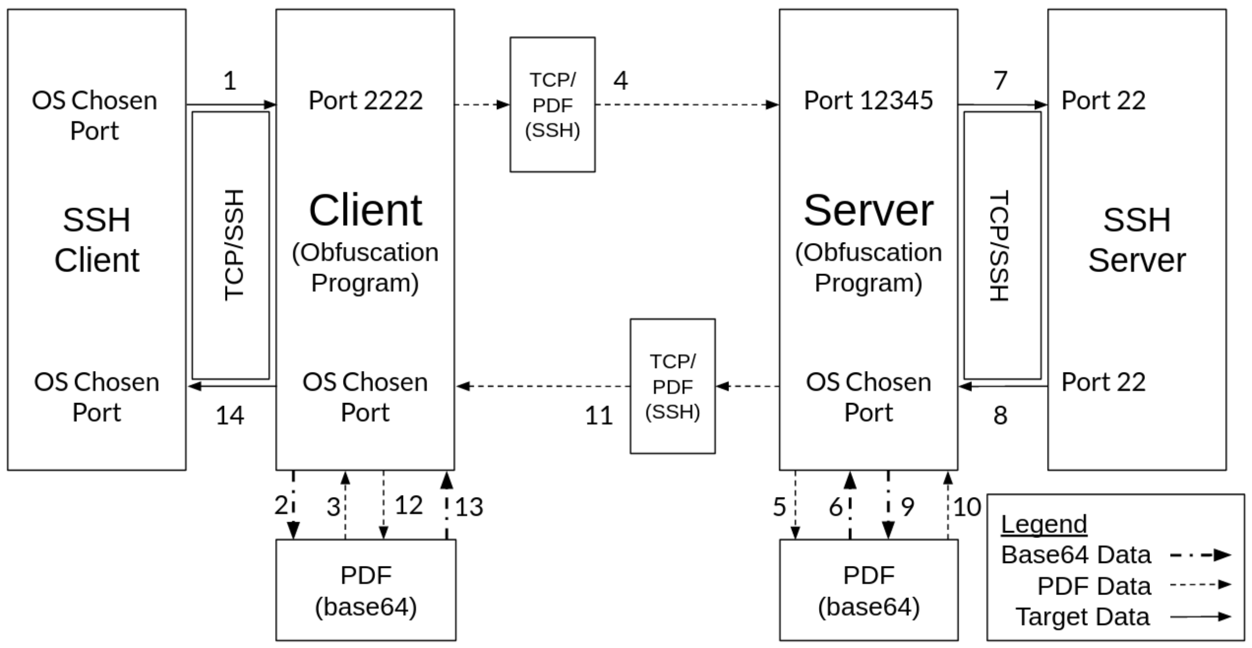


Figure 24: PDF obfuscation steps for both directions and the TCP ports used. The client obfuscation program inserts the SSH client's target data into a PDF file, which is then transmitted to the server obfuscation program. The server obfuscation program reads the target data from the PDF file, and then forwards the target data to the SSH server. This same procedure is performed when transmitting data from the SSH server to the SSH client. And as seen, all of the TCP port numbers are above 1023.

(7) Finally, the server obfuscation program transmits the target data to the SSH server.

This is the same procedure conducted for Steps 8 through 14; however, the SSH data is flowing from the SSH server to the SSH client.

The obfuscation program itself was developed in Python, and it utilized the PyFPDF API to read and write PDF data. As mentioned earlier, the obfuscation program was developed using socket programming; therefore, the OS would manage and maintain the target protocol's and the obfuscation tunnel's lower OSI layer protocols. Since the target protocol session is not managed by the obfuscation program, the obfuscation program was able to focus solely on obfuscating the target protocol session.

After conducing some experiments, we found out the PDF API has a problem reading trailing whitespace from a PDF file. The PDF API had difficulties finding the whitespace if it is the last character within the PDF file. Therefore, we decided to encode the target data into base64 encoding, before it was inserted into a PDF file. This implies that the data in Steps 2, 6, 9, and 13 are in base64 encoding. Since the data in Steps 6 and 13 were still in base64 encoding, the data needed to be decoded before moving onto Steps 7 and 14. While the base64 encoding was used to overcome data lose, it inherently adds another layer of protocol obfuscation that can be used to thwart deep packet inspection.

5.3 Experimental Environment

The experimental study focused primarily on protocol obfuscation and how it can affect the host or the network's view of the host. Therefore, the hosts did not run any additional user processes beyond the obfuscation program, the SSH service, and a packet sniffer such as Wireshark [56]. In order to obtain a comprehensive understanding of the protocol obfuscation's effectiveness, we performed the same experiments within a small (isolated), medium (intranet), and wide area (Internet) network. Additionally, during each experiment, the same commands were executed in the same order, so that the hosts would generate similar network traffic for each experiment. These commands related to three different directory listing and traversal commands, a file output command, a text editor command, and the session termination command. We decided to use the commands listed within Table 1 because they come pre-installed with many Linux distributions and they can be used by an adversary to inject or exfiltrate data from a system.

Sequence/Label	Command
1	ls
2	pwd
3	cd
4	ls
5	cat
6	cat
7	vim
8	logout

Table 1: The sequence of commands executed within each SSH session. The sequence/label numerical value also corresponds to the result's annotation labels.

Since the obfuscation program had its own directory and adjoining files, the three files used within these experiments were all within the obfuscation program's directory tree. This directory was located within the user's home directory. So after SSH'ing into the user's account, the change in directories was to a subdirectory that was one level away from the user's home directory (cd pdfobf/). With respect to the file sizes used in these experiments, we considered three different sizes - large, medium, and small. The first file (FILENAME1) was the largest file used during the experiments and for this we used the obfuscation program itself (cat pdfobf.py). The second file (FILENAME2) was the medium sized file used during the experiments, where this file was the obfuscation program's README file (cat README.md). The last file (FILENAME3) was the smallest file accessed during the experiments, where this file was a configuration file. Furthermore, the contents within the user's home directory differed from host to host; and, the exact path to the user's home directory changed as well, primarily due to the change in username.

5.3.1 Small Isolated Network

The small (isolated) experimental network consisted of two hosts A and B that were connected to each other by a single switch. This isolated network allowed us to confirm that the program operated as intended, as well as determine any system or service delays.

Hardware	**Specification**
CPU	Two Intel Xeon E5-2637 v3 3.5Ghz
RAM	32GB that operated at 2133MHz
NIC	Operated at 5000 MBps

Table 2: The hardware specifications for the two hosts within the isolated network.

Host A had the hardware specifications stated within Table 2; plus, it had 3.7TB of disk space that operated up to 6Gbps, a disk write speed of 500 MBps and read speed of 1.8 GBps [9], and utilized the Ubuntu 16.04.6 64bit OS with kernel version 4.15.0-29-generic. Host B had the same hardware specifications stated within Table 2, but it had 931GB of disk space that operated up to 6Gbps, a disk write speed of 125 MBps and read speed of 1.6 GBps [9], and utilized the Ubuntu 18.04.5 64bit OS with kernel version 4.15.0-128-generic.

5.3.2 Local Area Network

The local area network (LAN or intranet) experimental network consisted of two hosts that were located within an organization's intranet. This internal network allowed us to determine how the obfuscation program operated when two hosts are within the same environment that contains multiple users and are trying to covertly communicate with each other. As for the hosts themselves, the hosts utilized in this experimental environment were host A, except the NIC card's interface was set to operate at 12.5MBps, and host C. Host C had a disk write speed of 94.7 MBps and read speed of 97.2 MBps [9] as well as the following hardware specifications:

Hardware	**Specification**
CPU	Intel Core i7-4790 3.6GHz processor
RAM	16GB that operated at 1600MHz
Disk Space	115GB that operated up to 6Gbps
NIC	Operated at 12.5 MBps

Table 3: The hardware specifications for the host within the Local Area Network (host C).

5.3.3 Wide Area (Internet) Environment

The wide area (Internet) experimental network consisted of two hosts that communicated with each other through the Internet. This large scale network allowed us to determine how the obfuscation program operates in the real world. Hosts utilized within this experimental environment were host C and host D. Host D was a bare metal endpoint in the NSF Chameleon's platform [30]. So the results for the large environment were obtained using the Chameleon testbed supported by the National Science Foundation.

NSF's Chameleon cloud environment used key authentication for the login process. The other environments used passphrase authentication for the login process. Key authentication could have modified an SSH session's initiation procedure, but the remainder of the network session should not have been affected by this setting.

NSF Chameleon was exactly the type of environment we wanted for experiment on a wide area network specifically because it is a cloud environment. It is common to perform file transfers, such as PDF file transfers, to and from a cloud environment; therefore, our obfuscation methodology pairs well with this type of infrastructure due to the methodology's use of the PDF file format as the cover. This provides a really good adversarial scenario. We were able to determine how successfully an adversary can obfuscate an SSH session to a cloud infrastructure.

5.4 Results

In order to understand the behavior of an SSH session's network traffic, a control experiment did not use the PDF obfuscation program. This un-obfuscated SSH session allowed us to analyze the inter-packet delay (IPD) and packet size characteristics within the session. After a base case SSH session was obtained, three additional experiments were conducted using PDF obfuscation. One of these three additional experiments consisted of establishing an SSH session using the PDF obfuscation program without explicitly hiding the target protocol's IPD or packet size characteristics. This basic obfuscated SSH session allowed us to determine any changes to the session's network traffic due to the obfuscation procedure. Another one of the three experiments consisted of establishing an SSH session using the PDF obfuscation program, which explicitly hid the target protocol's IPD characteristics, by increasing the obfuscation tunnel's transmission rate. This experiment did not explicitly hide the packet size characteristics. This IPD obfuscated SSH session allowed us analyze changes to the session's network traffic due to the tunnel's increased transmission rate, and it allowed us to determine if the obfuscation program can successfully hide the target data as well as the target protocol's IPD characteristics. The final of the three experiments consisted of establishing an SSH session using the PDF obfuscation program, which also explicitly hid the packet size characteristics. It did not explicitly hide the target protocol's IPD characteristics. This packet size obfuscated SSH session allowed us analyze changes to the session's network traffic due to the obfuscation program transmitting packets at a constant size, and it allowed us to determine if the obfuscation program can successfully hide the target data as well as the target protocol's packet size characteristics.

As mentioned in Section 5.3, during each SSH session, a specific set of commands were executed in a predefined sequence, in order for the hosts to respond in the same manner for each experiment. These commands consisted of listing and traversing directories (pwd, cd, and ls), outputting two files to standard out (cat), opening and closing a file in an text editor (vim), and terminating the session (logout). We executed these commands in the following sequence: ls, pwd, cd <DIRECTORY>, ls, cat <FILENAME1>, cat <FILENAME2>, vim <DIRECTORY>/<FILENAME3>, logout. We decided to use these type of commands for the experiments because they can be used by an adversary to inject into, or exfiltrate data from, a system.

The figures in this section have numerical labels that specify a command's execution within the network session. If multiple packets correspond to a single command, then they are grouped together in a box. The labels correspond to the commands in the following manner, as seen within Table 1. Labels 1-4 correlate to the four directory listing and traversal commands (ls, pwd, cd, ls). Labels 5 and 6 correlate to the two files that were outputted to standard out (cat, cat). Label 7 correlates to a file being opened and closed in a text editor (vim). Label 8 correlates to the user logging out of the remote host (logout).

For our experiments, we utilized three different environments of varying sizes in order to determine how the run-time context affects a network session. These environmental sizes consisted of an isolated network, a local area network (an organization's intranet), and a wide area network (the Internet). The three different types of experiments detailed above were conducted within each of these environments in order to determine if the obfuscation methodology, with and without the IPD reduction or constant packet size feature, can prevent a monitoring entity from identifying the session. Furthermore, the sequence of commands we previously detailed were executed within each SSH session; because, they can be used to insert, delete, or obtain data from a system. Table 4 is where we assigned each experiment an ID (Experimental ID or EID) with details of the performed experiment including the environment, whether or not the experiment obfuscated the SSH session, and whether or not the obfuscation program explicitly hid the target protocol's IPD or packet size characteristics.

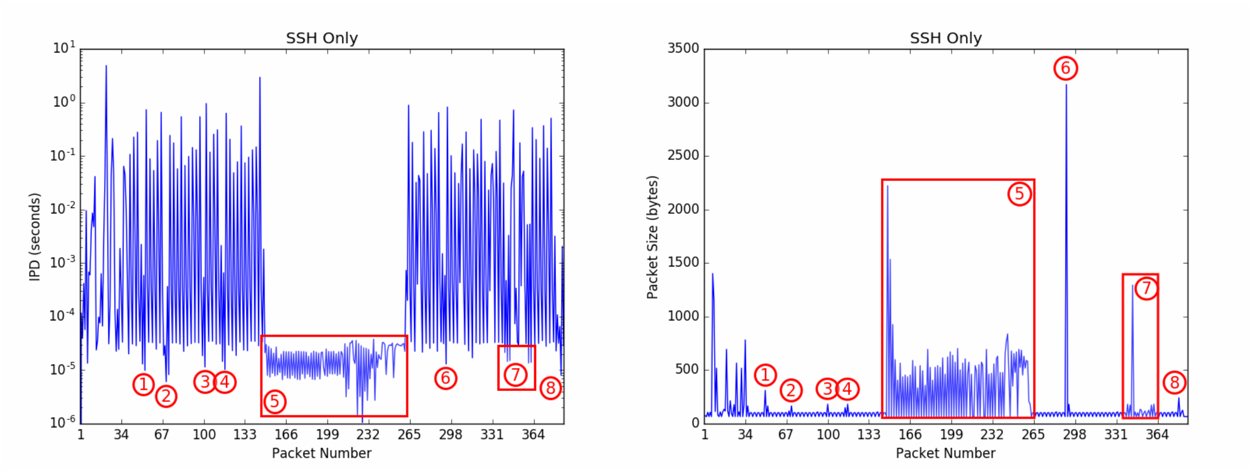
Assuming the network session is operating an official (un-altered) SSH application, it is possible for a monitoring entity to identify command execution. First, we will discuss the results obtained for Experimental IDs 1 and 2 especially their similarities. The isolated environment provided us the base case characteristics; and in this environment, we identified a pattern within the un-obfuscated and obfuscated session's charac-

Experimental ID (EID)	**Experiment**
1	Isolated Network with an un-obfuscated SSH session
2	Isolated Network with an obfuscated SSH session
3	Isolated Network with an obfuscated SSH session plus hiding the target protocol's IPD characteristics
4	Isolated Network with an obfuscated SSH session plus hiding the target protocol's packet size characteristics
5	Local Area Network with an un-obfuscated SSH session
6	Local Area Network with an obfuscated SSH session
7	Local Area Network with an obfuscated SSH session plus hiding the target protocol's IPD characteristics
8	Local Area Network with an obfuscated SSH session plus hiding the target protocol's packet size characteristics
9	Wide Area Network with an un-obfuscated SSH session
10	Wide Area Network with an obfuscated SSH session
11	Wide Area Network with an obfuscated SSH session plus hiding the target protocol's IPD characteristics
12	Wide Area Network with an obfuscated SSH session plus hiding the target protocol's packet size characteristics

Table 4: Protocol Obfuscation Experiment Quick Reference Guide. These Experimental IDs (EIDs) are utilized throughout this section in order to quickly reference the experiment(s) performed while discussing its/their results.

teristics such that we can identify when a command was executed within the session. Since SSH sessions are based on the user's input, a monitoring entity can identify when the SSH server was executing a command (Figure 25 and 26). The IPD characteristics visually identify when the server was executing and transmitting a command's output back to the client with the un-obfuscated and obfuscated SSH session (Figures 25a and 26a). If the server was executing and outputting the contents of a command, then there was a sudden decrease in the IPD (around 22.5 microseconds). The packet size characteristics also can be visually identified when a command transmitted data to the client (Figures 25b and 26b). Specifically, if the server was transmitting a command's output to the client, then the packet size was larger than the benign packet size. Otherwise, if the user was idle or typing in a command, the packet sizes were around the size of every other benign packet sent during the session. A benign packet size is a packet that does not contain data corresponding to an executed command.

We will discuss some of the differences within the results obtained from Experimental IDs 1 and 2 since we now have a good understanding of the similarities between an un-obfuscated and obfuscated SSH session. Interestingly, the IPD does not fluctuate as frequently when the obfuscation program is incorporated into the SSH session (Figure 25a vs Figure 26a), and it appears significant IPD decrease only occurred for command execution (Figures 26a). So if anything, the obfuscation program actually accentuates and clearly marks when a command is executed within the SSH session's IPD characteristics, which is the exact opposite of its



(a) IPD per Packet

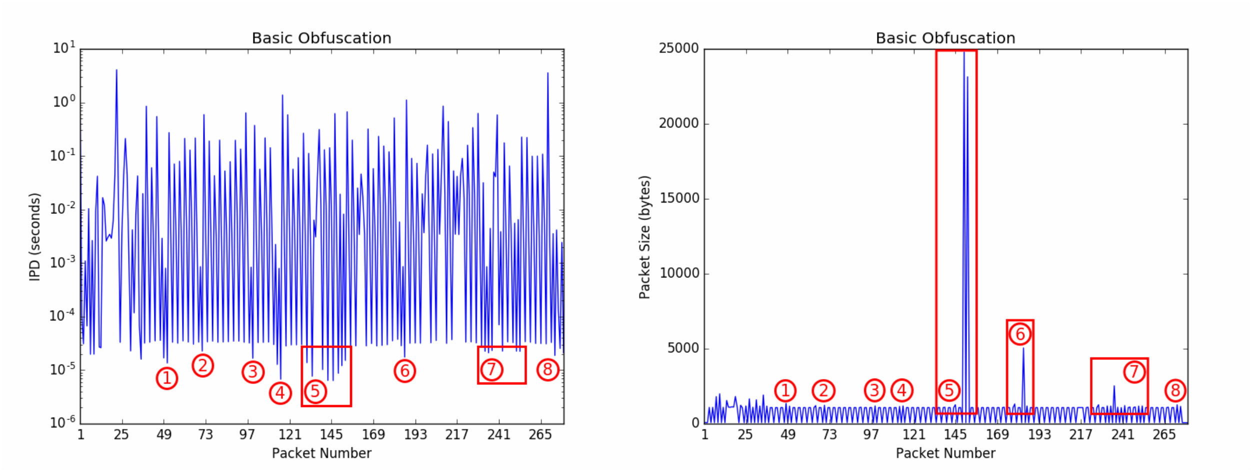
(b) Bytes per Packet

Figure 25: SSH session characteristics. This un-obfuscated SSH sessions shows the IPD and packet size characteristics of the SSH protocol. It shows how a monitoring entity can determine when a command was executed based on the SSH protocol's characteristics, as well as the type of command executed.

intended goal. Most of the command types are also accentuated and can be successfully identified through IPD analysis, but an auditor could potentially misidentify an individual command as multiple commands, as well as misidentify the type of command due to the lack of successive IPD decreases (commands 5-7).

The obfuscated SSH session's benign packet sizes are consistently an order of magnitude larger than the non-obfuscated SSH session's benign packet sizes. More specifically, the obfuscated SSH session's benign packet sizes are about 10 times larger than the un-obfuscated SSH session's benign packet sizes. This increase in packet size is a direct result of wrapping the SSH data within a PDF file. Therefore, the exact increase in packet size is dependant on the type of obfuscation that is being performed. The packet size characteristics (Figure 26b) have clear indicators for the file input/output command type (commands 5-7). But, it becomes more difficult to identify the other command types that output only a minor amount of data to standard out (commands 1-4 and 8), because they can inherently be disguise as a benign packet. We also find it interesting how command 5 was transmitted with packets that had a significantly larger packet size than the un-obfuscated SSH session, which could indicate the obfuscation program changes the packet size characteristics for commands that output a lot of data to standard out.

We will discuss some final thoughts for the results obtained from Experimental IDs 1 and 2. The isolated environment also allowed us to determine a monitoring entity can identify the type of command that was executed, except when the IPD reduction and constant packet size feature were utilized. While it may be possible to use characteristics of the session's network traffic to identify an SSH session as well as when a command was executed, it would be very difficult to identify the exact command that was executed. However, it is possible to use the IPD and/or packet size characteristics to identify whether or not a command was either



(a) IPD per Packet

(b) Bytes per Packet

Figure 26: PDF Obfuscation Session Characteristics. Even though the SSH characteristics are still present, the IPD of command 5 appears as if multiple commands were executed, instead of a single command. Additionally, other than the file input/output commands, it became difficult to identify command execution by analyzing the packet size characteristics. All these subtle changes merely came from incorporating the basic obfuscation procedure into the SSH session.

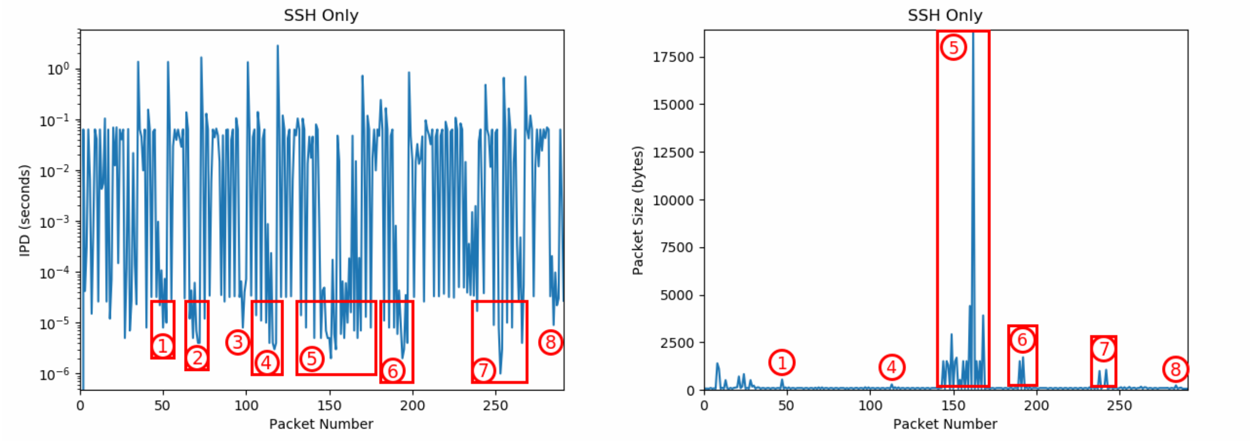
a file input/output; directory listing or traversal; or session termination command, as we have discovered through our analysis of the un-obfuscated SSH session within the isolated environment. The directory listing, directory traversal, and session termination commands all have similar IPD and packet size characteristics. Specifically, these three type of commands have an identifiable packet with a sudden IPD decrease and non-benign packet size. This makes it difficult to distinguish between the execution of the directory listing and traversal command from the session termination command (commands 1-4 vs command 8) based on the network session's IPD and packet size characteristics. But we argue that it is safe to assume the final IPD decrease or non-benign packet size of an SSH session is the session termination command (logout command). As for the file input/output commands, the large and small file input/output commands (commands 5 and 7) have multiple packets with a low IPD and significant packet size(s), which is especially emphasized with command 5. While it maybe difficult to use the IPD characteristic to distinguish command 6 as a command that outputs the medium file to standard out, it is possible to use the packet size characteristics to identify it as a file input/output command (e.g.: cat, head, tail, vim, nano), when compared to the packet sizes of commands 1 through 4 and 8.

That being said, it is extremely difficult if not impossible for a monitoring entity to identify the exact command that was executed (ex: cat, head, tail, vim, nano, nmap, or tcpdump), because the network session's characteristics only indicate either the data's transmission rate or the amount of data that the command outputted. The executed commands could be anything that prints data to standard output. Thus, depending on the environment and/or run-time context, a monitoring entity can use one of the session's characteristics to successfully identify a command as either a directory listing and traversal command, a session termination command, or a file input/output command type.

To finalize the analysis of the SSH session's characteristics within the isolated network, when a user inputs a character, the server either stores the character into a buffer until the buffer gets full, or a newline occurs, or the server waits until a timeout expires before echoing the character back to the user [49]. It is normal for a remote login service to echo the user's input back to the client, which indicates why the IPD widely fluctuates between command execution. Therefore, these echo messages can be used to characterize the network traffic as an SSH session. There were a lot of packets transmitted during the session with a benign packet size, which indicates these packets correspond as echo messages.

Now that we have a good base understanding of an un-obfuscated and an obfuscated SSH sessions' characteristics, we will discuss the results obtained from Experimental IDs 5, 6, 9, and 10, but first we will go over Experimental IDs 5 and 9. The local and wide area networks demonstrated how an SSH session can generate either IPD characteristics that mimic our base case characteristics, which we will term call a "clean" session (Figure 27); or, IPD characteristics completely unrelated to our base case characteristics (Figure 28), which we will call an "un-clean" session. In other words, these experiments conducted within both of these networks show a session's characteristics are able to change from session to session. This indicates the environment's run-time context (ex: client's and server's processing load, OS's scheduler, network congestion, network QoS, etc) has an impact upon a session's characteristic, most of which can be directly influenced by the environment's size.

Moreover, these two networks (local and wide area networks) also demonstrated how the increase in environment size (i.e. compared to the isolated network) modified almost all of the un-obfuscated SSH command's IPD characteristics used to correlate the file input/output command type for "clean" and "un-clean" sessions; especially when using the wide area network. The experiment within these networks also showed that the IPD characteristics could cause a misidentification of the amount of data being transmitted per command. For example, the IPD characteristics in the worse case could make a command correlate to a file input/output command outputting a large file to standard output when the command was a directory listing command. Furthermore, the increase in environment size could cause a command to be un-identifiable (command 3), or the IPD characteristics of sequential commands merged together (commands 5 and 6, where command 6 is labelled with command 5 in the figure), or an individual command could be seen as multiple commands (command 7), which is especially displayed within the "un-clean" session (Figure 28a). The IPD characteristics might not have the identification information for identifying command execution so a command might not be identified.



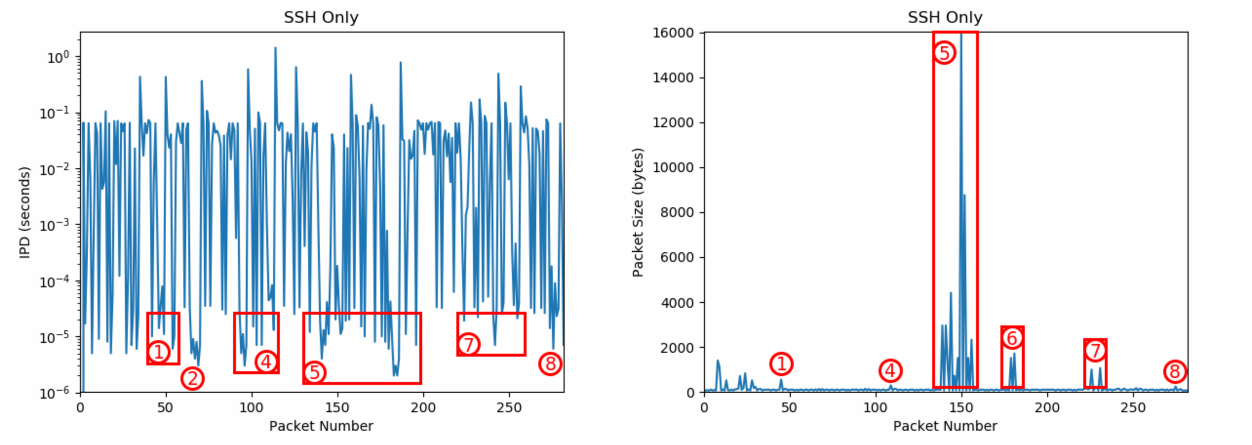
(a) IPD per Packet (b) Bytes per Packet

Figure 27: SSH Session Characteristics with the Internet. This un-obfuscated SSH session shows the IPD and packet size characteristics of the SSH protocol within a large environment. And, it shows how a monitoring entity will need to cross-reference the IPD and packet size characteristics in order to successfully identify command execution plus the type of command executed.

The packet size characteristics for the local and wide area network's un-obfuscated SSH session show it is practically the same as to our isolated network's packet size characteristics; specifically, these characteristic prove to be a more reliable method for command and protocol identification than the IPD characteristics. Whether it is a "clean" or "un-clean" session (Figures 27b and 28b), a monitoring entity could successfully identify both command execution and the type of command executed by analyzing the packet size characteristics for almost all of the commands (commands 1 and 4-8). As for commands 2 and 3, the local and wide area networks demonstrates how it becomes increasingly difficult to impossible to identify these two directory listing and traversal commands (commands 2 and 3) because they did not output a lot of data to standard out. Additionally, it becomes very difficult to identify the session termination command's (command 8) execution within the wide area network.

We will discuss the results obtained from Experimental IDs 6 and 10 because of our understanding of an un-obfuscated SSH session within a local area network and a wide area network (EIDs 5 and 9). The obfuscation program in the local and wide area networks accentuated and clearly marked a command's execution within the SSH session's IPD and packet size characteristics, which was also the case within the isolated environment and counteracts its intended goal. And interestingly, both networks morphed their obfuscated SSH session's IPD and packet size characteristics to better resemble the isolated network's un-obfuscated SSH session over the isolated environment's obfuscated SSH session (Figure 25a vs Figure 26a).

The "clean" and "un-clean" obfuscated SSH sessions within the local area network really started to show us how the run-time has a profound impact upon a session's characteristic; to be precise, the IPD

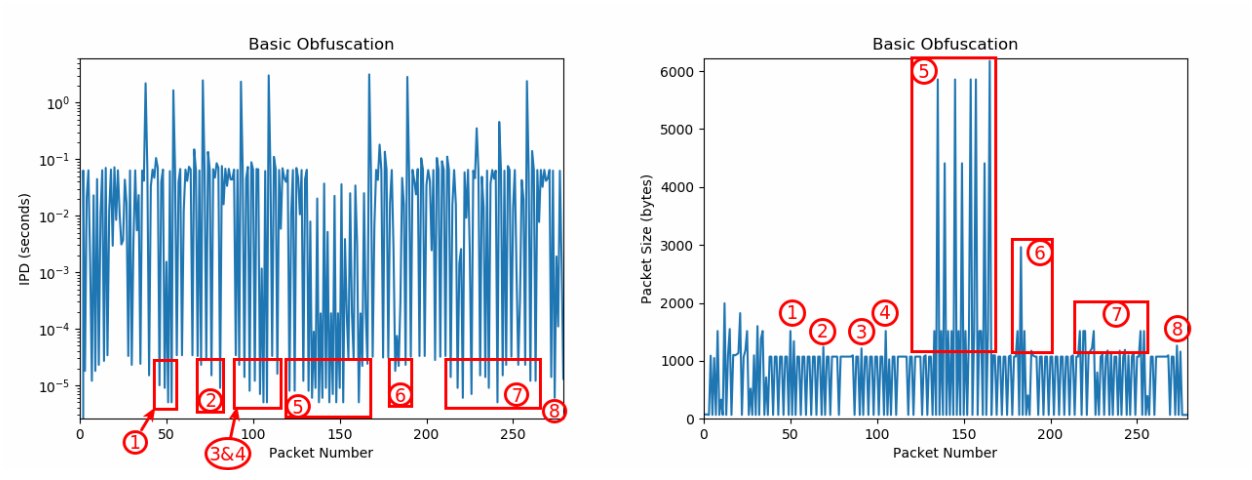


(a) IPD per Packet (b) Bytes per Packet

Figure 28: "Un-clean" SSH Session Characteristics with the Internet. This un-obfuscated SSH session shows the IPD and packet size characteristics of the SSH protocol within a large environment. And, it shows how the IPD characteristics cannot be used to identify all of the commands execute; plus, the IPD characteristics rarely correspond to the type of command that was executed. However, the packet size characteristics can still be used to identify almost all of the commands executed and the type of command that was executed.

characteristics. The local area network's "clean" obfuscated SSH session had clearly marked command execution as well as accurately characterize the type of command executed except for possibly command 6, which we go over below. On the other hand, some of the executed command's type could be misidentified within this environment's "un-clean" obfuscated SSH session's IPD characteristics, for example this "un-clean" session's IPD characteristic's for command 5 became irregular and two of the directory listing and traversal command's execution (commands 2 and 3) cannot be found. As the environment became larger (wide area network), it becomes more difficult to identify the type of command executed by analyzing the obfuscated SSH session's IPD characteristics. Figure 29a displays how the obfuscation program's IPD characteristics of two commands (commands 3 and 4) were merged together, which was also seen within the wide area network's "un-clean" un-obfuscated SSH session; so, a monitoring entity would not be able to identify all of the commands executed. Furthermore, the wide area network's "un-clean" obfuscated SSH session's IPD characteristics (Figure 30a) had various commands (commands 3 and 4, 6 and 7, 7 and 8) merged together as well as grouped two commands (commands 1 and 2) together so that they are extremely difficult to be separately identified. These figures also shows how a lot of the IPD characteristics correlate to the file input/output command type. So, the IPD characteristics could misidentify the amount of data being transmitted per command within the local and wide area networks, which we saw within these networks' un-obfuscated SSH sessions.

And there was an interesting change with the text editor's (command 7's) characteristics. Command



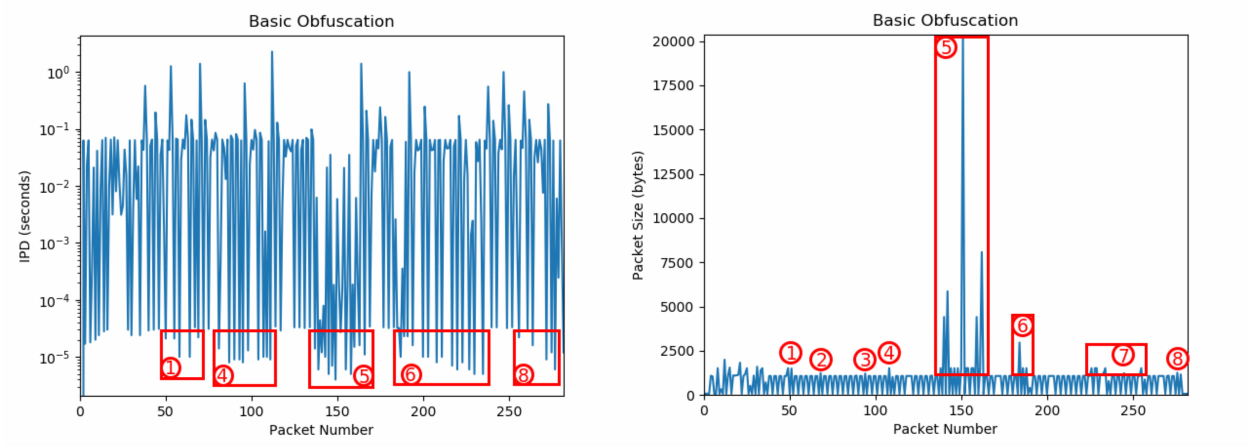
(a) IPD per Packet

(b) Bytes per Packet

Figure 29: PDF Obfuscation Session Characteristics within the Internet. This obfuscated SSH session shows how the IPD characteristics for an executed command could merge with its neighbor's characteristics, which prevents a monitoring entity from successfully identifying all of the executed commands. In addition, the IPD characteristics for an executed command does not always represent the type of command execute. So, a monitoring entity will need to cross-reference the IPD and packet size characteristics in order to successfully identify command execution plus the type of command executed.

7 increased to an identifiable single file input/output command type within the local area network and then increased again to an identifiable single file input/output command type that outputted a large file to standard output within the wide area network; so, the text editor command became more pronounced as the environment got larger. Whereas, command 6's IPD characteristics better correspond to the file input/output command type, but it was not fully accentuated and so could still be misidentified as a directory listing and traversal command within the local and wide area network. A monitoring entity could mistakenly correlate command 6 as multiple directory listing and traversal commands performed with quick succession when analyzing the local area network's "un-clean" obfuscated SSH session IPD characteristics; in other words, it is difficult for to successfully identify the type of command when analyzing the IPD characteristics for command 6.

The local and wide area network's obfuscation SSH sessions' packet size characteristics are nearly identical to those in the isolated network's. Both large network session's packet size characteristics accentuate and clearly mark all of the commands executed; plus, the type of command executed can be accurately identified and the characteristics resemble the amount of data transmitted per command (Figure 29b). Although, command 7 could be misidentified as multiple file input/output commands sequentially executing within the wide area network. Another important similarity is what these networks' obfuscated sessions had with the isolated network's obfuscated session is that they all had benign packet sizes about 10 times larger than the corresponding network's un-obfuscated SSH session's benign packet sizes. These results clearly



(a) IPD per Packet

(b) Bytes per Packet

Figure 30: "Un-clean" PDF Obfuscation Session Characteristics within the Internet. This obfuscated SSH session shows how the IPD characteristics cannot be used to identify all of the commands executed; plus, the IPD characteristics usually correspond to the file input/output command type. However, the packet size characteristics can still be used to identify almost all of the executed commands and the type of command.

show that the packet size characteristics can be used as a metric for analysis, because they were consistent throughout all three networks and for both "clean" and "un-clean" sessions. The obfuscation program allows a monitoring entity to identify two of the directory listing and traversal commands (command 2 and 3) that were previously un-identifiable within the wide area network's un-obfuscated SSH session.

We will discuss some final thoughts on protocol obfuscation since we know about an SSH session's characteristics with and without obfuscation (EIDs 1, 2, 5, 6, 9, 10). These environments' results show that a monitoring entity could use either the IPD or packet size characteristics to successfully analyze an SSH session. However, when using a large environment like the wide area network or the run-time context meets a certain state (such as if the session was "un-clean"), a monitoring entity will need to cross-reference both the IPD and packet size characteristics in order to successfully identify command execution as well as the type of command executed. A monitoring entity can succeed by cross-referencing the IPD and packet size characteristics primarily due to the reliability of the analysis of the packet size characteristics. In addition, cross-referencing the characteristics can assist a monitoring entity in identifying a file input and output command type vs a file output command type, because the file output command type has a constant and un-interrupted IPD decrease and packet size increase; but, the file input and output command type has an inconsistent IPD decrease and a constant and un-interrupted packet size increase.

In both the local and wide area networks, the PDF metadata information (cover data) took up approximately 1000 of the 1500 bytes available for processing each packet (Figure 29b). Even though the metadata does not have to do with the target data, the metadata has to be forwarded by the networking devices, which takes up resources and reduces the overall goodput. Additionally, the client is further away from the server, unlike within the isolate network, and the obfuscated session is not the only connection being processed within the local and wide area networks, which directly affects the ACKs and Nigle's algorithm. The greater the distance between the two endpoints will most probably increase the amount of networking devices that forward the session's packets, which will increase the chances of the SSH session sharing resources with traffic heavy or highly prioritized connections. This paired with the run-time context, which in this case was the networking devices processing and possibly prioritizing various other sessions, could have prevented some packets from containing more than 1500 bytes of data.On the other hand, there were some packets that contain more than 1500 bytes of data, which could be a result of packet switch and routing algorithms that caused those packets to be forwarded through a different routing path than the 1500 byte packets. This different routing path(s) contained networking devices that were able to transmit more than 1500 bytes of data per packet; or, the environment's run-time context briefly processed the session long enough to transmit more than 1500 bytes.

The un-obfuscated SSH session was sometimes more efficient at transmitting command 5's output to the client because the session has to follow the TCP standards and Nigle's algorithm [40]. So, even if command 5 was able to quickly fill the server NIC's transmit buffer with the entire contents of the file, the server has to wait for the corresponding TCP ACKs before transmitting the next set of data; and, the server has to limit its transmission rate based on Nigle's algorithm. In other words, the environment would not or could not support transmitting command 5's output plus the PDF metadata in a single packet because the amount of data was too big.

Whether or not it is a "clean" or "un-clean" session, there are some IPD characteristics that consistently appeared within all of the three environment's network sessions. The fluctuation range (less than 10^1 but greater than 10^{-6}) did not change due to the increase in environment size. Command execution can be identified if the IPD was around 22.5 microseconds. Also, the SSH initiation procedure (key exchange, coordinating the operating SSH version, login message that is outputted to the client, etc.) is identifiable or closely resembled at the beginning of an SSH session; however, the wide area network had some network sessions with an SSH initiation procedure difficult to identify. The packet size characteristics had almost the exact same patterns throughout all of the three environments and all of the network sessions, which we mentioned previously. These characteristics constantly allowed a monitoring entity to identify command execution and the type of command executed, except for the few cases mentioned above. Moreover, the packet size characteristics marked the SSH initiation procedure.

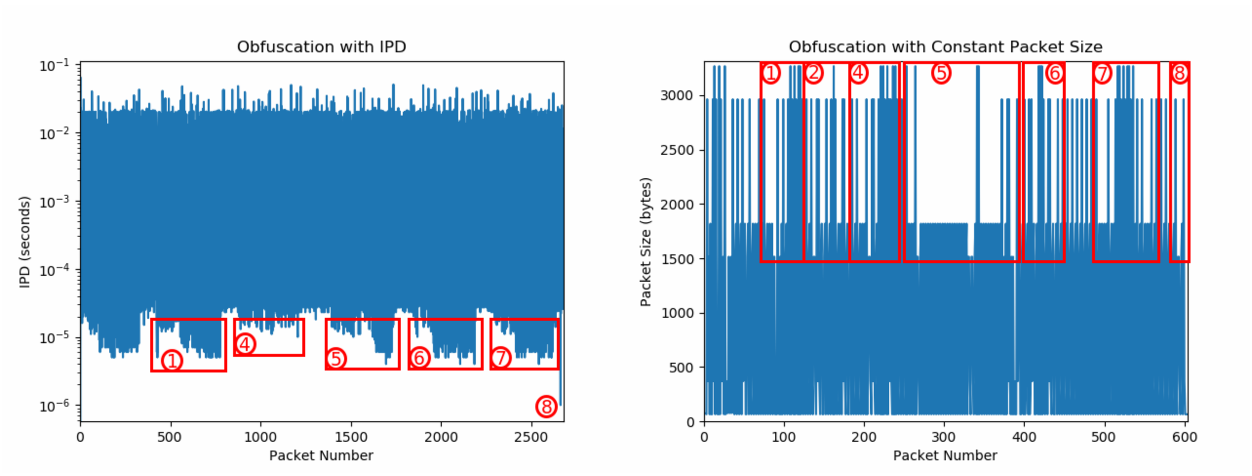
These results clearly state that the obfuscation program does not completely and holistically hide a command's execution, where holistically means the session characteristics such as the IPD characteristics.

Furthermore, the environment's run-time context cannot subsidize for the obfuscation program to hide the commands' execution, so it is still possible for a monitoring entity to identify an obfuscated SSH session as an SSH session. It is necessary for the obfuscation program to purposely hide the SSH protocol's characteristics while obfuscating the target data at the same time.

In order to improve the obfuscation methodology and help prevent a monitoring entity from identifying the obfuscated session as an SSH session, two features were incorporated into the obfuscation program. These two features include:

1. Reducing the packets IPD so that the obfuscation program transmits a packet within a specified time; and

2. Ensuring all of the transmitted packets have the same size.



(a) IPD per Packet (b) Bytes per Packet

Figure 31: PDF obfuscation plus hiding the SSH characteristics within the Internet. If the IPD modifying feature and constant packet size feature does not use a value optimized for the environment, then a monitoring entity can identify some command execution.

We will discuss the results obtained for the obfuscated sessions that hide either the IPD or packet size characteristics (EIDs 3, 4, 7, 8, 11, and 12). The IPD modifying feature slightly to significantly modified the session's IPD fluctuation range (less than 10^0 but greater than 10^{-6} to less than 10^{-1} but greater than 10^{-5}), and it appears to have hidden the SSH session's initiation procedure and the login message that is sent to the client. There were occurrences of the IPD decreasing below 22.5 microseconds to indicate command execution; but it is extremely difficult, if not impossible, to accurately identify the IPD decreases as commands without knowing the executed commands plus their sequence of execution. The isolated network had only some occurrences of the IPD decreasing below 22.5 microseconds where the local and wide area networks had the IPD constantly and randomly drop below 22.5 microseconds. Interestingly, this feature's use within the local area network did not fully hide executed commands because command execution might be indicated if the IPD reached and did not stayed at a peak value. But, this requires analyzing IPD increases along with IPD decreases for command execution and we can argue this effect would disappear if the IPD modifying feature's value is further tailored towards the local area network.

This feature's use within the local and wide area networks show that its exact timeout value depends on the operating environment. If the timeout is set too low, then a monitoring entity could still identify command execution (Figure 31a). However, if the timeout is set too high, then the target protocol's goodput will several decrease due to the obfuscation tunnel's use of random data.

The constant packet size feature significantly modified the session's packet size characteristics. This feature was also able to successfully hide the SSH session's initiation procedure and the login message that is sent to the client. It also hides the command's execution as well as the target protocol session (i.g. an SSH session). The constant packet size feature's value also needs to be tailored towards the operating environment; because if not, we noticed command execution could still be identified or ascertained within the local and wide area networks (Figure 31b). The network session's packet size fluctuates, but this fluctuation is unavoidable because the OS automatically transmits TCP ACK packets through the TCP socket. While these ACK packets have zero bytes in the payload, they have a packet size of 66 bytes due to the header information.

The constant packet size feature was successfully able to transmit packets with a constant payload within the wide area network, except for the SYN and ACK packets. Interestingly, the local area network had some packets with a different sized payload but it is very difficult to determine any information from analyzing the entire session's bytes per packet due to the near identical packet sizes, ignoring the SYN and ACK packet sizes. Therefore, we consider the constant packet size feature successful within the local area network. There were two occurrences where the packet size was 78 bytes, which could be due to a TCP data retransmission. A majority of the packet sizes were either 1516 bytes or 1514 bytes, where a 1514 byte packet either proceeded or was followed by a 68 byte packet. We argue this 68 byte packet contains the 2 bytes (66 bytes are header information) a 1514 byte packet contained, which would have made the packet a 1516 byte packet size. This packet size variation could have been caused by a run-time context variable.

Since the obfuscation program is using a TCP socket, and since TCP is a byte oriented protocol, the socket API's send function call would insert the data into the kernel's network buffer before sending the data through the wire. This led to the OS occasionally "optimizing" the amount of data sent within a single packet by combining the payloads that were intended to be transmitted within multiple packets, which prevented some packets from having the pre-defined packet size. We overcame this affect by purposely executing a delay before the next packet's data was inserted into the buffer, in order to ensure that the buffer was empty.

This delay would need to be customized for the environment (i.e. hosts and network) that the obfuscation program is operating in and it could conflict with the IPD feature, if both features are enabled at the same time. As our experiments expanded into the local and wide area networks, it was necessary to replace the delay function with the socket function that returns the amount of data present in the kernel's network buffer, because the increase in the environment's size made it more trying to find the tailored packet size value. So, the constant packet size feature would give the kernel the next packet's payload if and only if the NIC's transmit buffer was empty. In addition, Nigle's algorithm was disabled for the obfuscation tunnel's socket.

Throughout the three environments we saw how the constant packet size feature was successful, but it does have a drawback. A monitoring entity could identify the network session as some un-identifiable protocol that is being obfuscated because all of packets have a constant packet size. In order to better hide the SSH session's packet size characteristics, the obfuscation program could transmit packets with varying packet sizes that mimic the network's normal packet size distribution. Or the obfuscation program could transmit varying PDF sizes that follow the PDF size distribution normally seen within the operating network.

We have analyzed an SSH session's characteristics with and without obfuscation, so we will now summarize our final thoughts of the results based on the experiments performed (EIDs 1 to 12). To summarize, the LAN and WAN environments demonstrated how the run-time context can generate "clean" and "unclean" session characteristics; more specifically, how the IPD characteristics do not always cleanly display each executed command and/or correspond to the type of command executed. Furthermore, the run-time context can prevent a monitoring entity from identifying command execution due to multiple command's merging their IPD characteristics together or when the server does not transmit a substantial amount of data back to the client. However, when the environment's size was not too large, the obfuscated session's characteristics clearly marked command execution and closely resembled the control (i.e. un-obfuscated base case) session's characteristics. Within too large environments, the environment would inherently modified the session's IPD characteristics of an executed command so that the command's execution corresponded to the file input/output command type. Additionally, the environment would sometimes merge some commands together; so, even though a monitoring entity would be able to identify command execution, they could not determine the correct amount of commands executed by just looking at the IPD characteristics. This means a monitoring entity would need to analyze and utilize both the IPD and packet size characteristics in order to successfully identify command execution and the type of command executed.

After all of the experiments were conducted, we noticed obfuscating the target data does not completely and holistically prevent a monitoring entity from identifying a network session as an SSH session. The IPD characteristics always mark command execution when dropping below 22.5 microseconds, even though the IPD characteristics are not always reliable for a holistic analysis. On the other hand, the packet size characteristics are very reliable for analysis because all of the experiments packet size characteristics almost always marked command execution and accurately corresponded to the type of command executed. Therefore, any obfuscation program will need to hide the target protocol's characteristics as well as the target data, especially the packet size characteristics, in order to successfully obfuscate the protocol. And for the obfuscation methodology that can mask all of the protocol's characteristics, the exact operation that masks the target protocol's characteristics will need to be tailored to the running environment, as we saw from our experiments with the IPD modifying feature and constant packet size feature. Some additional characteristics an obfuscation program should consider hiding include the target protocol's IPD fluctuation rate and range plus the initiation procedure, if applicable.

5.5 Conclusion

Through these experiments and various environments we have shown that an obfuscation program will need to hide the target protocol's network characteristics, especially the packet size characteristics, along with the target data, in order to completely and holistically obfuscate the target protocol. Additionally, we have shown that the obfuscation program itself pronounces a command's execution, so a monitoring entity can easily identify command execution. The obfuscation program can use features such as the IPD reduction and constant packet size feature to help hide the network session's characteristics, but these features' threshold values have to be tailored for the operating environment. The environments (medium and large) that had multiple users showed use that the environment's run-time context inherently modifies the session's IPD characteristics to the point where they become unreliable for an accurate individual analysis of the network session. But, a monitoring entity can cross-reference both of the characteristics to determine command execution and the type of command executed, even in a multiple user environment. Lastly, a monitoring entity cannot use the network session's characteristics to identify the exact command executed (ex: cat, vim, or tcpdump).

5.6 Future Work

This research can be expand into a wide array of protocol obfuscation research topics. First and foremost, researchers can continue researching protocol obfuscation through other types of file formats other than PDFs (ex: PNG, JPEG, etc). A second potentially promising protocol obfuscation research topic is hiding the target protocol session (ex: SSH) within a different protocol (ex: HTML). For example, one could obfuscate the experiment's SSH sessions within the HTML protocol instead of a PDF file. The third obfuscation research topic includes intentionally altering the obfuscated session's characteristics to correspond to a different aspect of the target protocol's characteristics than the actual running state. For example, the obfuscation method could be modified to make a file input/output command, that is accessing a medium sized file, look like multiple directory listing commands of a nearly empty directory. Or, the obfuscation methodology could intentionally morph the obfuscated session's characteristics to mimic a different protocol's characteristics, so that the obfuscated session does not reproduce the target protocol's characteristics.

As seen throughout these experiments, there were occasions where the network session's characteristics did not correspond to the type of command executed. Additional research is required for determining if a command's execution can consistently correspond to the wrong type of command. For example, can the execution of a directory listing or traversal command consistently correspond to the file input/output command type, because either the directory contains a vast amount of files and/or sub-directories or the path to the new directory is excessive. We speculate if the directories contained an abundant amount of files and/or sub-directories, then the directory listing commands could have the same characteristics as a file input/output command. Afterwards, the next step is to determine if the obfuscation methodology can intentionally alter the characteristics of the obfuscated session such as altering a directory listing and traversal command's characteristics to correspond to the file input/output command type, or a command emitting a large amount of data to look like it is multiple commands accessing a small amount of data. We further speculate command 5's characteristics could be altered so that it does not look like a lot of data is being printed to standard output. By printing the large file in smaller partitions (e.g.: print 10 lines at a time), instead of printing it all at once, the session characteristics could look like any command such as a directory traversal command type. This option could be implemented within the obfuscation methodology by rate limiting the amount of target data it transmits within a certain time period (ex: transmitting the target data in small partitions every 15 milliseconds).

Moreover, further research is required for adding additional features into the obfuscation methodology in order to obfuscate a protocol in its entirety. A monitoring entity might be able to use the IPD characteristics to identify the network session as a remote access protocol (i.e. SSH) even with the IPD modifying feature by determining if the IPD drops below 22.5 microseconds, or by analyzing the network session's fluctuation range. In order to further mask the SSH protocol's IPD characteristics, the obfuscation program would need to be configured so that it does not transmit packets faster than some user defined threshold value (i.e. reduce the fluctuation range), which in this case would be a value higher than 22.5 microseconds (ex: 20 milliseconds). Setting this threshold value to a high delay (ex: 20 milliseconds when compared to 22.5 microseconds) would also reduce the IPD fluctuation rate because the server would send less echo messages to the client.

On the other hand, since the network session's IPD fluctuates because of the remote protocol's use of echo messages, the obfuscation program could use another method to prevent a monitoring entity from identifying the network session as an SSH session. Instead of the obfuscation program increasing the amount of packets sent per timeout, the obfuscation program could decrease the number of packets transmitted per timeout, which can be done in one of two ways. The first method is for obfuscation methodology to use rate limiting to prevent the obfuscation program from transmitting packets faster than some upper bound threshold value. The upper bound threshold value paired with the IPD reduction feature (lower bound threshold value) can be utilized together to reduce the IPD fluctuation range, which helps prevent a monitoring entity from identifying the network session as an SSH session. The second method is for the obfuscation program to be configured to transmit a single packet when the threshold timeout value expires, instead of using an upper and lower bounds threshold value. This timeout expiration value will reduce the IPD fluctuation rate; thus, helping prevent a monitoring entity from identifying the network session as an obfuscated SSH session. However, this method of sending a single packet per timeout expiration could cause the SSH session to become un-responsive due to its inherent decrease in goodput.

Additionally, further research is also required in obfuscating a protocol's packet size characteristics because transmitting packets at a constant packet size is not normal for most network sessions; therefore, it can alert a monitoring entity that the network session is being obfuscated. So, another possible option for hiding the packet size characteristics is to transmit packets with varying sizes, where these varying sizes follow a specified size distribution. This size distribution can be the average packet size distribution of all network sessions that are transmitted within the operating environment, or the average size distribution of the obfuscating cover that is normally seen within the operating environment (ex: the average PDF size distribution transmitted within the network). The obfuscation methodology can also mimic a different protocol's packet size characteristics by transmitting packets that follow the protocol's average session's packet size distribution.

Another research topic for protocol obfuscation is to analyze the characteristics of protocols other than the target protocol, and modify the obfuscation methodology so that the obfuscated session's characteristics is configured to mimic the other protocol's characteristics, while sustaining an efficient amount of goodput for the target protocol. This can be seen as the ideal outcome for protocol obfuscation, because both the target data and the network session's characteristics will appear as a benign transmission of data. But, it will need to alter both the transmission rate and the amount of data transmitted per packet, in order to follow the obfuscated protocol's IPD and packet size distribution. And indirectly, analyzing other protocol's characteristics, especially non-remote access protocols, will verify that remote protocols (i.e. SSH) can be identified as remote access protocols due to their unique characteristics.

6 Conclusion

A resourceful adversary is a patient and skillful entity whom can obtain information (i.g. infrastructure design) that is intentionally hidden so that a vast array of exploitations can be prevented. A resourceful adversary's attack vector can widely vary, from actively scanning a network or system to obtain any and all details about it, to passively scanning the communicating data of a network or system to obtain any and all details about the communicating data and the network or system itself. Our research kept these two attack vectors in mind by determine if an adversary can gain knowledge about an SDN network's control plane and if an adversary (or monitoring entity) can identify a network session with and without protocol obfuscation.

Our work with the controller side-channel attack showed an adversary can gain information about the infrastructure of and SDN network's control plane by being able to determine the number of controllers within an SDN network. When comparing the response times to the control experiment, the controller side-channel attack determined two controllers were operating within the control plane for the Round Robin environment due to a 50% efficiency improvement. The controller side-channel attack determined two controllers were operating within the control plane for the protocol based environment because of the significant difference in processing time based on the packet's protocol. The controller side-channel attack inherently determine the load balancing algorithm utilized by identifying if there is a large and linear increasing difference between the the multiple controller and control experiment environment's average response times or if there are two significantly different average response times due to the packet's protocol. Our results show the Round Robin load balancing algorithm takes as few as 120 mismatched packets and the protocol based load balancing algorithm takes as few as 20 mismatched packets (recommend 80 mismatched packets) in order to determine the number of controllers within the environment and the load balancing algorithm utilized.

The results also showed the OS scheduler as the control plane's (i.e. SDN network's) bottleneck instead of the system's hardware. This is easily seen within the protocol based environment's results, where the default controller becomes congested and the secondary controller has a lower response time. An effective defense against the controller side-channel attack, or any reconnaissance attack type, is to essentially hide the $t_{controller}$ time. The defensive technique is successful because more resources need to be utilized in order to obtain the same or less information. For example, the packets' response times started to increase linearly when 400 or more unique request packets where transmitted on the network, a request needed to be sent multiple times before a reply was received, and the defensive technique had a high response time near 8 milliseconds when the control experiment's environment had a high response time well above 200 milliseconds.

Our work with protocol obfuscation showed that an obfuscation program needs to hide more than just the target data in order to completely and holistically obfuscate the target protocol. The results show that it is important for the obfuscation program to hide the packet size characteristics, as well as the target protocol's other network characteristics (i.g. IPD characteristics), alongside with the target data. Interestingly, an SSH command's execution is pronounced when using the obfuscation program, which makes it easier for a monitoring entity to identify a command's execution.

It is possible to help hide the target protocol's IPD and packet size characteristics with the IPD reduction and constant packet size feature. These two features have to be tailored for the operating environment. The medium and large environments showed how the run-time context inherently modifies the session's IPD characteristics such that they can become an unreliable metric for accurate analysis. But, the packet size characteristics are a reliable metric for analysis so they can be cross-referenced with the IPD characteristics in order to determine command execution and the type of command executed throughout the environments. Lastly, the exact command executed (ex: cat, vim, or tcpdump) cannot be identified by a monitoring entity through the use of network session's characteristics.

7 References

[1] Ahmed Abdelaziz et al. “Distributed controller clustering in software defined networks”. In: *PLoS ONE* 12.4 (Apr. 2017), e0174715. DOI: `10.1371/journal.pone.0174715`.

[2] Andreas Abel and Jan Reineke. “Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation”. In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014, pp. 141–142. DOI: `10.1109/ISPASS.2014.6844475`.

[3] Maurizio Aiello, Maurizio Mongelli, and Gianluca Papaleo. “DNS tunneling detection through statistical fingerprints of protocol messages and machine learning”. In: *International Journal of Communication Systems* 28.14 (2015), pp. 1987–2002.

[4] Talal Alharbi, Marius Portmann, and Farzaneh Pakzad. “The (in)security of Topology Discovery in Software Defined Networks”. In: *2015 IEEE 40th Conference on Local Computer Networks (LCN)*. 2015, pp. 502–505. DOI: `10.1109/LCN.2015.7366363`.

[5] Moreno Ambrosin et al. “LineSwitch: Efficiently Managing Switch Flow in Software-Defined Networking While Effectively Tackling DoS Attacks”. In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS ’15. Singapore, Republic of Singapore: Association for Computing Machinery, 2015, pp. 639–644. ISBN: 9781450332453. DOI: `10.1145/2714576.2714612`. URL: `https://doi.org/10.1145/2714576.2714612`.

[6] Moreno Ambrosin et al. “LineSwitch: Tackling Control Plane Saturation Attacks in Software-Defined Networking”. In: *IEEE/ACM Transactions on Networking* 25.2 (2017), pp. 1206–1219. DOI: `10.1109/TNET.2016.2626287`.

[7] Blake Anderson and David McGrew. “OS fingerprinting: New techniques and a study of information gain and obfuscation”. In: *2017 IEEE Conference on Communications and Network Security (CNS)*. 2017, pp. 1–9. DOI: `10.1109/CNS.2017.8228647`.

[8] Abdelhadi Azzouni et al. “Fingerprinting OpenFlow Controllers: The First Step to Attack an SDN Control Plane”. In: *2016 IEEE Global Communications Conference (GLOBECOM)*. 2016, pp. 1–6. DOI: `10.1109/GLOCOM.2016.7841843`.

[9] *Benchmarking.* 2021. URL: https://fasterdata.es.net/science-dmz/DTN/tuning/benchmarking/.

[10] Roberto Bifulco et al. "Fingerprinting Software-Defined Networks". In: *2015 IEEE 23rd International Conference on Network Protocols (ICNP).* 2015, pp. 453–459. DOI: 10.1109/ICNP.2015.26.

[11] Kuan-yin Chen et al. "SDNShield: Towards more comprehensive defense against DDoS attacks on SDN control plane". In: *2016 IEEE Conference on Communications and Network Security (CNS).* 2016, pp. 28–36. DOI: 10.1109/CNS.2016.7860467.

[12] M. Crotti et al. "Detecting HTTP Tunnels with Statistical Mechanisms". In: *2007 IEEE International Conference on Communications.* 2007, pp. 6162–6168. DOI: 10.1109/ICC.2007.1020.

[13] Mostafa Dehghan et al. "A Utility Optimization Approach to Network Cache Design". In: *IEEE/ACM Transactions on Networking* 27.3 (2019), pp. 1013–1027. DOI: 10.1109/TNET.2019.2913677.

[14] Lucas Dixon, Thomas Ristenpart, and Thomas Shrimpton. "Network Traffic Obfuscation and Automated Internet Censorship". In: *IEEE Security Privacy* 14.6 (2016), pp. 43–53. DOI: 10.1109/MSP.2016.121.

[15] M. Dusi et al. "Tunnel Hunter: Detecting Application-Layer Tunnels with Statistical Fingerprinting". In: *Comput. Netw.* 53.1 (Jan. 2009), pp. 81–97. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2008.09.010. URL: https://doi.org/10.1016/j.comnet.2008.09.010.

[16] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. "Marionette: A Programmable Network Traffic Obfuscation System". In: *24th USENIX Security Symposium (USENIX Security 15).* Washington, D.C.: USENIX Association, Aug. 2015, pp. 367–382. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/dyer.

[17] Open Networking Foundation. *OpenFlow Switch Specification Version 1.3.4 (Protocol version 0x04).* Mar. 2014. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.3.4.pdf.

[18] Sheila Frankel et al. *Guide to SSL VPNs.* en. July 2008. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=152086.

[19] Aditya Ganjam et al. "C3: Internet-Scale Control Plane for Video Quality Optimization". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15).* Oakland, CA: USENIX Association, May 2015, pp. 131–144. ISBN: 978-1-931971-218. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ganjam.

[20] Ehab Ghabashneh and Sanjay Rao. "Exploring the interplay between CDN caching and video streaming performance". In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications.* 2020, pp. 516–525. DOI: 10.1109/INFOCOM41043.2020.9155338.

[21] Liuyong He and Yijie Shi. "Identification of SSH Applications Based on Convolutional Neural Network". In: *Proceedings of the 2018 International Conference on Internet and E-Business.* ICIEB '18. Singapore, Singapore: Association for Computing Machinery, 2018, pp. 198–201. ISBN: 9781450363754. DOI: 10.1145/3230348.3230458. URL: https://doi.org/10.1145/3230348.3230458.

[22] Brandon Heller, Rob Sherwood, and Nick McKeown. "The Controller Placement Problem". In: *SIGCOMM Comput. Commun. Rev.* 42.4 (Sept. 2012), pp. 473–478. ISSN: 0146-4833. DOI: 10.1145/2377677.2377767. URL: https://doi.org/10.1145/2377677.2377767.

[23] E Hjelmvik and W John. "Breaking and Improving Protocol Obfuscation". In: *Department of Computer Science and Engineering, Chalmers University of Technology* Technical Report No. 2010-05 (2010). ISSN: 1652-926X.

[24] Irvin Homem, Panagiotis Papapetrou, and Spyridon Dosis. "Entropy-based Prediction of Network Protocols in the Forensic Analysis of DNS Tunnels". In: *arXiv preprint arXiv:1709.06363* (2017).

[25] Sungmin Hong et al. "Poisoning network visibility in software-defined networks: New attacks and countermeasures." In: *Ndss.* Vol. 15. 2015, pp. 8–11.

[26] Amir Houmansadr et al. "SWEET: Serving the Web by Exploiting Email Tunnels". In: *IEEE/ACM Transactions on Networking* 25.3 (2017), pp. 1517–1527. DOI: 10.1109/TNET.2016.2640238.

[27] Neminath Hubballi and Mayank Swarnkar. “BitCoding: Protocol Type Agnostic Robust Bit Level Signatures for Traffic Classification”. In: *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. 2017, pp. 1–6. DOI: 10.1109/GLOCOM.2017.8254001.

[28] *I2P Anonymous Network*. 2021. URL: https://geti2p.net/en/.

[29] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Systematic Reverse Engineering of Cache Slice Selection in Intel Processors”. In: *2015 Euromicro Conference on Digital System Design*. 2015, pp. 629–636. DOI: 10.1109/DSD.2015.56.

[30] Kate Keahey et al. “Lessons Learned from the Chameleon Testbed”. In: *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC ’20)*. USENIX Association, July 2020.

[31] S. Kent and K. Seo. *Security Architecture for the Internet Protocol*. RFC 4301. RFC Editor, Dec. 2005. URL: https://datatracker.ietf.org/doc/html/rfc4301.

[32] *Time Stamps*. 2022. URL: https://www.wireshark.org/docs/wsug_html_chunked/ChAdvTimestamps.html.

[33] S. Knight et al. *The internet topology zoo.*

[34] Mutalifu Kuerban et al. “FlowSec: DOS Attack Mitigation Strategy on SDN Controller”. In: *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*. 2016, pp. 1–2. DOI: 10.1109/NAS.2016.7549402.

[35] Junyuan Leng et al. “An inference attack model for flow table capacity and usage: Exploiting the vulnerability of flow table overflow in software-defined network”. In: *arXiv preprint arXiv:1504.03095* (2015).

[36] Sheng Liu, Michael K. Reiter, and Vyas Sekar. “Flow Reconnaissance via Timing Attacks on SDN Switches”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 196–206. DOI: 10.1109/ICDCS.2017.281.

[37] Xi Liu et al. “A Case for a Coordinated Internet Video Control Plane”. In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 359–370. ISBN: 9781450314190. DOI: 10.1145/2342356.2342431. URL: https://doi.org/10.1145/2342356.2342431.

[38] Mininet. *Mininet*. URL: http://mininet.org/.

[39] Hooman Mohajeri Moghaddam et al. “SkypeMorph: Protocol Obfuscation for Tor Bridges”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 97–108. ISBN: 9781450316514. DOI: 10.1145/2382196.2382210. URL: https://doi.org/10.1145/2382196.2382210.

[40] John Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. RFC Editor, Jan. 1984. URL: https://datatracker.ietf.org/doc/html/rfc896.

[41] *Obfsproxy: the next step in the censorship arms race — Tor Blog*. 2012. URL: https://blog.torproject.org/obfsproxy-next-step-censorship-arms-race.

[42] Shankaranarayanan Puzhavakath Narayanan et al. “Reducing Latency Through Page-Aware Management of Web Objects by Content Delivery Networks”. In: *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*. SIGMETRICS ’16. Antibes Juan-les-Pins, France: Association for Computing Machinery, 2016, pp. 89–100. ISBN: 9781450342667. DOI: 10.1145/2896377.2901472. URL: https://doi.org/10.1145/2896377.2901472.

[43] *socket — Low-level networking interface — Python 3.10.2 documentation*. 2022. URL: https://docs.python.org/3/library/socket.html.

[44] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, Aug. 2018. URL: https://datatracker.ietf.org/doc/html/rfc8446.

[45] Gao Shang et al. “FloodDefender: Protecting data and control plane resources under SDN-aimed DoS attacks”. In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 2017, pp. 1–9. DOI: 10.1109/INFOCOM.2017.8057009.

[46] Seungwon Shin and Guofei Gu. “Attacking Software-Defined Networks: A First Feasibility Study”. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN ’13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 165–166. ISBN: 9781450321785. DOI: 10.1145/2491185.2491220. URL: https://doi.org/10.1145/2491185.2491220.

[47] John Sonchack, Adam J. Aviv, and Eric Keller. “Timing SDN Control Planes to Infer Network Configurations”. In: *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks &; Network Function Virtualization*. SDN-NFV Security ’16. New Orleans, Louisiana, USA: Association for Computing Machinery, 2016, pp. 19–22. ISBN: 9781450340786. DOI: 10.1145/2876019.2876030. URL: https://doi.org/10.1145/2876019.2876030.

[48] John Sonchack et al. “Timing-Based Reconnaissance and Defense in Software-Defined Networks”. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACSAC ’16. Los Angeles, California, USA: Association for Computing Machinery, 2016, pp. 89–100. ISBN: 9781450347716. DOI: 10.1145/2991079.2991081. URL: https://doi.org/10.1145/2991079.2991081.

[49] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Reading, Massachusetts, USA: Addison-Wesley professional computing series, 1995. ISBN: 0-201-63346-9.

[50] *The Tor Project — Anonymity Online*. 2021. URL: https://www.torproject.org/.

[51] Venkatanathan Varadarajan et al. “A Placement Vulnerability Study in Multi-Tenant Public Clouds”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 913–928. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/varadarajan.

[52] Gunjan Verma et al. “Network Traffic Obfuscation: An Adversarial Machine Learning Approach”. In: *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*. 2018, pp. 1–6. DOI: 10.1109/MILCOM.2018.8599680.

[53] Liang Wang et al. “Seeing through Network-Protocol Obfuscation”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS ’15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 57–69. ISBN: 9781450338325. DOI: 10.1145/2810103.2813715. URL: https://doi.org/10.1145/2810103.2813715.

[54] Zachary Weinberg et al. “StegoTorus: A Camouflage Proxy for the Tor Anonymity System”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 109–120. ISBN: 9781450316514. DOI: 10.1145/2382196.2382211. URL: https://doi.org/10.1145/2382196.2382211.

[55] Philipp Winter, Tobias Pulls, and Juergen Fuss. “ScrambleSuit: A Polymorphic Network Protocol to Circumvent Censorship”. In: *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*. WPES ’13. Berlin, Germany: Association for Computing Machinery, 2013, pp. 213–224. ISBN: 9781450324854. DOI: 10.1145/2517840.2517856. URL: https://doi.org/10.1145/2517840.2517856.

[56] *Wireshark · Go Deep*. 2022. URL: https://www.wireshark.org/.

[57] Guowei Zhu and Weixi Gu. “User Mapping Strategy in Multi-CDN Streaming: A Data-driven Approach”. In: *IEEE Internet of Things Journal* (2021), pp. 1–1. DOI: 10.1109/JIOT.2021.3112561.

[58] Thomas Zink and Marcel Waldvogel. “BitTorrent traffic obfuscation: A chase towards semantic traffic identification”. In: *2012 IEEE 12th International Conference on Peer-to-Peer Computing (P2P)*. 2012, pp. 126–137. DOI: 10.1109/P2P.2012.6335792.

8 Appendix

We use the terms such as "1000 request experiment" or "experiment 460" within this dissertation to reference the response times of an experiment; in this case, "1000 request experiment" is referencing the experiment that had the controller side-channel attack transmit 1000 request packets onto the network and "experiment 460" is referencing the experiment that had the controller side-channel attack transmit 460 request packets onto the network.

The controller side-channel attack used just the ARP protocol for the single controller environment and the Round Robin environment. Table 5 specifies how many ARP requests were transmitted for the single controller and Round Robin environments for each experiment.

Experiment Reference Term	**Number of ARP Requests**
Experiment 1	1
Experiment 20	20
Experiment 40	40
Experiment 60	60
Experiment 80	80
Experiment 100	100
Experiment 120	120
Experiment 140	140
Experiment 160	160
Experiment 180	180
Experiment 200	200
Experiment 220	220
Experiment 240	240
Experiment 260	260
Experiment 280	280
Experiment 300	300
Experiment 320	320
Experiment 340	340
Experiment 360	360
Experiment 380	380
Experiment 400	400
Experiment 420	420
Experiment 440	440
Experiment 460	460
Experiment 480	480
Experiment 500	500
Experiment 520	520
Experiment 540	540
Experiment 560	560
Experiment 580	580
Experiment 600	600
Experiment 620	620
Experiment 640	640
Experiment 660	660
Experiment 680	680
Experiment 700	700
Experiment 720	720
Experiment 740	740
Experiment 760	760
Experiment 780	780
Experiment 800	800
Experiment 820	820
Experiment 840	840
Experiment 860	860
Experiment 880	880
Experiment 900	900
Experiment 920	920
Experiment 940	940
Experiment 960	960
Experiment 980	980
Experiment 1000	1000

Table 5: Experiment Reference Term for the Single Controller and Round Robin Environment

The controller side-channel attack used the ARP and ICMP protocols for the protocol based environment. The default controller processed the ARP packets, and the secondary controller processed the ICMP packets. The ARP packets were flooded onto the network in order to saturate the default controller; but, the ICMP packets were periodically transmitted to the secondary controller. The controller side-channel attack would transmit the ICMP protocol for every tenth mismatched packet; whereas the other mismatched packets contained the ARP protocol. However, if the controller side-channel attack was specified to transmit a single packet for the protocol based load balancing environment, then the controller side-channel attack would transmitted two packets, where one of the packets contained the ARP protocol and the other contained the ICMP protocol.

Table 6 specifies how many ARP and ICMP requests were transmitted within the protocol based environment for each experiment. Due to the use of two protocols, Table 6 states the total number of requests transmitted by the controller side-channel attack and in parenthesis states the number of ARP and ICMP requests transmitted from the total.

Experiment Reference Term	**Total Number of Requests (Number of ARP Requests/Number of ICMP Requests)**
Experiment 1	2 (1/1)
Experiment 20	20 (18/2)
Experiment 40	40 (36/4)
Experiment 60	60 (54/6)
Experiment 80	80 (72/8)
Experiment 100	100 (90/10)
Experiment 120	120 (108/12)
Experiment 140	140 (126/14)
Experiment 160	160 (144/16)
Experiment 180	180 (162/18)
Experiment 200	200 (180/20)
Experiment 220	220 (198/22)
Experiment 240	240 (216/24)
Experiment 260	260 (234/26)
Experiment 280	280 (252/28)
Experiment 300	300 (270/30)
Experiment 320	320 (288/32)
Experiment 340	340 (306/34)
Experiment 360	360 (324/36)
Experiment 380	380 (342/38)
Experiment 400	400 (360/40)
Experiment 420	420 (378/42)
Experiment 440	440 (396/44)
Experiment 460	460 (414/46)
Experiment 480	480 (432/48)
Experiment 500	500 (450/50)
Experiment 520	520 (468/52)
Experiment 540	540 (486/54)
Experiment 560	560 (504/56)
Experiment 580	580 (522/58)
Experiment 600	600 (540/60)
Experiment 620	620 (558/62)
Experiment 640	640 (576/64)
Experiment 660	660 (594/66)
Experiment 680	680 (612/68)
Experiment 700	700 (630/70)
Experiment 720	720 (648/72)
Experiment 740	740 (666/74)
Experiment 760	760 (684/76)
Experiment 780	780 (702/78)
Experiment 800	800 (720/80)
Experiment 820	820 (738/82)
Experiment 840	840 (756/84)
Experiment 860	860 (774/86)
Experiment 880	880 (792/88)
Experiment 900	900 (810/90)
Experiment 920	920 (828/92)
Experiment 940	940 (846/94)
Experiment 960	960 (864/96)
Experiment 980	980 (882/98)
Experiment 1000	1000 (900/100)

Table 6: Experiment Reference Term for the Protocol Based Environment