Worcester Polytechnic Institute
Digital WPI

Masters Theses (All Theses, All Years) | Electronic Theses and Dissertations

2014-04-30

Leveraging Software-Defined Networking and Virtualization for a One-to-One Client-Server Model

Curtis R. Taylor
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/etd-theses

Repository Citation
Taylor, Curtis R., "*Leveraging Software-Defined Networking and Virtualization for a One-to-One Client-Server Model*" (2014). *Masters Theses (All Theses, All Years)*. 577.
https://digitalcommons.wpi.edu/etd-theses/577

This thesis is brought to you for free and open access by Digital WPI. It has been accepted for inclusion in Masters Theses (All Theses, All Years) by an authorized administrator of Digital WPI. For more information, please contact wpi-etd@wpi.edu.

Leveraging Software-Defined Networking and Virtualization for a One-to-One Client-Server Model

by

Curtis R. Taylor

Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2014

APPROVED:

Professor Craig A. Shue, Major Thesis Advisor

Professor Krishna K. Venkatasubramanian, Thesis Reader

Professor Craig E. Wills, Head of Department

Abstract

Modern computer networks allow server resources to be shared. While this multiplexing is the unsung hero of scalability and performance, the fact that clients are sharing resources and each client's network traffic is transmitted in a larger pool of the total network traffic, poses distinct challenges for security. By adopting multiplexing so broadly, the networking and systems communities have implicitly favored performance over security.

When servers multiplexing clients are compromised, the attack is able to spread by exploiting unsuspecting clients sharing the resource. Drive-by-downloads are an example of an attack where a Web server is compromised and begins distributing malware to connecting clients. As a result of using today's many-to-one client-server network model, current approaches are inadequate at protecting the network and its resources.

We propose a redesign of the modern network infrastructure. Our approach involves moving from the current many-to-one client-server model to a one-to-one client-server model. In redesigning the network, we provide a means of better accountability for traffic between clients and servers. With accountability, we enable the ability to quickly determine which client is responsible for an attack. This allows us to quickly repair the affected entities. To accomplish this accountability, we separate each client's communication into separate flows. A flow is identified by various network features, such as IP addresses and ports. Further, instead of allowing multiple clients to be multiplexed at the same server, we use a technique that allows each client to communicate with a server that is logically separate from all other clients. Accordingly, a server compromise only effects a single client.

We create a one-to-one client-server model using virtualization techniques and OpenFlow, a software-defined network (SDN) protocol. We complete our model in three phases. In the first, we deploy a physical SDN using physical machines and a commodity network switch that supports OpenFlow to gain an initial understanding of SDNs. The next phase involves implementation of Choreographer, a DNS access control mechanism, in a virtualized SDN environment for better scalability over our physical configuration. Finally, we leverage Choreographer to dynamically instantiate a server for each client and create network flows that allow a client to reach the requested server.

Acknowledgements

First and foremost, I would like to express my sincerest gratitude to Professor Craig Shue for being a research advisor, mentor, and friend during my graduate education at WPI. Without him, I would not have begun the journey through graduate school and would not be where I am today. In times of peril, his presence and expertise always helped guide me and is truly appreciated. I will forever be grateful for this life changing experience.

I also would like to thank my thesis reader, Professor Krishna Venkatasubramanian. His time and effort has given me opportunities and experiences that I would not have otherwise had. His perspective has also helped to provide me with a better understanding of my research.

Both the PEDS and ALAS research groups have provided useful and intellectually stimulating information and feedback. Our regular meetings have helped cultivate a truly beneficial and unique learning experience.

Without my parents, I would not be where or who I am today. They have always provided a positive attitude and support throughout all my endeavors. Finally, I would like to thank my best friend and fiancé for continuing to give me unconditional love and support, even though we are a great distance apart.

Contents

1 Introduction **6**

2 Background and Related Work **8**
2.1 Virtualization . . . 8
2.2 Software-Defined Networking . . . 10
2.2.1 OpenFlow . . . 10
2.2.2 OpenFlow Components . . . 10
2.2.3 OpenFlow Security . . . 11
2.3 DNS Based Access Control . . . 11

3 Phase 1: OpenFlow on Physical Switches **13**
3.1 Introduction . . . 13
3.2 Implementation . . . 13
3.2.1 Configuring an OpenFlow Switch . . . 14
3.2.2 Configuring the OpenFlow Controller . . . 16
3.2.3 Client-side Preparations . . . 17
3.2.4 Developing POX Applications . . . 17
3.3 Verifying Functionality . . . 18
3.4 Discussion . . . 19
3.5 Conclusion . . . 20

4 Phase 2: Choreographer to SDN **20**
4.1 Introduction . . . 20
4.1.1 External Choreographer . . . 21
4.1.2 Internal Choreographer . . . 21
4.2 Implementation . . . 23
4.2.1 Mininet Overview . . . 24
4.2.2 Mininet Configuration . . . 24
4.2.3 Initial Switch Configuration . . . 26
4.2.4 External Choreographer . . . 26
4.2.5 Internal Choreographer . . . 31
4.3 Performance . . . 32
4.3.1 DNS Overheads . . . 32
4.3.2 Connection Setup Overheads . . . 33
4.3.3 Transmission Overheads . . . 33
4.4 Discussion . . . 34
4.4.1 Redundant TCP Connections . . . 34
4.4.2 Redundant DNS Requests . . . 34
4.4.3 Matching Client Source Port . . . 34
4.4.4 Flow Timeouts . . . 35
4.4.5 IPv6 Support . . . 35
4.4.6 Controller DNS Functionality . . . 35
4.4.7 Enabling Direct IP Communication . . . 35
4.5 Conclusion . . . 35

5 Phase 3: Creating a One-to-One Model **36**
5.1 Introduction . . . 36
5.2 Implementation . . . 37
5.2.1 Mininet Alterations . . . 37
5.2.2 Controller Modifications . . . 38

5.3 Performance . . . 41
5.3.1 Scalability . . . 41
5.4 Discussion . . . 42
5.4.1 Performance Considerations . . . 42
5.4.2 Security Considerations . . . 43
5.5 Conclusion . . . 44

6 Future Work **44**
6.1 Virtualization Adjustments . . . 44
6.1.1 A KVM One-to-One Model . . . 44
6.2 OpenFow Controller Implementation . . . 45

7 Conclusion **45**

List of Figures

1 We transform a traditional network (a) to use use separate virtual instances of a server for each client (b). By using flows and individual servers for clients, we can attribute security faults to end-users and examine related traffic. . . . 7
2 Phase 1 network topology . . . 14
3 Our `/etc/config/openflow` configuration file. . . . 16
4 MAC address whitelist and NAT entries . . . 17
5 Mininet architecture with a single host . . . 24
6 Phase 2 network topology. . . . 25
7 Simple Mininet design. . . . 26
8 Configuring the network. . . . 27
9 Directing switch to send all packets to the controller. . . . 28
10 BIND zone file . . . 28
11 Direct a switch to send all DNS packets to the controller. . . . 28
12 Choregrapher DNS interception and modification. . . . 29
13 Choregrapher TCP flow interception and flow approval. . . . 30
14 ARP Proxy for Choreographer in Mininet . . . 31
15 Create a flow for incoming packets from the client to the Web server. . . . 32
16 DNS response times of the baseline in Figure (a) and Choreographer in Figure (b) 33
17 TCP connection times of the baseline in Figure (a) and Choreographer in Figure (b) 34
18 Phase 3 network topology . . . 37
19 Network topology with 10 clients, 10 front-end servers and a back-end server. . . . 38
20 Configuring the network for a one-to-one model. . . . 39
21 DNS interception and modification with controller state. . . . 40
22 TCP interception and flow installation by controller. . . . 40
23 Configuring front-end and back-end flows on attempted TCP connection . . . 41

List of Tables

1 Linksys WRT54GL Router Specifications . . . 14
2 TP-LINK TL-WR1043ND Router Specifications . . . 15
3 OpenFlow switch flow entry table fields . . . 16
4 Header fields for MAC whitelisting . . . 18
5 Actions table for MAC whitelisting . . . 18
6 Header fields for NAT . . . 18
7 Actions table for NAT . . . 18

8 DNS response statistics . . . 33
9 TCP response statistics . . . 33
10 DNS response statistics with 10 simultaneous requests . . . 42
11 TCP response statistics with 10 simultaneous requests . . . 42

1 Introduction

Modern computer networks often have two broad classes of servers: user-facing "front-end" servers, which are rich in functionality and features, and support servers that provide low-level resource management (often called "back-end" servers). Front-end servers often support a variety of clients with diverse workflows and needs. By multiplexing requests from clients, these servers can share resources, benefit from caching efficiencies, and economically scale services to thousands of users. While this multiplexing is the unsung hero of scalability and performance, the fact that clients are sharing resources and each client's network traffic is transmitted in larger pool of the total network traffic poses distinct challenges for security. By adopting multiplexing so broadly, the networking and systems communities have implicitly favored performance over security.

Figure 1(a) presents an example of today's network infrastructure and shows how security becomes difficult to manage. A single Web server handles numerous clients simultaneously. Because clients are multiplexed, situational awareness within the network becomes difficult. The difficulty arrises from the inability to distinctly view information about what a particular client is communicating. This also makes attribution a nontrivial task. While there are best practices in place to help secure networks, there will always be faults in the system, giving attackers the advantage. After a machine is compromised, trying to uncovered how the attack took place and exactly what parts of the machine were infected can be a hard-fought process. We need a better approach at fixing these issues than exists today.

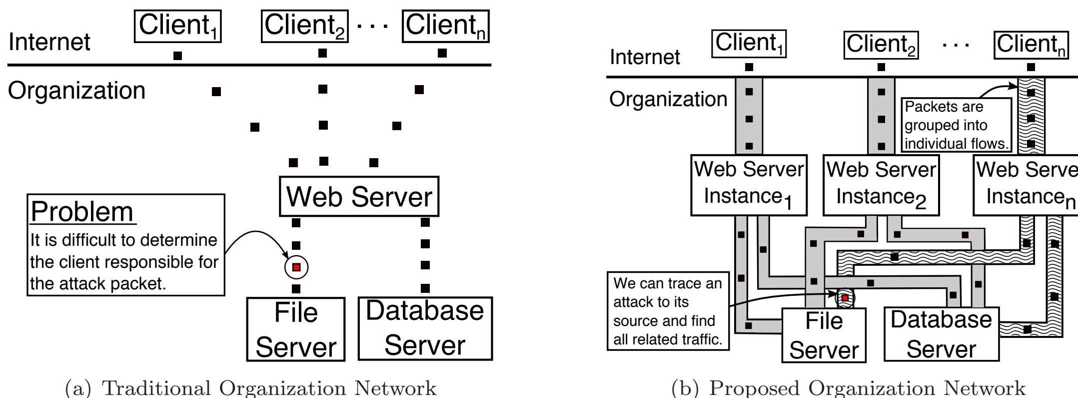
We propose a redesign of the modern network infrastructure. Our approach involves moving from the current many-to-one client-server model to a *one-to-one* client-server model. In redesigning the network, we provide better accountability for traffic between clients and servers. With the accountability, we are able to quickly determine which client is responsible for an attack. This allows us to quickly assuage the affected entities.

To achieve this accountability, we separate each client's communication into separate *flows*. A flow is identified by various network features, such as IP addresses and ports. Rather than allow multiple clients to be multiplexed at the same server, we use a technique that allows each client to communicate with a server that is logically separate from all other clients. We are able to dynamically create this server as well as destroy it when it is no longer needed.

When clients are able to compromise front-end servers, it is also difficult remedy the issue quickly without affecting other clients. This paradigm results in being able to stop any ongoing attacks immediately and prevent any persistent attacks. If we choose not to destroy a compromised server, our approach also allows us to forensically analyze an attack. In essence, the attacker aids us in determining how the server was compromised.

By both separating client communication into separate flows and allowing each client to connect to a separate server, we achieve our desired one-to-one network model. The two technologies that allow us to do this are Software-Defined Networking (SDN) and Virtualization. Although virtualization and SDN have been considered in other work, our work explores a combination of the two areas. We depict how using these two ideas together helps us achieve our one-to-one model in Figure 1(b). SDN promises to facilitate more efficient network management, and using a controller, will enable remote management of network switches. Virtualization on the other

hand provides a means of isolating software execution within a single host. The combination of agile networking switches and dynamic virtualization enables us to demultiplex servers and provide clients with a one-to-one relationship where each client connects to a separate server.



(a) Traditional Organization Network

(b) Proposed Organization Network

Figure 1: We transform a traditional network (a) to use use separate virtual instances of a server for each client (b). By using flows and individual servers for clients, we can attribute security faults to end-users and examine related traffic.

The realization of a one-to-one model, as shown in Figure 1(b), happens in a series of 3 Phases.

- **Phase 1 - OpenFlow on Physical Switches**: The first phase encompasses the initial installation and configuration of a physical SDN, including a controller, as well as a switch that supports the OpenFlow protocol. As we deploy this network, we describe how to program the controller to direct the switch's behavior. Specifically, we using OpenFlow to implement a MAC address whitelist and to perform Network Address Translation (NAT). Using a single controller and switch with multiple clients connected provides an ideal starting point because it is a simple network topology compared to our overall goal of a dynamic and fluid topology.

  There are details that a virtualized SDN environment abstracts away. By completing this phase, we have a better understanding of SDNs and are able to program a function SDN on physical hardware. The logic we develop for this phase is then brought to the next phase of our work.

- **Phase 2 - Choreographer to SDN**: Phase 2 involves transitioning our previous phase to a virtualized environment for better scalability. We then implement Choreographer, which is a DNS access control mechanism [56]. After implementing the current Choreographer approach, we make additional improvements that allow Choreographer to run internal to a network and adds additional security benefits.

  Implementing Choreographer in a SDN solves some issues with the current implementation. The virtualized environment used to complete this phase guides us into building a one-to-one model in the same framework. Choreographer in a SDN also enables a mechanism for dynamically connecting clients to the appropriate servers.

- **Phase 3 - Creating a One-to-One Model**: In this final phase of work, we move towards creating a one-to-one client-server infrastructure. For each client connecting to our network,

we connect the client to a new front-end virtual server. We use Choreographer and DNS to passively discover which services clients are requesting. In our virtualized SDN, we are able to scale the number of clients and servers well beyond our physical setup, and the result is a fully functional one-to-one model.

By completing this p hase, we have a systematic process of determining which services clients are requesting. When a connection is attempted by a client, we are able to create network flows allowing only a single client to access a specific server. This completes our goal of implementing of a one-to-one client-server model.

2 Background and Related Work

Our networking model relies on SDN, virtualization, and DNS. We now provide some background and related research in these areas.

2.1 Virtualization

Virtualization has been around since the 70's [46], but due to hardware restrictions, its widespread adoption is a much more recent phenomenon. Garfinkel *et al.* [25] discussed how the dynamic nature VMs machines could potentially pose problems to an organization whose employees could start a VM, which may contain malware, and have that malware spread to other hosts on the network before being able to isolate the infected client. In later work, Garfinkel *et al.* [26] noted many benefits of using VMs for security, as opposed to security issues they may cause. This work described how VMs are used for isolation between running processes on a single machine for security reasons. If an application running in one VM is compromised, it is isolated from other applications on other VMs. Other advantages offered by virtualization include between logging, the ability to take snapshots, and forensics after a VM is compromised. The notion of being able to dynamically start VMs is partly what a one-to-one client-server model will leverage. Virtual machine introspection (VMI), introduced by Garfinkel *et al.* [24], is the process in which a host running VMs can look inside a VM to check its state. In many cases, when machines are infected with malicious software, the malware will attempt to cover its tracks and erase logging data or remove files that might reveal its presence. VMI allows a host machine to not only check VM state but do so undetected. Later work details approaches to VMI [52], [33]. This area of research suggests that if clients were able to communicate with individual VMs, VMI could be used to detect clients that are attempting to infect servers. This approach is in contrast to detecting when a piece of software server many clients has been compromised.

In commodity systems there are several approaches that a server may use to enable virtualization. The two most popular in the open source and research community are Xen [7] and a combination of Kernel-Based Virtual Machine (KVM) [28] and QuickEMUlator (QEMU) [5]. The main difference in these virtualization techniques is their architecture. In Xen's architecture the Virtual Machine Manager (VMM) or *hypervisor*, which is the software that manages running VMs, sits directly on hardware and controls access to the physical hardware. The user's management console, called dom0 (short for domain0), is the "secured" VM running on this architecture. For

security, domain0 allows a user to monitor VMs by intercepting system calls. This approach is limited to using operating systems that have been modified to allow for these hooks.

KVM's architecture is different in that it is an addition to the kernel via a kernel module. In order to run KVM, processor support such as Intel's VT-x extensions are required, and most newer Intel processors have these extensions [6]. Furthermore, KVM also requires QEMU to be installed for emulating hardware. The KVM kernel module allows access to the processor extensions in order to more quickly emulate the x86 architecture. QEMU is the software that actually runs the emulation of hardware. It is possible to solely run QEMU and emulate all hardware in software, but this approach incurs significant performance degradation.

Virtualization is also a popular technique for creating honeypots. Honeypots are made to be deceptive and act as a decoy to would-be attackers. Honeypots can divert attackers away from real production servers while educating organizations about vulnerabilities in their systems, without risking loss of private data or down time [42]. Honeypots are also deployed as a solution to detect insider threats [58]. We recognize the benefits of honeypots and use them in our work as a "sink" for malicious clients.

Vrable *et al.* also recognized the benefits of honeypots but saw that for wide-scale deployment, virtualization can be costly [64]. Their work introduces two important techniques, *flash cloning* and *delta virtualization*. Combined, these techniques help rapidly spawn honeypot VMs. Flash cloning copies an already running honeypot VM to a new instantiation. By cloning, the overhead of boot and application load times reduces. Delta virtualization optimizes a flash clone using copy-on-write. Copy-on-write is beneficial since virtual machines share a lot of the same memory. Thus, not having to copy memory until it is modified by a VM speeds up the process of copying. This work, along with virtualization techniques that leverage the cloud [40], [39], helps to reduce the performance concerns of our approach.

Parno *et al.* investigated large-scale data leak prevention [51]. Their work indirectly looks at implementing a one-to-one model for Web server interactions and used delta virtualization in the process. The goal was to prevent an attacker from stealing other client's data if the attacker managed to compromise a Web server. Our work considers the same problem but takes a larger aim at the problems caused by multiplexing clients at a server, such as being able to easily tell which client is acting maliciously. We consider a one-to-one client-server model that uses software-defined networking. Importantly, our work is not focusing on what a one-to-one system model should be. We only consider the model from a networks perspective that uses software-defined networking.

We have presented background on virtualization to show its stronghold in modern research, support, and increased efficiency. In essence, this work partly relies on the success of virtual's security and performance benefits, but virtualization is just one key element of the one-to-one model. The other side of this approach is developing a networking infrastructure that allows virtualization to be thoroughly utilized. We believe software-defining networking to be the answer to creating this infrastructure, and the remainder of this work focuses on the networking aspects associated with this vision.

2.2 Software-Defined Networking

Software-defined networking is an approach that allows the separation of the control plane and data plane in networking switches. In modern, unmanaged switches, a user has no control over the logic of the switch. As an example, consider a commodity unmanaged switch connecting machines at a university. Unfortunately, after these switches are deployed throughout the network, administrators do not have the ability control or alter the logic. Moreover, switches that do allow limited control, such as Cisco's Catalyst switches, require physical access and a special adapter for configuration and are limited to the functionality provided by proprietary software.

However, if vendors were able to decouple the control and data plane, consumers would still be able to use manufacturers' hardware and have the ability to implement their own control software. The restrictions of modern switches has already been recognized by cloud providers and data centers. Accordingly, operators of these networks are using SDN to allow operators to quickly reconfigure the network or provision resources based on dynamic network conditions [35].

2.2.1 OpenFlow

SDN itself is a networking paradigm that separates control plane from the data plane. A SDN protocol is needed for the two components to communicate. OpenFlow is one such communication protocol [44]. OpenFlow is an open source protocol that may be compared to other commercial SDN solutions such as Juniper's QFabric product [49]. Although Juniper has implemented their own SDN solution, many other manufacturers have built switches supporting the OpenFlow protocol (IBM [31], HP [29]). OpenFlow has gained attention for its capability of seamlessly controlling Quality of Service (QoS) and load balancing [63] improving performance, reducing cost, and providing security benefits [30].

2.2.2 OpenFlow Components

In order to take advantage of the OpenFlow protocol, two different components are needed. A SDN using OpenFlow requires at least a *controller* and a *switch* supporting the protocol. Due to OpenFlow's popularity, multiple controller applications [4] [48] [62] [36] exist. Different controllers support different programming languages and some have better performance. Similarly, both physical [29] [31] and virtual [53] switches exist. While physical switches may be costly to deploy, Open vSwitch [53] allows seamless integration into virtual machine infrastructures (e.g., KVM). While OpenFlow was originally created to be extremely simple and the approach is often represented as using a centralized controller, other research looks at creating distributed controllers [62] [36] to provide load balancing and redundancy in the event the controller were to fail. These controllers allow physically distributed but logically centralized controllers. Our research does not leverage distributed controllers, but we recognize any logic built on top of a centralized controller can be migrated to a distributed platform for better reliability and scalability.

Mininet is a network prototyping software that we leverage for building our model. Mininet was designed with several goals in mind. The authors wanted 1) flexibility for dynamic topologies, 2) deployability so that no changes were required to deploy in a real network, 3) interactive for things

to happen in real time, 4) realistic meaning the behavior was not a result of virtualization, and 5) share-able to allow collaborators to easily run and modify code developed using the software [37]. Mininet recognizes the overhead of using full virtualization (VMs) for prototyping and wanted to allow a "network in a laptop" solution. Using Linux containers [3], Mininet has the potential to scale to thousands of hosts on a single machine. Containers allow for different network namespaces to isolate client processes. Containers also allow "lightweight" virtualization and increases scalability. The Mininet architecture allows users to create networks consisting of hosts, switches, SDN controllers, and links to connect these components. We leverage this approach to build our network topology.

2.2.3 OpenFlow Security

Since we are using a relatively new networking paradigm and implementation, we must consider security risks associated with the approach itself. In particular, Benton *et al.* [16] address security concerns associated with the OpenFlow protocol. Since being released, OpenFlow has changed the requirement on TLS between the controller and switch. TLS was originally enforced between all switches and the controller. The TLS requirement has since been removed. A consequence of this change is that the communication link between the devices is subject to man-in-the-middle attacks (MITM). MITM attacks on OpenFlow can create unauthorized rules at switches, causing traffic to be redirected to an adversaries machine. Another consequence is that some controllers do not provide support for TLS. Only recently did the POX controller gain TLS support [12].

While security concerns are important, OpenFlow is well supported, not only by the open source community but by vendors as well, and will continue to improve the security of OpenFlow. Our work takes the SDN concept uses its ability to create custom software to dynamically control network switches.

2.3 DNS Based Access Control

The Domain Name System (DNS) is an application layer protocol that helps users navigate the Internet. In simple terms, DNS is the process which turns a human readable hostname into an IP address. Clients typically issue DNS requests through their browser. Although browsers may cache the IP address returned for an extended amount of time, browsers typically follows the DNS protocol and issue a new DNS request after the Time-to-Live (TTL) of the domain name to IP address mapping expires. As first noted by Shue *et al.* [57], malicious clients typically do not follow the protocol and may continue using an IP address after its TTL has expired. In this way, the DNS protocol can be used for client-based access control. In particular, if a client does not issue a new DNS request after the TTL has expired, the server IP address can be moved, and the client may not know of the change. We created an implementation of this approach called Choreographer*et al.* [60].

Our approach relies on IP address randomization. Each time a client issues a DNS request, a DNS response is provided with a random IP address. An application running on the DNS server notices a DNS request was made, generates a new random IP address for the next client, and updates the the DNS zone file of the DNS server.

Although the client receives a random IP address, the Choreographer system creates a Network Address Translation (NAT) rule that transparently converts the random, global IP address into a static, private IP address. If a client attempts to access an IP address that does not have a corresponding NAT rule, the client is directed to a special server called a honeypot. The original NAT rule allows anyone to use the random IP address. This is a necessity for the approach because the application running alongside the DNS server does now know the IP address of the client issuing the request. All that may be seen is the IP address of the client's DNS resolver, which is typically provided be the Internet Service Provider (ISP). Choreographer leaves the NAT mapping allowing anyone to use a particular random IP address available for the duration of the TTL of a response. For this reason, the TTL should be kept as minimal as possible.

If a client attempts a connection to the random IP within the timeframe of the TTL, another NAT mapping containing the client's IP address and the random IP address are added. This allows that client to continue using the IP address for an extended amount of time without worrying about the mapping expiring.

Authoritative DNS servers do not know the IP of the client issuing a DNS request. This information would be particular useful for preemptively creating flows. If this information was available, the initial NAT mapping would map a single client to the IP address returned by the DNS server and avoid a client using the address that did not make a legitimate request. Mao *et al.* [41] recognized this issue and attempted a solution which caused clients connecting to a Web server to issue a specially crafted DNS request encoded in JavaScript that uniquely identified the client. While this approach had promising results, it is limited in its application. Another solution to solve the issue was proposed by Shue *et al.* [55]. Their approach attempted to passively link clients to their resolvers based on the characteristic that clients typically issue requests based on DNS responses very quickly.

A later approach by Jafarian *et al.* [32] attempts to transition Choreographer's approach into a SDN environment using OpenFlow. They focus on preventing scanning attacks and worm propagation. Their approach uses the NOX [27] OpenFlow controller and Mininet to create virtual clients and workloads. While their approach lacks details on the implementation, we believe they use an artificial DNS server in which the NOX controller captured DNS requests and returned custom responses with a randomly-generated IP address. They refer to this process as Random Host Mutation. The controller then issues OpenFlow commands to insert flows into each of the switches to allow the NAT translation from the randomly generated public IP address to the internal static IP address. This approach has a few limitations. First, they do not use an actual DNS implementation such as BIND9, which many organizations use. They also do not attempt to link a particular client to the random generated IP addresses. Instead, they assume the first person connecting to IP address is the person that issued the request. Unfortunately, this may not be the case. When creating flows for clients, they only the source and destination IP addresses. As a result, a worm could check to see which IP addresses a machine is connected to and attempt to spread. Further, any worm or on-going scanning attacks can connect to an IP address before the TTL expires. They have no way of determining if the client establishing the connection is the same client that issued the request. Last, they do not address what happens when a client (possibly a

worm) attempts to access an IP address that is not active. This information would be useful for determining when a client is infected.

3 Phase 1: OpenFlow on Physical Switches

This phase describes the installation and configuration of a physical SDN, including a controller as well as a switch that supports the OpenFlow protocol. We find the configuration of clients to be trivial as clients should not be involved in the network configuration. The controller requires more effort. However, we discover that the switch installation and configuration requires the most effort. Accordingly, we discuss our efforts on both the controller and switch in detail.

Once the network infrastructure is in place, we begin programming the controller to direct the switch's behavior. We use a single controller and a single switch with multiple clients connected because it provides an ideal starting point. We later extend this simple network topology into our overall goal of a dynamic one-to-one topology.

3.1 Introduction

We begin the process of building a basic SDN network using physical devices. Having little prior exposure to SDNs, this phase allows us to configure a real SDN and become familiar with the concepts of OpenFlow and controlling the data plane. In some situations, assumptions may be made about virtual environments that may not be true in a real environment. For example, performance results in a virtualized setting cannot be directly compared to performance results of a physical network. In a virtualized setting, packets are processed extremely quickly because packets need only to be moved through random access memory (RAM). Without special considerations, transfer speeds across RAM do to represent actual transmission speeds. In practice, packets physically traverse long distances and encounter unexpected delays. A virtualized environment also abstracts important details. For example, Mininet abstracts all details about the switch, such as which port the controller is connected to, from the programmer. This detail may not be important for building and testing a network in a virtual environment, but if the system designer attempted to transition the work to a real network, this detail is extremely important. Without connection to the proper work, the SDN would not function.

For this Phase, we use the following components: (3) physical hosts, (1) physical controller, and (1) physical switch. These components and how they are interconnected are shown in Figure 2.

3.2 Implementation

With our topology configured as that of Figure 2, we must first configure our switch to be OpenFlow compatible. After configuring the switch, we install the POX SDN and develop our logic. Although there are several SDN controller implementations, POX is written in Python, which is well supported and allows rapid prototyping. POX also does not require compilation when making changes to the code base. Last, we connect clients to our switch and test our implemented logic.

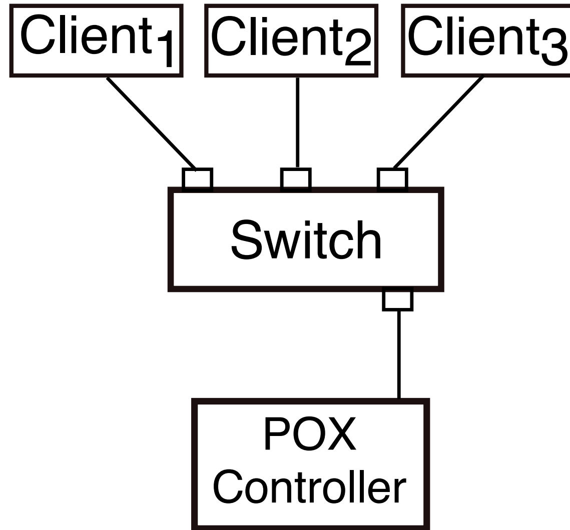


Figure 2: Phase 1 network topology

3.2.1 Configuring an OpenFlow Switch

As OpenFlow has grown in popularity, the support for OpenFlow on commodity switches and routers has also increased. Consumer grade routers such as Linksys's WRT54G series and more recently, TP-LINK's TL-WR1043ND router, support multiple specifications of OpenFlow. However, the routers' default firmware does not immediately support OpenFlow. In order to support the protocol, the router's firmware must be updated with a specific branch of the popular OpenWrt [10] firmware called Pantou [11].

Consumer routers operate on layers 2 through 4 of the OSI model. Accordingly, these routers also have built-in switching functionality. Although OpenFlow runs on switches, it can manipulate information on any of the 3 layers as well as making routing decisions internal to the switch. For implementation, we chose to purchase a Linksys WRT54GL router and flash it with a version of Pantou supporting OpenFlow specification 1.0. We chose this router over others because it provides 4 basic 100 Mbps switched Ethernet ports in addition to one uplink (WAN) port. With this port count, we can support our specified number of clients. Another consideration for choosing the router is the processor and memory constraints of the device. The constraints for both routers are shown in Table 1 and Table 2.

CPU	RAM	Flash Memory	Ethernet Speed	Wireless Speed
Broadcom @ 200 MHz	16 MB	4 MB	100 Mbps	54 Mbps

Table 1: Linksys WRT54GL Router Specifications

CPU, RAM, and flash memory are all constraints that need careful consideration. During this phase, we do not push these constraints to limits, but we will briefly mention how each aspect may affect a SDN.

Version	CPU	RAM	Flash Memory	Ethernet Speed	Wireless Speed
1.x	Atheros @ 400MHz	32 MB	8 MB	1000 Mbps	450 Mbps
2.x	Qualcomm @ 720MHz	64 MB	8 MB	1000 Mbps	450 Mbps

Table 2: TP-LINK TL-WR1043ND Router Specifications

3.2.1.1 CPU Considerations Like many consumer grade routers, the number of clients connected can expected to be lower than those in industrial usage. Accordingly, the CPU is more limited in its processing ability. Depending on how a network is configured, the amount of communication between the controller and the router will vary. If the amount of communication is high, for example, sending all packets to the controller for decisions, the CPU becomes highly utilized. This is why the switch is able to install rules locally to avoid having to send traffic through the controller. Even then, CPU will still be doing the processing but will not have the additional overhead of controller communication.

3.2.1.2 RAM Considerations RAM is extremely important to OpenFlow's performance. For a SDN to work efficiently, the controller tries to store rules on the switch so future packets matching the same rule can have actions taken locally rather than having to elevate the packet. The number of rules that can be installed on the switch is limited by the size of its RAM. For higher grade routers, the constraint changes from standard RAM to a much quicker and expensive memory type called Ternary Content-Addressable Memory (TCAM). TCAM memory allows for checking the entire memory storage for rules simultaneously.

3.2.1.3 Flash Memory Considerations Finally, flash memory limits the size of the firmware on the device. Flash memory can possibly limit the OpenFlow specification as it continues to grow and evolve if the device is incapable of storing the necessary logic. A more obvious concern for consumers is that the WRT54GL router only supports wireless speeds up to 54 Mbps (802.11g), but because we do not focus on wireless, this was not a concern for us. It should be noted that there is no difference between being wired or wireless in this work. Fundamentally, the approach is agnostic to the knowledge of the client being physically connected through an Ethernet port or wirelessly communicating.

Using the Pantou firmware, we flash the Linksys WRT54GL router. The firmware binary is around 3 MB, which leaves plenty of room for other storage if necessary. The uplink port provides direct access to the router's firmware using SSH or Telnet from a connected machine. We must configure the device before being able to use it as an OpenFlow switch. There are three files that require attention:

1. `/etc/config/network`: Configure the router's IP address on the WAN port for external access and communication to the controller.

2. `/etc/config/wireless`: Configure the router's wireless settings such as wireless channel. This file is ignored for our purposes.

```
config 'ofswitch'
     option 'dp' 'dp0'
     option 'ofports' 'eth0.0 eth0.1 eth0.2 eth0.3'
     option 'ofctl' 'tcp:192.168.1.10:6633'
     option 'mode'  'outofband'
     option 'ipaddr' '192.168.1.1'
     option 'netmask' '255.255.255.0'
     option 'gateway' '192.168.1.1
```

Figure 3: Our `/etc/config/openflow` configuration file.

3. `/etc/config/openflow`: OpenFlow configuration specifics. Our particular configuration is given in Figure 3.

The configuration provided in Figure 3 provides the switch with the IP address and port of the controller (192.168.1.10:6633). When the switch boots, it uses its IP address (192.168.1.1) and this information to establish a connection. It also requires each port on the switch to be added to a list of OpenFlow capable ports. We also name the data path where rules are stored as dp0. After making changes to these files, the switch must be rebooted before it is ready.

When rules are stored in the switches data path in a flow entry table structure. The flow entry table has 3 fields as shown in Table 3. Header fields contains match information that is checked against every packet traversing the switch. For example, this could include a check if the destination port is 80. OpenFlow specification 1.0 has a total of 12 fields that can be checked [23]. The counters field maintains information such as the number of packets matching an entry, the number of bytes transmitted using that entry, and how long the entry has been active. Finally, the actions field specifies what actions should be taken if a given match occurs. Example actions include NAT, which physical port to send a packet out, and modifying MAC addresses.

Header Fields	Counters	Actions

Table 3: OpenFlow switch flow entry table fields

3.2.2 Configuring the OpenFlow Controller

With the WRT54GL router configured, we then install and configure the controller on one of our four machines and configure its IP address to be `192.168.1.10`, which the switch will attempt to connect to. For an OpenFlow controller, we use a Python based controller called POX [4]. After download and installation and before we begin developing software for the controller, we install an OpenFlow Wireshark packet dissector. The dissector allows us to view OpenFlow packets going to and from the controller and helps us understand and debug our code.

To familiarize ourselves with the POX controller, this phase implements two basic, but fundamental, network control applications. The first application is building a MAC address whitelist application (layer 2). The other application designed for Phase 1 enables configuration of NAT from one IP address to another (layer 3). We will now provide implementation specifics on both applications.

```
# MAC address whitelist
id, mac
1, 00:11:22:33:44:55
2, 99:88:77:66:55:44

# NAT ip_dst to ip_nat
id, ip_dst, ip_nat
1, 192.168.1.2, 192.168.1.200
```

Figure 4: MAC address whitelist and NAT entries

MAC address whitelisting allows a network administrator to explicitly allow communication between two hosts with particular MAC addresses. Consumer grade routers usually provide this functionality. The benefit of whitelisting is that if it is known in advance all clients that may connect the network, a whitelist can be created so no unwanted clients can connect. If two hosts attempt to communicate without their addresses given in the whitelist, their communication is blocked. To implement this, we use first create a whitelist in the format of Figure 4. The `id` is a unique value to maintain the number of entries in the list but is not necessary for a whitelist. The other two entries (`mac_1` and `mac_2`) are the MAC addresses of two clients which are allowed to communicate. The order of the addresses does not matter. Similarly, we require an input file for addresses participating in NAT also shown in Figure 4.

With these two files created where the controller has access, it can then program the switch to enforce them.

3.2.3 Client-side Preparations

Before beginning to develop applications to run on the controller, we configure each client on our network. Because a SDN involves only configuring the networking equipment connecting clients, no client-side modifications are necessary. Most consumer routers provide a built-in DHCP server, but rather than setting up a DHCP server, we configured each client to have a static IP address, all within the same subnet (`192.168.1.0/24`). The lack of configuring a DHCP server has no effect on our work.

3.2.4 Developing POX Applications

When a switch first connects to the controller, information is exchanged such that the controller knows the capabilities of the switch (i.e., highest specification supported). After this exchange, the controller has the ability to *proactively* configure the switch. An example proactive rule might be to configure the switch to forward all traffic destined to port 53 out a particular physical port on the switch known to be connected to the DNS server. Another example might be that an organization that wants to block all SSH traffic could choose to drop all traffic on port 22. Proactive rules prevent the switch from having to initially forward packets to the controller. In our first phase, we use a proactive solution and install rules for MAC address whitelisting and NAT.

First, we discuss MAC address whitelisting. Looping through the list as shown in Figure 4, we add a *rule* to the switch for each entry. The rule looks for traffic coming from from `mac` and allows

it to to be forwarded. In this context, a *rule* consists of two parts, namely a *match* and an *action*. Table 5 shows the flow entry table after each entry in Figure 4 is added. The flow table contains a series of matches that must be met before the action can be taken. A wildcard (*) indicates that anything can be in the respective field.

Ingress Port	Ether src	Ether dst	...	IP src	IP dst	Action#
*	00:11:22:33:44:55	*	...	*	*	1
*	99:88:77:66:55:44	*	...	*	*	1
*	*	*	...	*	*	2

Table 4: Header fields for MAC whitelisting

Action#	Action to perform
1	Accept()
2	Drop()

Table 5: Actions table for MAC whitelisting

Using the information in the flow table, any packet arriving at the switch that does not match the first two rules is dropped. If it does match a rule, the switch forwards the packet onward. Thus, we have achieved our MAC address whitelist.

As with the MAC whitelist, configuring the switch to implement NAT requires iterating through the list in Table 4. For each entry, we add two rules consisting of a match and action. Assuming a previously empty table, the resulting flow entry table on the switch is shown in Table 7.

Ingress Port	Ether src	Ether dst	...	IP src	IP dst	Action#
*	*	*	...	*	192.168.1.2	1
*	*	*	...	192.168.1.200	*	2

Table 6: Header fields for NAT

Action#	Action to perform
1	mod_dst_ip=192.168.1.200
2	mod_src_ip=192.168.1.2

Table 7: Actions table for NAT

As shown in Table 7, any packet with a destination IP address of 192.168.1.2 or a source address of 192.168.1.200 is altered. For this approach to be successful, it assumes there is exist a client with the IP address of 192.168.1.200. If only one of these rules were in place, traffic would only be successful in one direction and not the other.

3.3 Verifying Functionality

After developing our controller software, we ran tests to ensure the switch functions as we intend. Testing is conducted using two separate passes, one for MAC whitelisting and one for NAT. This is because our controller was not programmed to simultaneously implement a whitelist and NAT. Therefore, we must ensure each operates correctly, but when executed separately.

To test our MAC address whitelist approach, we provide a whitelist with one entry. The entry is set to only allow two of our three clients to communicate. The resulting flow entry table should look similar to Table 5 except with the MAC addresses reflecting our clients. Recall these flows are installed when the switch connects to the controller in a proactive manner. We can check to see whether the switch installed the flows correctly by directly probing the switch and retrieving all active flows. Retrieving flow statistics from the switch is accomplished by first connecting to the switch and then using the `dpctl` executable to dump all flow statistics. Seeing the flows exist, we also manually test if we can communicate between the hosts by running the `ping` command and Wireshark on the clients. The whitelist approach is determined to be successful if the `ping` command succeeds between the whitelisted clients, and traffic destined to or originating from the third client fails. Finally, we again dump the flows on the switch again and verify the statistics match what we expect. This ensures that rules we added were in fact the rules used when experimenting.

Next, we want to verify our NAT implementation. Again, we are able to directly probe the switch after connection to the controller to see if the expected flow table entries exist. Our switch should have entries similar to Table 7 if using the NAT entry in Figure 4. Using `ping` and Wireshark, we verify that when traffic is sent to `192.168.1.2` that the destination is modified to `192.168.1.200`, and also in the reverse direction, the source IP address is modified. Further, by allowing one of our client machines to have the IP address `192.168.1.200`, we see the translation is successful if the ping successfully returns.

3.4 Discussion

We developed and tested two SDN applications using the POX controller. Both the whitelist and NAT applications worked as expected. While these applications are very limited in scope, the concepts are relied on in the remainder of our work.

Even with a limited scope, we still saw one problem that we did not consider during development but noticed during testing. Table 5 shows that we only filter based on the source MAC address. This means data can actually be sent to a MAC address that is not in the whitelist. Potentially, authorized MACs can send broadcast traffic, and the traffic would go unnoticed. This introduces a method for data to be leaked. Steganography techniques allows type of message passing by embedding hidden information within a file or other data. This type of vulnerability for data leakage is a recognized problem [1] [51].

The issue of whitelisted clients leaking information can be prevented in our scenario. There are two viable solutions, but both require the controller to intercept all ARP broadcast packets, which may not be desirable for performance reasons. The first solution requires the controller to know in advance which physical port connects a particular client. Using this information, the controller can essentially unicast the broadcast message out of that specific port, assuming the destination is also whitelisted. The ARP response would then be a unicast between two whitelisted clients and thus allowed by the switch. The would be problematic if machines are allowed to switch ports. Wireless communication also presents a challenge because anyone nearby can listen to wireless communication. The other method is to use an approach known as ARP proxy [19]. We will

discuss this method more during Phase 2.

Another consideration that arises from our implementation of this phase is that we did not combine whitelisting with NAT. Indeed, these two applications are not mutually exclusive and can naturally work in unison. However, for our purpose of Phase 1, combining the two ideas has no added benefit. Rather, it is simpler in terms of implementation and conveying the ideas to have them separate.

Performance concerns have been loosely mentioned throughout this section, but due to this phase's limited functionality, we defer performance considerations to the remaining phases.

3.5 Conclusion

In this phase, we have shown an example physical SDN deployment. Further, we wrote two fundamental networking applications for the POX controller. We use the basic functionality of these two applications as they are necessary for more complex networking scenarios in the future. Rather than going directly to a virtualized environment, we chose to implement a physical SDN. In doing this, we also better understand the subtleties that the virtualized environment abstracts away.

4 Phase 2: Choreographer to SDN

Although virtualization abstracts certain details, it becomes a necessity for prototyping and scaling. Rather than continuing to build our physical SDN, we transition our setup into a virtualized environment for ease of development and scalability.

In this phase, we seek to implement Choreographer, a DNS access control mechanism, in a virtualized SDN. We first provide a detailed discussion on the existing Choreographer software and then provide an introduction to the changes we make to this software to support internal client protection.

4.1 Introduction

Choreographer is an access DNS control mechanism. Access control mechanisms attempt to distinguish legitimate from illegitimate users and allow or deny access as a result of this determination [50]. From this point of view, Choreographer is an access control mechanism because it forces clients to use the DNS protocol before access is allowed to a resource. In the same sense, Choreographer is a capability system [15]. The response to an incoming request is a response with an accessible, random IP address. Thus, the capability Choreographer provides to the client is a randomly generated IP address, which has a short TTL associated with it. One misconception about capability systems is that once a capability is given out, it cannot be revoked [47]. However, in our system, once the TTL has expired, the capability is automatically revoked. To revoke the capability, Choreographer waits for a period of time equal to the TTL of the DNS response. After this time has elapsed, Choreographer denies any connections attempts to that particular IP address and directs the connection to a honeypot, thus revoking the capability.

4.1.1 External Choreographer

Choreographer was initially developed focusing on protecting an organization from external threats. The protection was enforced at the organizational boundary, which is typically a router. If an initial incoming request arrives at the router and a NAT entry matching the intended destination IP address, the flow is remapped to a destination honeypot. There are weaknesses associated with Choreographer's implementation. These weaknesses are related to the two different execution modes: centralized and distributed. A shortcoming of the centralized mode is derivative of the fact that the router, NAT device, and DNS server are unlikely to co-located on the same device. To resolve this issue, we developed a distributed mode of operation. This mode allows decoupling of routing, NAT, and DNS. These "agents" can run on different machine as along as they are accessible to one another.

The centralized Choreographer synchronized the NAT and DNS to work together, but in the distributed case, these agents must communicate with each other in order for the system as a whole to work. In the distributed version of Choreographer, when a DNS agent receives a request, it must notify all NAT agents and then send back the DNS response. With this mode, there is no centralization of control or state.

One issue plaguing this approach is that there is now a race condition between the DNS and the NAT agent. Choreographer works in a passive state such that it only sees network traffic after it has been sent. As a consequence, when the DNS server receives a request, the agent running must wait for the outgoing response to be sent to the client. Once this packet is seen, the DNS agent knows what rule each NAT agent should install. Unfortunately, if the delay between seeing the packet, sending the command to the NAT agent, and the NAT agent being able to act on the new rule is too long, the results it the client being directed to the honeypot server. According to recent work by Shue *et al.* [55], clients often use DNS responses immediately upon receiving a response. Currently, no real solutions to handling improperly directed clients (i.e., those sent to the honeypot instead of the corresponding legitimate resource) exists. Solutions to solving this problem include using limited and poorly documented divert sockets or to somehow apply packet redirection so that artificial delay can be introduced. While the later is currently being used to remedy the issue, it can difficult to optimize delays for the best performance. If the delay is not long enough, a client will end up at the honeypot, and if there is too much delay, clients are unnecessarily waiting before being able to access a resource. Another limitation to Choreographer is that regardless of its execution mode, it is reliant upon the organization using a particular, albeit popular, implementation of a DNS server (BIND9) and `iptables` for NAT.

We argue that SDN can solve both the race condition as well as providing an implementation that is DNS and NAT agnostic. Furthermore, by implementing Choreographer in a SDN, we can combine the two modes of execution and once again have a centralized point of control for the software's logic.

4.1.2 Internal Choreographer

An internal Choreographer has the ability to help combat two major security problems. Choreographer can reduce malware spread and reconnaissance inside the network. It can also address

concerns related to the insider threat. Additionally, an internal implementation can reduce IPv4 address exhaustion.

4.1.2.1 Malware Forcing potential external adversaries to use DNS in order to access internal resources has the ability to better protect these resources, but provides no real protection internal to an organization. An internal implementation of Choreographer can protect from the spread of malware they may otherwise force administrators to shutdown entire portions of the network [45].

There are many methods an attacker can use to infect clients within an organization. Occasionally, phishing emails manage to evade detection by filters. Malware installation via drive-by download [43] is also another method. Machines infected with malware often become part of a bonet [34], which can be used to carry out large scale, malicious attacks. A common way botnets increase in size is by IP address scanning looking for other hosts to infect. Malware intelligence has also grow. Instead of randomly scanning IP addresses, newer malware first checks the subnet of the infected host and begins scanning those IP address [59]. This is particularly useful if the malware is trying to evade detection by avoiding IDS systems or firewalls that might be monitoring only at routers, which will not be traversed if the clients are in the same broadcast domain. This notion of infecting an internal machine and attempting to spread locally is known as "pivoting" [9].

Running Choreographer internally may help in addressing the problem of insiders and preventing outbreaks of malware. By forcing internal clients to participate in Choreographer, we again have the notion of handing out capabilities to explicitly allow or deny particular services from being accessed. Previously, Choreographer did not know the IP address of the client because the DNS request is made on behalf of the client's DNS resolver. On the other hand, clients that are asking for internal sources will make requests directly to the organization's authoritative nameserver. This allows us to know the issuing client's IP address and be more strict when creating initial flows. Instead of temporarily allowing a match on any source to a particular IP address, we are able to restrict the initial flow to match the source and destination. The advantages to being able to initially restrict the flow are two fold. First, it ensures the that person being allowed to access a resource is the one that requested it. The other advantage is that we avoid potentially having multiple clients issue requests for a valid IP. This collision becomes more likely as the number of random IP addresses an organization has at its disposal reduces.

4.1.2.2 Insider Threat Another reason to implement Choreographer internally is to protect against an internal client that is malicious. This is also known as the "insider threat" problem. The insider threat problem deals with users internal to an organization abusing their access within the network [17]. For example, a payroll employee may choose to add a fake employee (e.g., a spouse) to the payroll, collect the fake employee's check, and remove the employee from the system. While the payroll employee is in the position to add and remove employees, these particular actions are not necessarily what the organization wants to happen. This type of adversary is not only prevalent today but in some cases, have cost organizations over $500 million dollars [17]. With internal Choreographer, we better understand which services are being accessed by particular clients and can easily deny access to resources when deemed necessary.

4.1.2.3 Direct IP connections and Network Reconnaissance

In some cases, communication internally may not require DNS at all. If there is a particular machine a client connects to often, they may become familiar with the IP address and make direct connections. Intelligent malware may also use system tools, such as `netstat`, to determine the IP addresses a machine is connected to and attempt to also connect. Situations such as these are no longer possible with an internal instantiation of Choreographer. Instead, *hostnames will act as passwords* for machines. If an attacker does not know the hostname or domain name of the destination, it will not have the ability to issue a DNS request and will not be granted the capability to connect. As a result, any attempt to connect to the IP address will result in redirection to the honeypot.

Naturally, malware or even curious users that may want to know the domain name associated with an IP address, and domain names may give information away about the IP address. For example, `sql.example.com` is likely an SQL server. Malware may find this reconnaissance useful in determining effective attack types. A reverse DNS lookup converts an IP back into a domain name. In practice, reverse DNS requests are not that common, which makes them a better indicator for malicious activity. A reverse lookup is done by issuing a PTR record lookup to the DNS server along with the IP address. The DNS server then consults its zone file to determine the domain associated with a particular IP address and returns the domain name. Choreographer thwarts such attempts by allowing the controller to intercept these responses and changing the domain name associated with the IP address. This step is similar to IP address randomization at the controller and requires no modifications to the DNS server. For our purposes, the domain name is replaced with a domain that will always be directed to the honeypot, regardless of valid DNS queries. Although, we could just as easily ignore PTR records.

In the remainder of Phase 2, we describe our SDN solution to both the original Choreographer implementation, and the approach that allows Choreographer to run within an organization. We then provide performance results on the approach and show the overhead is reasonable. Finally, discussion and the conclusion are presented.

4.1.2.4 IPv4 Address Exhaustion

An organization concerned about security and are using Choreographer have the ability to ensure clients are IPv6 compatible. If clients within the network have an IPv6 address, the number of addresses Choreographer has to choose from is virtually unlimited. A large IPv6 address space allows Choreographer to return unique IP addresses for, potentially, years before reuse is necessary. It also means that addresses that might have been used internally can then be given out externally to clients that may not yet support IPv6. This also means that Choreographer will never enter a "fail open" state where all IP addresses are active as a result of a large number of requests.

4.2 Implementation

In this section, we cover the SDN implementation specifics of Choreographer that runs both internally and externally. For scalability, our development environment changes from a physical configuration to a virtual one. We first describe the virtualized environment, Mininet, and then go into implementation specifics and design decisions.

4.2.1 Mininet Overview

Mininet is a network prototyping application that comes conveniently packaged in a preconfigured virtual machine. A detailed overview of Mininet's design is shown in Figure 5. In short, clients and servers in Mininet, represented as "host hX namespaces", are `bash` processes that are isolated using Linux containers. These processes are created and controlled through the `mn` (Mininet) process. Each host is then connected to a switch using a virtual Ethernet pair, one on the client process and one on the switch. The Ethernet port on the switch is also connected to the `ofdatapath` (OpenFlow data path), which is used to quickly process packets on the switch. The data path is controlled by the `ofprotocol` (OpenFlow protocol), which receives commands and communicates with a controller, POX in our case. The `ofdatapath`, `ofprotocol`, and the ports on the switch are collectively considered the Open vSwitch. While the bash processes are isolated, a potential concern is that the filesystem processes are using is shared. Mininet is a development environment, and it is not anticipated that Mininet would be used in a production environment. Since Mininet uses a shared disk, a compromised process would have access to the same files, e.g., web pages, used by trusted processes.

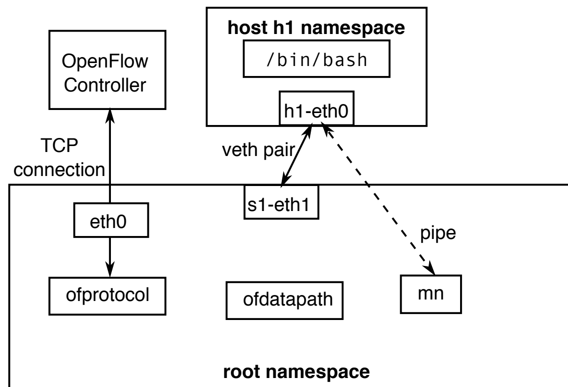


Figure 5: Mininet architecture with a single host

4.2.2 Mininet Configuration

Mininet configuration consists of two different steps. We first create a network topology that lays out the structure, including hosts and physical links to network devices. We then need to configure the hosts connected to the network.

4.2.2.1 Mininet Network Topology Configuration Before we are able to use Mininet, we first need to create a network topology. The network topology must define the following items:

- Host: Any machine connected to the network. This includes clients and servers.
- Switch: Represents the physical switches that will interconnect the network.
- Link: The physical link (or cable) that is connecting devices. A switch is allowed an unlimited number of links.

- Controller: If Mininet needs to provide its own controller, this must be defined.

```
# Add switches
SWITCH1 = self.addSwitch('switch1')
SWITCH2 = self.addSwitch('switch2')

# Add hosts
ROUTER = self.addHost('router')
CLIENT = self.addHost('client1')
DNS_SERVER = self.addHost('dns_server')
HONEYPOT = self.addHost('honeypot')
REALWEB = self.addHost('realweb')

# Add links
self.addLink(CLIENT, SWITCH1)
self.addLink(ROUTER, SWITCH1)
self.addLink(ROUTER, SWITCH2)
self.addLink(DNS_SERVER, SWITCH2)
self.addLink(HONEYPOT, SWITCH2)
self.addLink(REALWEB, SWITCH2)
```

Figure 6: Phase 2 network topology.

Our network topology configuration file is shown in Figure 6. In the configuration above, we did not define the controller. Rather, we want to provide our own controller and not one controlled by Mininet. As a result, when the topology is created it looks similar to Figure 7 where Mininet and the POX controller are running as separate processes on the host operating system. To actually create the topology, we use the `mn` executable:

```
mn --custom Phase2_topology.py --topo router --mac --switch ovsk --controller remote
```

This command creates the topology in Figure 7 and sets MAC addresses to auto-increment for simplicity. It also tells Mininet to use Open vSwitch as opposed to another virtual switch implementation and that the SDN controller is remote. By default, it will attempt to connect a remote controller on the loopback interface, `127.0.0.1`, on the standard OpenFlow port 6633. If no connection is made, it continues attempting a connection at set intervals. Although the client network is also connected to an Open vSwitch that will connect to our controller, the only functionality provided to this switch is basic layer 2 learning, which all switches implement.

4.2.2.2 Mininet Host Configuration

Once client processes have been created and the physical infrastructure in place, we must configure the hosts on the network. At this point, we simply have unconfigured machines. To automatically configure the hosts after starting the network, we leverage Mininet's ability to script host configuration. Using Linux command line tools, host configuration is simple a matter of executing commands in a particular order. Figure 8 shows our configuration script for Phase 2. After running the previous `mn` command and the network is created, a Mininet prompt is provided. It is as this prompt we can provide command for hosts to execute. The format the Mininet interpreter expects is:

```
mininet> hostname command(s)
```

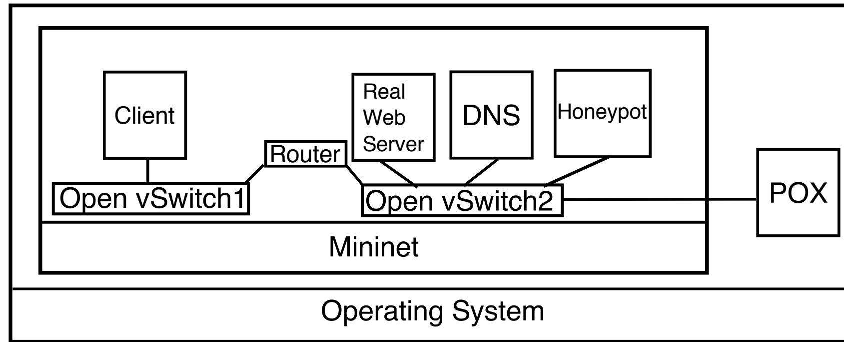


Figure 7: Simple Mininet design.

As Figure 8 shows, we provide the router, client, DNS server and Web servers with several commands. After executing these commands, each host has an IP address, including both the router's interfaces. Additionally, a honeypot and real web server are started, and last, the DNS server is started.

The network topology and configuration for Phase 2 will remain static, and once the Open vSwitches successfully connect to controller, we are ready to begin taking advantage of OpenFlow's features.

4.2.3 Initial Switch Configuration

In Figure 8's configuration we did not provide any information to the switches. Accordingly, the switches initially have no logic, and will drop any packet received. When a switch connects to the controller, the first rule added directs the switch to send all traffic to the controller. In POX, we can add this rule to switch using code in Figure 9. Sending every packet to the controller has performance overheads. Ideally, we want the switch to be a layer 2 learning switch. The notion of SDN is that one separates the control plane and data plane, and as a result, we write the layer 2 learning logic on the controller. This is contrary to modern networks where firmware sits on the switch to make decisions.

Creating a layer 2 learning switch is relatively straight forward. To act as a learning switch, we keep a mapping of MAC address to port. Anytime a packet originates from a host, the packet is elevated to the controller. The controller checks to see if the source MAC address has been seen before. If not, the controller has *learned* the port location of the originating client and creates a new rule for traffic *destined* to that MAC address to go out the corresponding port. Next, the controller looks at the destination MAC address. If the MAC address has originated traffic before, it has an entry in the dictionary and the controller can send the packet out that particular port. On the other hand, if the destination MAC's port is not known, the controller floods the packet out all ports. Eventually, all hosts will have originated traffic which means their MAC address will have an entry stored on the switch and traffic will no longer be elevated to the controller.

So far we have defined a network topology and then created and configured the hosts. The final component that we need for a functional network is to provide the switches with basic layer 2 learning.

4.2.4 External Choreographer

With a functioning network using a learning switch, we move into discussion of our implementation of Choreographer for external clients. This section will provide the basic information on Choreographer's implementation. Along with Choreographer's implementation, it is necessary to discuss some basic OpenFlow and DNS information. This includes discussion of our DNS zone file and how packets are elevated to

```
# Configure the router's two interfaces. One to client network, one to server network
router ifconfig router-eth0 192.168.12.1 netmask 255.255.255.0
router ifconfig router-eth1 192.168.23.1 netmask 255.255.255.0

# Enable packet forwarding on router
router sysctl net.ipv4.ip_forward=1

# Allow router to permanently cache DNS servers MAC
router arp -i router-eth1 -s 192.168.23.2 00:00:00:00:00:0e

# Configure client's IP and default  gateway. Set the client's DNS server
client ifconfig client1-eth0 192.168.12.2 netmask 255.255.255.0
client route add default gw 192.168.12.1
client echo "nameserver 192.168.23.2" > /etc/resolv.conf

# Configure the honeypot and real web server's IP address and default gateway
realweb ifconfig server2-eth0 192.168.23.3 netmask 255.255.255.0
realweb route add default gw 192.168.23.1
honeypot ifconfig server1-eth0 192.168.23.4 netmask 255.255.255.0
honeypot route add default gw 192.168.23.1

# Start a web server on honeypot and real web server
honeypot cd /var/www_honey && python ./PHPHTTPServer.py 81 &
realweb cd /var/www && python ./PHPHTTPServer.py 80 &

# Configure DNS server's IP and default  gateway
dns_server ifconfig dns_server-eth0 192.168.23.2 netmask 255.255.255.0
dns_server route add default gw 192.168.23.1

# Start the DNS server
dns_server /etc/init.d/bind9 stop
dns_server /etc/init.d/bind9 start &
```

Figure 8: Configuring the network.

the controller. While covering the basics, we also discuss Choreographer's external implementation, and point out additions to the implementation that address internal clients requesting services.

4.2.4.1 Basic DNS Configuration The previous Choreographer's implementation relied on randomly generating IP addresses by changing BIND's zone file and reloading the file into memory. This approach was not only inefficient but also implementation-specific. That is to say, if you were to change the DNS server to be a Microsoft implementation, new additions would be required in Choreographer. In a SDN architecture, we move the randomization and updating of IP addresses to the controller. Since DNS IP address randomization occurs at the controller, our approach is DNS implementation agnostic. Accordingly, we allow the DNS server to have a static configuration as in Figure 10. In this zone file, we have an entry for the Web server (`www`) at IP address `192.168.23.3`. Besides the name server (NS) entry, the only other DNS entry is a wildcard. Essentially, this means that any domain besides `www.facebook.com` such as `web.facebook.com` or `ssh.facebook.com` will return IP address `192.168.23.253`.

```
msg = of.ofp_flow_mod() # Create a flow message to send to the switch
msg.match=of.ofp_match() # Create a match object
msg.match.dl_type=0x800 # Match Ethernet packets
# On a match perform the action that sends the packet the port
# that the controller is connected to
msg.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))
# Install flow mod on switch
event.connection.send(msg) # Send rule to the switch
```

Figure 9: Directing switch to send all packets to the controller.

```
$TTL     60
@   IN  SOA   facebook.com. root.facebook.com. (
     3                ; Serial
     604800      ; Refresh
     86400        ; Retry
     2419200    ; Expire
     604800 )    ; Negative Cache TTL
     ;
       NS     ns.facebook.com.
ns     IN     A     192.168.23.2
www    IN     A     192.168.23.3
*      IN     A     192.168.23.253
```

Figure 10: BIND zone file

4.2.4.2 DNS Packet Elevation

We handle DNS packet elevation by installing another rule on the protected network's switch . The new rule in Figure 11 directs the switch to send all packets originating from source port 53, (i.e., DNS responses), to the controller.

```
msg = of.ofp_flow_mod() # Create a flow message to send to the switch
msg.match = of.ofp_match() # Create a match object
msg.match.dl_type = pkt.ethernet.IP_TYPE # Match IP packets
msg.match.nw_proto = pkt.ipv4.UDP_PROTOCOL # Match UDP packets
msg.match.tp_src = 53 # Match UDP source port 53 (outgoing DNS packets)
# On a match perform the action that sends the packet the port
# that the controller is connected to
msg.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))
# Install flow mod on switch
event.connection.send(msg)
```

Figure 11: Direct a switch to send all DNS packets to the controller.

The controller is not intercepting packets going to the DNS server, the requests only traverse the data plane of the switch before the server sees it. Once the server receives and processes the request, the response is elevated to the controller. When receiving any packet, the POX controller processes a "PacketIn" event. In this event, the controller application is passed the original packet that causes the elevation encapsulated in an OpenFlow message. POX then provides many builtin features for processing and extracting packet information from OpenFlow encapsulated packets. The most basic of these features is the `parsed` function that extracts the original packet into the corresponding type, such as TCP or UDP.

```
packet = event.parsed # This is the parsed packet data.
```

4.2.4.3 DNS IP Address Randomization Packet elevation can happen for multiple reasons. It may be DNS or elevated since we are performing layer 2 learning. The first action we take is to decide what to do with the packet. Basic forwarding decisions should happen last, modifications to the packet, such as the destination IP address, could change the forwarding decision. Here we are concerned about Choreographer modifications. Therefore, we check to see if the packet is UDP, and if so, we check to see if the packet is a DNS packet. Upon receiving a DNS packet, we loop through all responses. For large websites, it is possible to return multiple IP address. Returning multiple IP addresses allows for recursive name servers to round-robin the different addresses to provide load balancing to its clients. If Choreographer was not aware of the multiple responses returned, the client may use one of the responses not randomized and appear to be making illegitimate requests. However, our server never returns more than one entry per domain.

As each response is processed, we randomized the address by pulling a random IP addresses from a list. We hardcode a random IP address list into the configuration file on the controller. After choosing a random IP address, the controller updates a structure that contains requests from clients. The controller adds an entry into the dictionary with the IP as the key and the value as the current timestamp. The timestamp is stored to allow responses to timeout. Because we are serving external clients, we have a brief period of time where anyone can attempt to use the IP address returned through DNS. We allowed this period of time for requests to arrive to be 3 seconds. Based previous research [55], legitimate clients tend to use the IP address provided by DNS very shortly after receiving it, which makes a 3 second window a reasonable choice. After choosing a random IP address and updating the dictionary, the DNS packet is updated to reflect the random IP address. The controller then forwards the packet on to its destination. DNS interception and modification in Choreographer is shown in Figure 12.

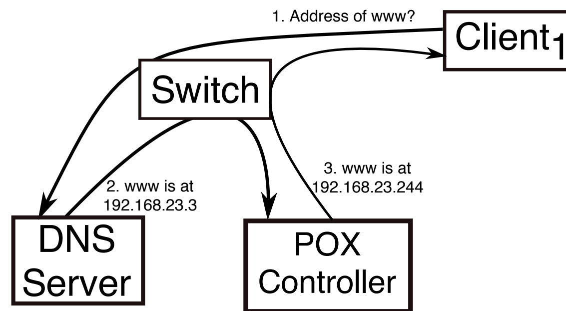


Figure 12: Choregrapher DNS interception and modification.

4.2.4.4 Handling Client Connections Receiving the response, the client can attempt a connection. While we support both TCP and UDP, our servers run Web server and use HTTP over TCP. Accordingly, the connection attempt to the random IP address will trigger a packet elevation to the controller as the packet do not match any flows, and we default to sending the packet to the controller. TCP packet elevation and flow approval in Choreographer is shown in Figure 13. Assuming the IP address was handed out within the last 3 seconds, the flow to the real Web server should be approved. Approval requires sending two different flow messages to the switch. One flow will match incoming packets and the other will match outgoing packets. Once the TCP SYN packet is received, the incoming and outgoing

flows are immediately created. The process for creating the incoming flow (Figure 15) has several steps. Some items are similar to past examples, but there are a few that we address. First, the incoming flow's match is looking for the client's IP address and source port. After the match is created, we add several actions to the flow. The first action performs a NAT translation from the random IP address to the actual internal IP address. The second action is a little more subtle. When a client attempts to a connect to a random IP address, the router has an interface that is on the same subnet as the random, destination IP address. Thus, the router believes it can communicate directly to the destination and will ARP for the destination's MAC address, but this IP address is not associated with any real host. If the router fails to receives a response to the ARP request, the client's connection will fail.

To solve this issue, we use an ARP proxy approach [61]. Using an ARP proxy, we add a new rule to the switch that forwards all ARP request traffic to the controller, but not ARP responses. When the controller receives the ARP request, it checks to see if the IP address in the ARP request is one of the randomly generated addresses. If so, the controller answers the request using a faux MAC address. Otherwise, the packet is forwarded normally. In our case, we always respond to the router's request with the MAC address `11:22:33:44:55:66`. With the proxy approach, just using NAT on packets coming from the client will not be enough because the destination will see the packet was intended for different MAC address (`11:22:33:44:55:66`). Accordingly, we must also change the MAC address of the packet sent from the router to correctly reflect the Web server's MAC. Likewise, the reverse direction must revert the IP address and MAC address back to the original values, or the router will not know how to handle the packets. Figure 14 shows this process. ARP proxying is not required in Choreographer's original implementation because NAT was happening at a router, but with SDNs, we are not manipulating packets until they arrive at the switch. Since packets are not manipulated until they arrive at the switch, the router must know the MAC address associated with the random IP address.

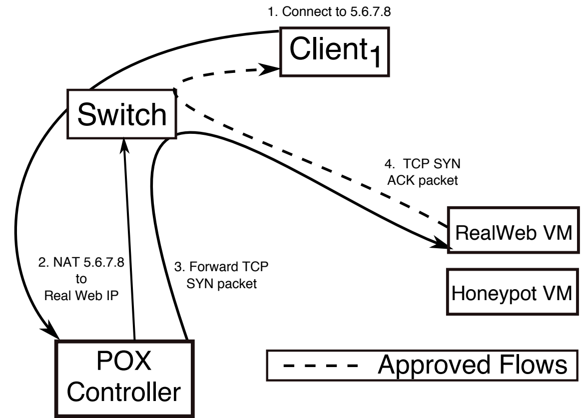


Figure 13: Choregrapher TCP flow interception and flow approval.

Each entry in the flow table has two different timeouts associated with it. One timeout, called an idle timeout, expires a flow after a period of inactivity. The other timeout is a hard timeout and expires a flow after a specific amount of time has elapsed, regardless of how active the flow is. In our implementation, we chose timeouts of 60 and 300 seconds, respectively. Neither timeout is required, and a flow can exist indefinitely.

When we configured the network for Phase 2 in Figure 8, we started the honeypot's Web server on port 81, rather than port 80. Because the real Web server and the honeypot are both on the same physical machine, we cannot bind two different processes (real server and honeypot server) to port 80. The remedy

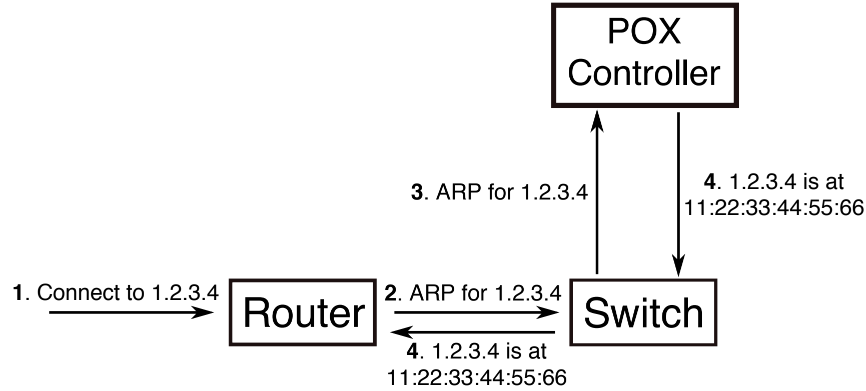


Figure 14: ARP Proxy for Choreographer in Mininet

to this solution is to use Port Address Translation (PAT), similar to NAT, which takes traffic destined to a particular port and translates it to another port. We bind the honeypot to port 81 and when adding flows to honeypot on the switch, Figure 15 has an additional action that performs PAT.

```
incoming.actions.append(of.ofp_action_tp_port.set_dst(81))
```

Symmetrically, an outgoing flow which is also added to the switch converts the port back and also has reverts the ARP proxy.

We have now covered in detail all of the aspects of Choreographer's implementation, which works for external clients requesting services. We started discussion looking at how we configured a simple BIND DNS server to provide DNS functionality but then showed how we could direct the switch connected to the server to elevate all DNS responses to the controller, which then manipulated the response with a random IP address, which is how Choreographer provides capabilities. After this, we showed the process of what happens with a client attempts to connect to that IP address over TCP. This includes adding two flows, incoming and outgoing, to the switch as well as using an ARP proxy, NAT, and PAT.

4.2.5 Internal Choreographer

Interestingly, the SDN paradigm allows us to easily implement the protections internally. In fact, the internal protection already exists because we are enforcing the capability at the switch. Without explicit permission by the controller, traffic is by default elevated to the controller and subsequently sent to the honeypot. Even though the protection is inherent in the current design, we add a few more features.

We now assume that clients requesting services are internal. This does not change anything in terms of our network topology and configuration. The only change is the logic on the controller. Before, it was assumed the client making the request was external. Due to this assumption, the controller could not make a more restricted mapping for the IP address issuing the request, but if an internal client makes a request directly to the internal, authoritative DNS server, the controller knows the IP address of the client that issued the request. Now, when a DNS request is elevated to the controller, it checks the source IP address to see if it lies within the organization's subnet. If the source IP is internal, the controller adds a new entry in a dictionary explicitly used for internal clients. The dictionary contains the client's IP, the random IP, and time of request. This more thorough bookkeeping allows the controller to deny any clients, internal or external, from attempting to guess an IP address given to an internal client. For example, an external

```
incoming = of.ofp_flow_mod() # Create a flow message to send to the switch
incoming.match=of.ofp_match() # Create a match object
incoming.match.dl_type=0x800 # Match Ethernet packets
incoming.match.nw_proto=pkt.ipv4.TCP_PROTOCOL # For TCP protocol only
incoming.match.nw_src = IPAddr(sourceIP) # Match the client's IP address
incoming.match.nw_dst = IPAddr(randomIP) # Match the random IP address from DNS
incoming.match.tp_dst=80 # Web server is on port 80
incoming.match.tp_src=source_port # Use the client's random source port
incoming.idle_timeout=inactivity_timeout # If idle for this many seconds, delete the flow
incoming.hard_timeout=hard_timeout # No matter what delete the flow after this many seconds

# Change the destination IP address to the Web server's real IP
incoming.actions.append(of.ofp_action_nw_addr.set_dst(IPAddr(web_server_ip)))

# Change the destination MAC address to the Web server's real MAC
incoming.actions.append(of.ofp_action_dl_addr.set_dst(EthAddr(web_server_mac)))

# Act like normal layer 2 learning switch
incoming.actions.append(of.ofp_action_output(port = of.OFPP_NORMAL))

event.connection.send(incoming) # Send incoming flow mod message the the switch
```

Figure 15: Create a flow for incoming packets from the client to the Web server.

adversary may by be attempting to connect to all of the IP addresses in a subnet in hopes that someone will have made a valid DNS request at that time, but because our dictionary knows which specific internal client made the request, other clients are denied access.

This approach also has other interesting possibilities we did not explore. For example, it is possible to provide internal clients with certain ranges of IP address. Perhaps if particular clients are more trusted, the IP space being randomized could be reduced. This may also be done if the amount of internal traffic is considerably less than the external traffic.

4.3 Performance

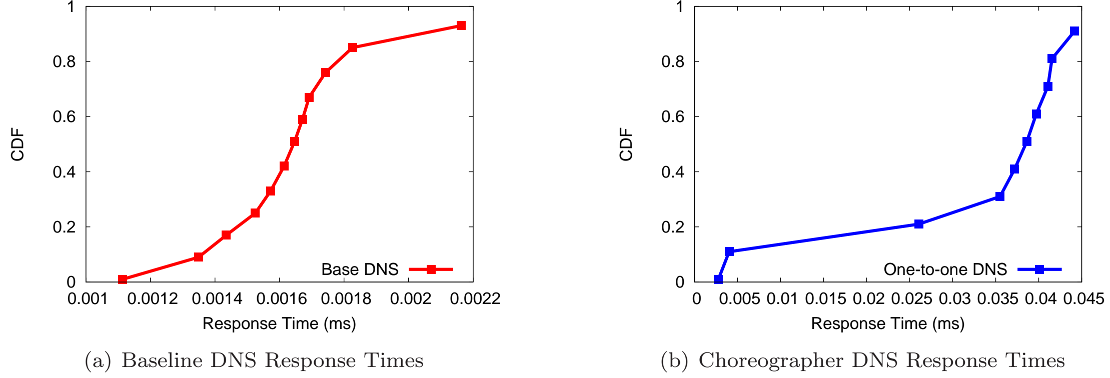
Packet elevation is a concept that does not exist in traditional networks. Accordingly, packet elevation to the controller and time the controller takes to perform functions and make decisions could introduce so much overhead the the approach becomes impracticable. Indeed, if every packet traversing a switch was elevated to the controller, it is likely to have noticeable performance degradation. A SDN tries to optimize by allowing the switch to make decisions locally when possible. Our approach to implementing Choreographer has a mixture of requiring information to be elevated, and when possible, allowing the switch to make decisions on its own. In this section, we explore the added overheads of our approach. We consider the DNS overhead, connection setup over head, and additional transmission time.

4.3.1 DNS Overheads

Previously, DNS manipulations occurred by manipulating BIND's zone file and reloading it into memory, but in the SDN implementation, we use packet elevation to control response randomization at the controller. We test the overhead of elevation and randomization to the controller by first quantifying the time to receive a response with no elevation or randomization, referred to as the baseline, and then with Choreographer

running. Our system configuration remains that of Figure 12. In these experiments, we do not attempt to connect to the address returned by Choreographer.

The baseline response times and response times with Choreographer running are shown in Figure 4.3.1 and Table 8.



(a) Baseline DNS Response Times

(b) Choreographer DNS Response Times

Figure 16: DNS response times of the baseline in Figure (a) and Choreographer in Figure (b)

	Mean (ms)	Median (ms)	Standard Deviation (ms)
Baseline	0.0032	0.0016	0.0157
Choreographer	0.0337	0.0386	0.0184

Table 8: DNS response statistics

The overhead introduced by Choreographer's DNS manipulation is minimal and unlikely to be noticed by clients. This is especially true for browsers that implement prefetching, as the request is made before the client actually attempts to connect.

4.3.2 Connection Setup Overheads

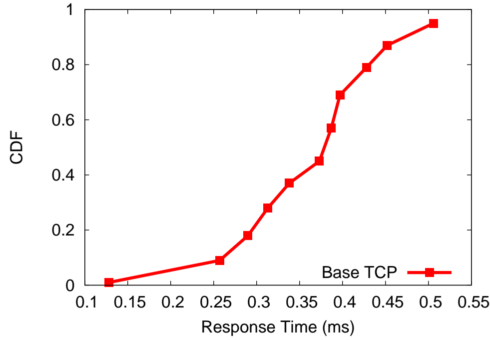
Like DNS responses, TCP (and UDP) connections have the initial packet elevated and is used to establish flows between the client and server. Figure 4.3.2 and Table 9 show the results of experimentation. A subtlety of these results is that they also include ARP proxying. The controller must intercept the ARP request from the router to random IP address from the DNS response and response with a faux MAC address. In these experiments, we must first issue a DNS request and then attempt to connect to the address returned by Choreographer. DNS overheads are not include in the connection setup overheads.

	Mean (ms)	Median (ms)	Standard Deviation (ms)
Baseline	0.4112	0.3825	0.4597
Choreographer	72.3486	74.6295	10.9594

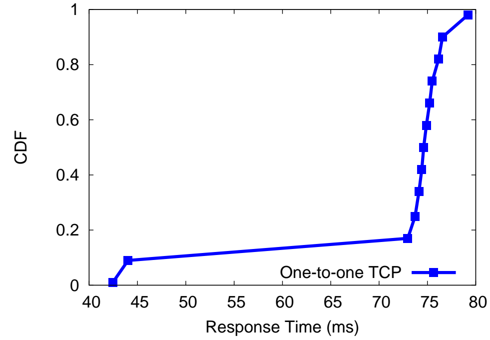
Table 9: TCP response statistics

4.3.3 Transmission Overheads

Once the DNS response is received and the TCP connection is established, we found that transmission speeds are not noticeably different with Choreographer is running versus not. Accordingly, the only delays a client may notice are in the initial connection setup, which includes DNS.



(a) Baseline TCP Connection Times



(b) Choreographer TCP Connection Times

Figure 17: TCP connection times of the baseline in Figure (a) and Choreographer in Figure (b)

4.4 Discussion

During development of this phase, there were many considerations that were made in some cases to simplify design. We address these considerations and their impact now.

4.4.1 Redundant TCP Connections

Our design assumes the client makes not only one DNS request per connection but also one TCP request. This may not be an accurate assumption. For example, the browser may issue multiple TCP requests in order to predict network performance issues [2]. Depending on network performance and which request the controller receives first, the result may be unexpected. For example, after the first request is received and successfully gets a flow to the real server, the response (SYN-ACK) is sent back to the originator, but the SYN-ACK could be dropped. Then the next connection could be sent to the honeypot since the first was answered.

4.4.2 Redundant DNS Requests

As with multiple TCP requests, using redundant DNS requests as a performance increase is also possible [65]. This is advantageous because DNS is over UDP, which is an unreliable protocol. For us, this would cause high congestion at the controller. It also causes the controller to had out IP addresses more frequently, potentially exhausting all IP addresses and resulting in a "fail-open" state with all IP addresses considered active. Part of these issues can be solved with more intelligent logic on the controller.

4.4.3 Matching Client Source Port

When a client connects to our network, we create incoming and outgoing flows. The incoming flow as shown in Figure 15 creates a match that uses the client's source port. The effect of this is two fold. It forces the client to need a new flow for each connection, but it also means that no two clients behind NAT can share a connection to a server. That is, if two external clients are behind NAT and one of them issues a DNS request, they can they cannot both access the resource. If we were to remove the source port, both clients could access the resource although only one of them requested it, and if one client is malicious, the outcome results in redirection to the honeypot for both of them.

4.4.4 Flow Timeouts

We did not deeply consider flow timeouts during development. OpenFlow uses hard timeouts and soft timeouts. Hard timeouts are set to remove flows after a certain period of time, and soft timeouts are used to expire inactive flows. In a real deployment, timeouts need careful consideration. If a user were downloading a large file using a Choreographer flow and the hard timeout were triggered, the user's flow would be redirected to the honeypot, which may result in unexpected behavior. Similarly, too short of a soft timeout may unnecessary remove a client's flow. We chose to set timeouts high enough that we did not consider these behaviors. However, if timeouts are too high, the switches flow entry table can become full, resulting in new connections not being successful.

4.4.5 IPv6 Support

IPv6 adoption is increasing [13], and it has new security concerns [18]. While we briefly explored this option for internal clients, POX bugs prevented further consideration. These bugs bring forward concerns about IPv6 applications have not been fully tested and are a potential security vulnerability for organizations that are beginning adoption.

4.4.6 Controller DNS Functionality

Our approach allows an organization's DNS infrastructure to go untouched and allows IP randomization to occur at the controller, but this is not a necessary requirement. In theory, the DNS server could itself be built into the controller. However, there are well designed DNS server applications that exist today. There are features such as DNSSEC [14] that are non-trivial implementations and already exist in open source software. It is also unclear what, if any, performance implications this approach would have.

4.4.7 Enabling Direct IP Communication

This Phase requires clients to communicate using DNS to obtain a capability to talk to a particular end host. Requiring DNS may not be desirable for certain applications. For example, there may be a logging application that requires IP address to be hardcoded in a configuration file. In situations where direct IP address access is desired, the controller can proactively insert these rules, and they can be restricted down to source/destination IP and port combinations for maximum protection.

4.5 Conclusion

With this approach, Choreographer now runs in a SDN setting and works for both internal and external clients. We have shown the ability to configure OpenFlow capable switches in a virtualized environment, which includes virtual hosts and a virtual switch. If clients do not use DNS to initiate their transactions, then communication is blocked.

Since hosts are only known by their DNS hostname, as opposed to IP, scanning attacks that attempt to scan IP ranges and spread malware are mitigated. Only being able to contact hosts by their domains also means that network reconnaissance using system tools for IP addresses is useless.

After implementing our work, we discussed considerations for active deployment. We then presented performance results from the external client's perspective.

5 Phase 3: Creating a One-to-One Model

With Choreographer running on a SDN, we have a mechanism for providing DNS responses with random IP addresses. Assuming the addresses are not exhausted, each client will receive a unique IP address. With each client connecting to a different IP address, we move towards a one-to-one model.

5.1 Introduction

In this final phase of work, we create a one-to-one client-server infrastructure. For each client connecting to our network, we instantiate a new front-end virtual machine. A one-to-one model adds security by isolating clients in their own virtual machine. If a client is malicious and compromises the server VM, the attack is localized to that VM, and after the connection is terminated,the VM image can be removed, preventing pervasive changes.

Traditionally, clients are multiplexed at servers. Choreographer leverages DNS to allow us to grant capabilities to clients in order to access services. Our implementation only uses a single service, `HTTP`, but in practice, it is likely organizations will want to provide more than one type of service. When this happens, clients using different services may be multiplexed on the same physical machine. An example of this is the popular LAMP architecture [8], which includes Linux, Apache, MySQL, and PHP. The LAMP architecture runs a Web server and an SQL server, which may be accessed by multiple users with different needs. In the case the Apache Web server is compromised, the other resources being shared on this machine, .e.g. MySQL, may is also compromised and vice versa. If these processes were running in separate VMs or Linux containers, the attack would be isolated and unable to affect other processes.

Today, multiple services on the same machine may use the same domain name. In our LAMP example, the Web server and SQL server could both be accessed using `example.com` since services on the same machine will have the same IP address. De-multiplexing which service a client is attempting to access will happen once the connection is attempted to a certain port. Potentially, a DNS request could be received for a client that wished to access the Web server, but the address returned could be used to access the MySQL server instead. De-multiplexing on port numbers is an issue when VMs are dynamically spawned on a per-client basis. Our approach must know ahead of time exactly which service the client is requesting in order to instantiate the proper VM. For this, we leverage domain names as indicators of the service. Now, instead of having one domain such as `example.com` mapping to multiple services, we require domain names such as `www.example.com` or `mysql.example.com`. Now `www` and `mysql` indicate the requested service. Using the host name as a cue, we create the corresponding front-end server type.

With Choreographer and a one-to-one client-server model, we are providing each user with a random IP address and directing them to a particular server instance. The motivation for IP address randomization is that DNS acts as an access control mechanism whereby if clients do not issue DNS requests, the IP address of the destination is not known. Direct IP-based connections is occasionally useful. In particular, a common IP address means Choreographer is no longer restricted by the number of IPv4 address it has to randomize between. If not using a common IP address, Choreographer has to continually randomize IP address, and if as many requests are made as Choreographer has to rotate between, then all IP addresses will appear active. Accordingly, if a common IP address is used internally for all services, IP address exhaustion can be external clients since multiple addresses are not needed internally.

5.2 Implementation

We now discuss our implementation of a one-to-one network model using Mininet and the POX OpenFlow controller. Our previous work is extended by allowing each client to connect to their own server process for isolation. For internal clients, a minor modification is made to allow a single IP address to given for all services to alleviate IP exhaustion.

5.2.1 Mininet Alterations

This phase differs from the previous phase since we are connecting each client to their own server rather than a shared front-end server. Still, we use Mininet with modifications to the previous topology. Rather than use a network topology with 1 client and 2 servers, we modify the topology to include 10 clients and 10 servers as shown in Figure 18. Accordingly, the network configuration is also updated to include the IP and gateway configurations in Figure 20.

```
# Add switches
SWITCH1 = self.addSwitch('switch1')
SWITCH2 = self.addSwitch('switch2')

# Add hosts
ROUTER = self.addHost('router')
CLIENT = self.addHost('client1')
DNS_SERVER = self.addHost('dns_server')
HONEYPOT = self.addHost('honeypot')
SERVER1 = self.addHost('server1')
...
SERVER10 = self.addHost('server10')

# Add links
self.addLink(CLIENT, SWITCH1)
self.addLink(ROUTER, SWITCH1)
self.addLink(ROUTER, SWITCH2)
self.addLink(DNS_SERVER, SWITCH2)
self.addLink(HONEYPOT, SWITCH2)
self.addLink(SERVER1, SWITCH2)
...
self.addLink(SERVER10, SWITCH2)
```

Figure 18: Phase 3 network topology

5.2.1.1 Back-end Server Other than the “backend” server, the topology and configuration is similar to phase 2, with the exception of the additional real Web servers (server1-server10). The back-end server acts as a file server that front-end servers use to remotely loads files. A visual representation of the network topology including clients, front-end servers, and the back-end server in Figure 19 show that each front-end server has a flow to the back-end server. This flow is only provided once a client establishes a connection to a front-end server. Importantly, no communication to the back-end server is allowed other than from authorized front-end servers.

5.2.1.2 Dynamically Spawning Servers Ideally, front-end servers would be spawned dynamically upon request, but due to the limitations of Mininet, we must have servers hot-spawned but not yet

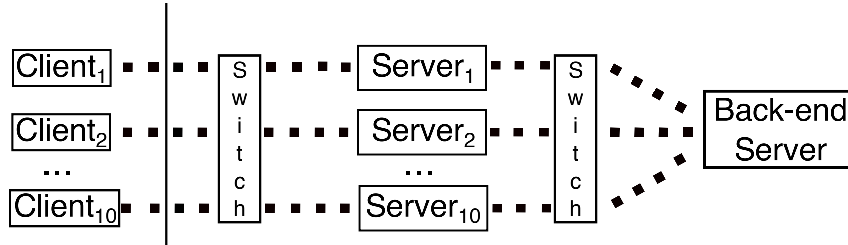


Figure 19: Network topology with 10 clients, 10 front-end servers and a back-end server.

configured. Hot-spawned servers are already booted and ready to be configured for connections. This notion is similar to ghost VMs [54]. Having servers pre-allocated has no effect on the networking aspect of the one-to-one model and is strictly a performance concern. Accordingly, server instances are created when Mininet starts and are given their own separate root folder e.g., `/var/www`, which contains necessary server files.

5.2.2 Controller Modifications

We first duplicate the functionality from phase 2. This includes the layer 2 learning switch and both the internal and external implementation of Choreographer. We do not need to modify these implementations. For example, the default rules on the organizational switch still need to elevate DNS responses and packets that do not match any flow to the controller. Accordingly, we use all of our previous work and continue making additions to the logic to support the one-to-one model.

5.2.2.1 Service Types For our one-to-one model, we introduce the notion of a “service type”. As discussed in Phase 2, we use domain names as passwords, where if clients do not know the domain name of a server, they cannot access it. We further this idea by associating service types subdomains. For example, a DNS request for `www.example.com` has a subdomain of `www`. Accordingly, we associate this subdomain with a service type “`http`”. Similarly, `ssh` and `mysql` subdomains also have different service types.

The service type attribute enables two different features. It allows us to check that a virtual machine of a particular type exists when a DNS response arrives at the controller. Assuming the types exists, the response is sent and the controller maintains the service type and IP information using a record as follows:

```
external_request_record{'random_ip'}{'service_type'} = time()
```

When a connection request arrives, it is no longer sufficient to only check the IP address and the time associated with the request. We now check the destination port number in the request and derive the service type. For example, service type `http` has the port number 80. If a request came in on the corresponding IP address to a different port, it would be redirected to the honeypot. In the example above where the controller stores the state of the request, we assume an external client made the request, but if the request had been made internally, we would also capture the source IP address.

5.2.2.2 One-to-One Flows We now have a larger network topology in place and have developed a systematic process of determining the proper server type to use to connect a client. The next modification allows us to create client to front-end and front-end to back-end flows. When a client issues a DNS request for a service and receives a response, the controller does not yet begin any configuration. After all, the

```
# Configure the back-end server that front-end servers will use
backend ifconfig backend-eth0 192.168.23.16 netmask 255.255.255.0
backend route add default gw 192.168.23.1

# Configure each client's IP and default gateway.
client1 ifconfig client1-eth0 192.168.12.2 netmask 255.255.255.0
...
client10 ifconfig client10-eth0 192.168.12.14 netmask 255.255.255.0
client1 route add default gw 192.168.12.1
...
client10 route add default gw 192.168.12.1

#Set each client's DNS server
client1 echo "nameserver 192.168.23.2" > /etc/resolv.conf
...
client10 echo "nameserver 192.168.23.2" > /etc/resolv.conf

# Configure the honeypot and each Web server's IP address and default gateway
honeypot ifconfig server1-eth0 192.168.23.3 netmask 255.255.255.0
...
server10 ifconfig server10-eth0 192.168.23.15 netmask 255.255.255.0
honeypot route add default gw 192.168.23.1
...
server10 route add default gw 192.168.23.1

# Start honeypot and all server VM's
honeypot cd /var/www && python -m SimpleHTTPServer 80 &
...
server10 cd /var/www10 && python -m SimpleHTTPServer 92 &
backend cd /var/www_backend && python -m SimpleHTTPServer 93 &
```

Figure 20: Configuring the network for a one-to-one model.

client could issue a DNS request and never attempt a connection. Rather, the controller tracks the state of client requests and waits until a connection is attempted at which point the flows are established.

For simplicity, discussion is focused on a connection to an HTTP server which will use TCP as the transmission protocol. We make slight changes to Figure 12 in Figure 21 to show when an external client issues a DNS request, the controller maintains a record of the IP address and time it was provided to the client.

Now the controller knows that client has been given the IP address `5.6.7.8` for service type **`http`**. When a client issues a HTTP request to that IP address, a TCP connection request (TCP SYN) packet is first transmitted. The default rule on the controller is still to elevate packets that do not match any flow. Accordingly, the TCP SYN packet is elevated to the controller. The PacketIn event is triggered and the controller now uses the packet's information to determine whether the request is legitimate and if so, configure a server of the matching service type. Pseudocode for this process is shown in Figure 23 and a visual representation in Figure 22. Essentially, the TCP SYN is elevated to the controller where the data structure containing information on valid requests is checked. If the IP address the client is attempting a connection to has been provided in a DNS response within the last 3 seconds, then two flows are established. One flow allows the client to communicate directly to the front-end server, and the other

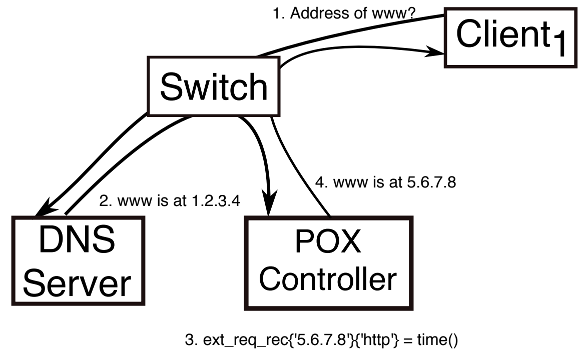


Figure 21: DNS interception and modification with controller state.

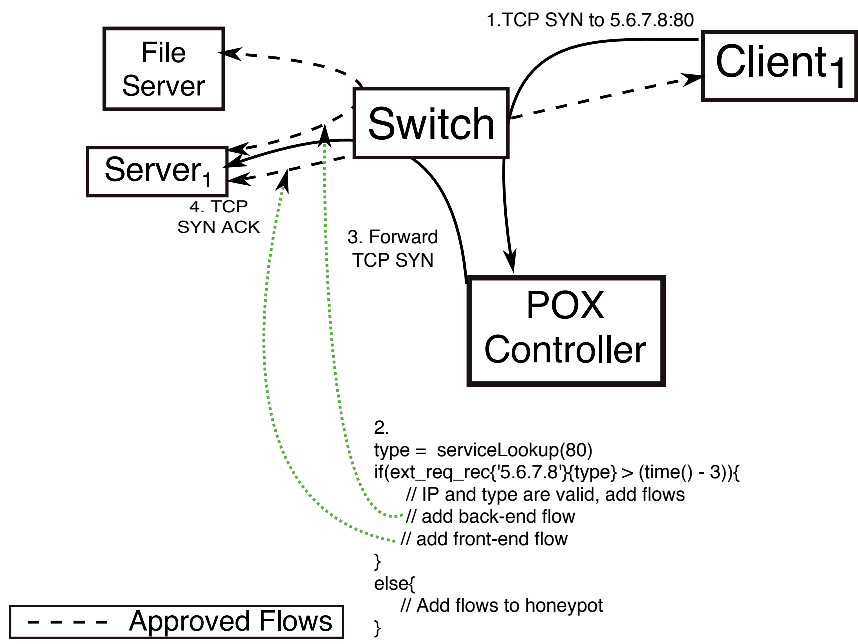


Figure 22: TCP interception and flow installation by controller.

rules allows the front-end server to communicate to the back-end server. If the connection attempt is not legitimate (i.e., `IPActive()`) fails, we perform similar actions but create flows to the honeypot and do not add back-end flows.

Clients are now able to issue DNS requests, receive a modified response, and have the controller maintain the state of the request along with the service type associated with the DNS request. When the client attempts to connect to the IP address in the response, the connection is elevated to the controller. Comparing the request to the controller's internal state, a legitimate request causes a back-end flow and a front-end flow to be instantiated. Once these flows are created, the controller forwards on the original TCP request to the corresponding front-end server. The connection then precedes uninterrupted by the controller.

After the connection is established, the client can issue an HTTP request. The front-end server's Web page uses PHP which causes a remote image file located on the back-end file server to be remotely loaded for the client. We configure the back-end server to also act as a Web server that hosts only a single image

```
# If the packet is TCP and TCP SYN flag is set
if(( packet.isTCP and packet.isTCP_SYN ):
    # Find service type associated with port
    serviceType = lookupServiceType(packet.dstport)

    # Dictionary lookup for valid IP and service type
    if( IPActive(packet.dstip), serviceType) ):
       # Find a free server of corresponding service type
        server = findFreeServer(serviceType)

       # Enable the front-end server to communicate to back-end server
        enable_backend(server.ip)

        # Add a flow that does NAT, PAT (if necessary), and
        #MAC address translation to front-end server
        add_flow(packet.srcip, packet.dstip, packet.srcport,
                       packet.dstport, server.ip, server.port, server.mac)
```

Figure 23: Configuring front-end and back-end flows on attempted TCP connection

file. Each time the Web page on the front-end server is loaded, a new copy of the image is transferred and stored locally.

5.2.2.3 Common IP Addresses Minimal modifications are needed on the controller to allow for a common IP address for internal clients. When parsing through DNS responses, we are already check to see if the client is internal. Instead of randomly generating an IP address, we always update the IP address to a known common address. The IP address is the same across all services, but because the controller is tracking service types requested, we always know if an attempted connection should be allowed based on the destination port and client IP address. However, changing the DNS packet can be avoided if the address provided by the DNS server is suitable, but since our approach is implementation-agnostic, we do not assume anything about the addresses returned by the DNS server. Packet elevation occurs because the controller must keep track of the state of requests.

5.3 Performance

In this phase, when continue using Choreographer to randomize IP addresses and allow the controller to additionally maintain a service type with each request. When a client attempts a connection, we elevate packets to the controller and configure a new server with front-end and back-end flow. Therefore, the only additional overhead in this phase is choosing a free VM and installing a back-end flow. The performance of spawning servers dynamically is not the focus of our work. Accordingly, we assume the virtualized servers are already created in our Mininet network and attempt to quantify the network performance as the number of clients increases.

5.3.1 Scalability

We now want to look at the overheads of our approach as the number of users grow since the overhead of a single client does not clearly predict the behavior of the model. For determining how the approach performs under load, we simulate first simulate 1 client and then 10 clients simultaneously attempting to

connect to a front-end server by first issuing a DNS request and then a TCP connection. A single client provides a direct comparison to the baseline and then we show the scalability.

	Mean (ms)	Median (ms)	Standard Deviation (ms)
Single Client	0.0337	0.0386	0.0184
10 Clients	0.1222	0.1206	0.0069

Table 10: DNS response statistics with 10 simultaneous requests

	Mean (ms)	Median (ms)	Standard Deviation (ms)
Single client	72.3486	74.6295	10.9594
10 Clients	132.7502	123.4295	21.2914

Table 11: TCP response statistics with 10 simultaneous requests

Table 10 and Table 11 show that the DNS response and connection setup time of 10 clients issuing DNS followed by TCP connections simultaneously. The response time for both DNS and TCP increases but is not overly concerning. Our POX controller is a single threaded application and has worse performance than many OpenFlow controller applications [22].

5.4 Discussion

A one-to-one client server model has several advantages and new areas for exploration. We now discuss potential performance and security concerns for our model and implementation.

5.4.1 Performance Considerations

We now consider different methods of spawning servers and denial of service considerations with such methods.

5.4.1.1 Server Spawning Faster connection setup times can be achieved using hot-spawned servers. These servers are created prior to a client requesting them and are given partial configuration ahead of time. Once a client requests a server, the controller only has to finish the configuration and does not need to wait for applications to start. We chose this option due to Mininet limitations, but this approach is not necessary. In particular, a low traffic network handling a small number of clients may not want to have servers running continually that are only utilized occasionally. If hot spawning is not used, the controller has to dynamically instantiate the server and configure it.

Regardless of when servers are spawned, there are two points during execution that the configuration may take place. On one hand, the controller does not take action after seeing a DNS request. Instead, the controller waits until a connection is attempted. This method aligns with our approach. The controller could also choose to react to client DNS requests instead of waiting for a connection attempt. Choosing to react to the DNS request allows the controller to begin the initialization before the client can attempt a connection. If the initialization happens in parallel with the DNS response going to the client, the amount of time to initialize is likely to be small. To increase the time for initialization, a DNS response could be held until the server instantiation is finished, but holding the DNS response may cause client applications to issue multiple requests.

5.4.1.2 Denial-of-Service A concern with spawning servers based on DNS requests may be problematic if clients issue requests but never attempt connections. Namely, unused servers would be running and needlessly using system resources. An adversary with enough resources could attempt a Denial of Service (DoS) attack causing the target network to become unresponsive. To mitigate these attacks, deploying our approach in the cloud for elasticity [20] is possible. These servers should not run forever. With Choreographer, if a client does not issue a request within the time range of the DNS response's TTL, the IP address is no longer valid. As a result, the server is no longer needed and those system resources can be reclaimed.

5.4.2 Security Considerations

As our work looks at improving security, we also consider security implications our work introduces. If the implication is negative, we discuss options to resolve the concern.

5.4.2.1 Service Types to Detect Malicious Activity Our one-to-one model uses service types to determine what resource type a client is requesting. Since some network configurations have multiple services running on the same machine, client requests to that machine are demultiplexed based on port numbers. Then it is possible for an attacker to perform DNS lookups, and upon receiving a response, attempt to connect to other services on different popular ports. In our system, this would quickly lead to detection and allow us to easily thwart the attacker. Detection is easy because DNS requests are associated with a service type, if a connection is attempted to a port other than what was requested, an alarm can be triggered.

5.4.2.2 Client-side NAT One assumption in our approach is that we are associating an IP address with a single client. In practice, this is not always the case. Residential clients are typically only given a single IPv4 address from their ISP. With WiFi-capable smartphones, tablets, computers, and gaming consoles, users may have more than one device connected to their ISP's modem using a wireless router that implements NAT. Therefore, we may see multiple clients coming from the same IP address. Approaches exist to differentiating clients behind NAT. One approach requires clients to use WiFi and needs physical access to the network [21]. Another approach could be using client identifiers such as browser User Agent strings to differentiate different clients [66]. These approaches are impractical and privacy concerning; therefore, we do no assume an organization has or will attempt to differentiate clients using these solutions. The problem of clients using NAT then reduces to our one-to-one model having to multiplex clients behind the same NAT device. This is still a significant reduction in clients over modern multiplexing.

5.4.2.3 Packet Spoofing DNS uses the connectless transport protocol UDP, which is susceptible to spoofing. For our one-to-one model, we use DNS not only as a capability system but also to as an indicator of the type of service the client is accessing. The indicator then dictates what server we spawn. DNS spoofing is a well-known attack vector for DoS using both DNS amplification and reflection attacks. Our main concern with DNS spoofing is that it can lead to IP address exhaustion and thus a fail-open state. It may also lead to a DoS if too many requests are issued. This is particularly important since our TTL values are too short to allow DNS caching to have an effect and requests must be answered by the organization's authoritative DNS server. Spawning servers based on DNS requests is also problematic if a spoofed request is received.

Our implementation uses new client requests, TCP or UDP, to connect clients to already spawned servers. For TCP we use the TCP SYN packet and for UDP we inspect the packet tuple to determine

if the flow new or not. However, both of these can be spoofed. Considering TCP first, we only require the SYN packet to determine a flow if being utilized. We do not track where the client ever acknowledges the request. TCP cookies [38] is an existing solution that would prevent this spoofing attack but is not implemented in our solution. UDP does not have a trivial solution and is still vulnerable to these types of attacks.

5.4.2.4 Common IP Address Deployment Using a common IP address for all server should only be deployed for internal clients. If used externally, each DNS request would result in a open mapping that allows any client to connect temporarily. A common IP address would allow and adversary to only guess a single IP address and greatly raises the success rate. This is essentially equivalent to a fail-open state. Internally, we know the client issuing the the request and can restrict the capability to the client IP making the request.

5.5 Conclusion

In this final phase of work, we create a one-to-one client-server infrastructure. For each client connecting to our network, we dynamically configure network paths to a new front-end server as well as the paths to a back-end file server. We again leverage DNS as capability system and further utilize domain names to passively notify the controller what server should be used based on a service type, such as `http`. We also eliminate the requirement that front-end servers, each handling a different client, require unique IP addresses when being deployed internally. We then show that with up to at least 10 clients issuing requests to 10 different front-end servers, the approach scales reasonably well. Finally, we considered performance and security concerns of our model.

6 Future Work

In our work, we design and implement a one-to-one client-server model using SDN and virtualization. This work introduces these two concepts but only considers them from the the point of view of the Mininet architecture. Therefore, we consider other architecture designs and more in-depth analysis of our model.

6.1 Virtualization Adjustments

Mininet provides isolation using Linux containers and network namespaces within a single machine with the goal of rapid prototyping. Consequently, an actual deployment of our one-to-one model is not going use Mininet. The first adjustment for future development and real-world testing to part with the Mininet architecture. Separation with Mininet leaves us to replace a method of isolating client processes and an OpenFlow compatible switch.

6.1.1 A KVM One-to-One Model

KVM is the ideal candidate for replace client process isolation from Mininet. With KVM, we are able to spawn complete VMs running different service type and we can rely on a trusted hypervisor to enforce isolation. KVM can also be executed on a single physical machine that is able to support many clients or deployed in a cloud environment. With a more powerful virtualization technique, we are not restricted to simple process isolation that containers restrict our implementation to. Instead, we have complete VMs running a full OS that has more rich features. Another component needed is a switch that supports the OpenFlow protocol. Fortunately, Open vSwitch, which Mininet uses, is also compatible with KVM.

With KVM, we will no longer be restricted to Mininet's network topology and configuration techniques. KVM will permit us to dynamically instantiate hosts and configure them based on more traditional, host-based mechanisms, rather than using the Mininet command-line interpreter. Accordingly, our controller implementation will need to be changed in order to coordinate VM spawning and configuration, which is currently pre-configured.

6.2 OpenFow Controller Implementation

Which controller implementation we continue to use has a significant effect on the performance of our approach. Currently, we are using POX, which is a Python based controller implementation. For prototyping, this is convenient because each change we want to test does not require a code recompilation because Python is an interpreted language. However, it does not provide low level management or optimizations found in a compiled language. Further, POX is not a multithreaded controller. For future development and more complicated interactions a multithreaded OpenFlow controller such as Beacon [22] may be necessary. Accordingly, we will explore different controller implementations.

7 Conclusion

The goal of this work is to consider the security concerns associated with a traditional networking that has a many-to-one relationship between clients and servers. This relationship has been developed over time, originating from a time when computing sources were more scarce and multiplexing had to happen for scalability. Advances in computing, virtualization, and networking reduce the constraints on resources and mean that we are able to consider a novel one-to-one networking model. Our proposed model recognizes that sharing resources between many clients makes servers a prime target for attacks to compromise. Accordingly, we remove the need for multiplexing by providing each client with their own server.

To realize our model, we apply SDN and virtualization techniques in three phases. Our first phase introduces us to OpenFlow, a SDN controller called POX, and Pantao, an open source firmware that supports the OpenFlow 1.0 specification. This phase demonstrated the process. We were able to transition it into the next phase of work where everything was virtualized. Using the popular Mininet architecture, we implemented an DNS access control application called Choreographer in POX. Not only did we replicate its original functionality, but we added support for running internally by using the requester's IP address when creating flows. We present performance results on these changes and show that the overhead is minimal. Finally, our last phase is a culmination of all our previous work in addition to modifications to our controller and Mininet's topology. These modifications allow us to create 10 clients and 10 servers, which we use to create a one-to-one client-server model. We end the phase with performance results as well as discussion on security and performance concerns. In total, our controller logic resulted in just under 600 lines of Python code.

While our work provides the foundation for a deployable one-to-one model, we note limitations along the way. In particular, our approach is not as dynamic as we would like due to limitations on the Mininet architecture. These shortcomings lead to our future work where we will look at transitioning this work into a KVM environment.

References

[1] SANS Institute - SANS Information Security Reading Room - Data Leakage - Threats and Mitigation. http://www.sans.org/reading-room/whitepapers/awareness/

data-leakage-threats-mitigation-1931.

[2] Chromium opens useless TCP connections. http://csrc.nist.gov/publications/nistir/7316/NISTIR-7316.pdf, 2012.

[3] Linux containers. http://sourceforge.net/projects/lxc/, 2013.

[4] POX. http://www.noxrepo.org/pox/about-pox, November 2013.

[5] QEMU. http://wiki.qemu.org/main_page, November 2013.

[6] Virtualization technology. http://ark.intel.com/products/virtualizationtechnology, November 2013.

[7] The Xen project. http://www.xenproject.org, 2013.

[8] LAMP software bundle. http://linuxsolutions.org.in/lamp.html, 2014.

[9] Metasploit unleased - pivoting. http://www.offensive-security.com/metasploit-unleashed/Pivoting, 2014.

[10] OpenWrt wireless freedom. https://openwrt.org, 2014.

[11] Pantou : OpenFlow 1.0 for OpenWRT. http://archive.openflow.org/wk/index.php/Pantou_:_OpenFlow_1.0_for_OpenWRT, 2014.

[12] POX: Does pox supports SSL? http://lists.noxrepo.org/pipermail/pox-dev-noxrepo.org/2014-April/001595.html, 2014.

[13] Akamai. The state of the Internet. http://www.akamai.com/dl/akamai/akamai-soti-q313.pdf, 2013.

[14] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), 2005. Updated by RFCs 6014, 6840.

[15] Katerina Argyraki and David Cheriton. Network capabilities: The good, the bad and the ugly.

[16] Kevin Benton, L. Jean Camp, and Chris Small. Openflow vulnerability assessment. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, HotSDN '13, pages 151–152, New York, NY, USA, 2013. ACM.

[17] Matt Bishop and Carrie Gates. Defining the insider threat. In *Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead*, page 15. ACM, 2008.

[18] Carlos E Caicedo, James BD Joshi, and Summit R Tuladhar. IPv6 security challenges.

[19] S. Carl-Mitchell and J.S. Quarterman. Using ARP to implement transparent subnet gateways. RFC 1027, 1987.

[20] Sean Carlin and Kevin Curran. Cloud Computing Security. *International Journal of Ambient Computing and Intelligence (IJACI)*, 3(1):14–19, 2011.

[21] Y. Chen, Z. Liu, B. Liu, X. Fu, and W. Zhao. Identifying mobiles hiding behind wireless routers. In *IEEE INFOCOM*, pages 2651–2659, 2011.

[22] David Erickson. The beacon openflow controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[23] Open Networking Foundation. Openflow 1.0 switch specification. archive.openflow.org/documents/openflow-spec-v1.0.0.pdf, 2014.

[24] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, 2003.

[25] Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: security challenges in virtual machine based computing environments. In *10th Workshop on Hot Topics in Operating Systems*, 2005.

[26] Tal Garfinkel and Andrew Warfield. What virtualization can do for security. *The USENIX Magazine*, 32(6):28–34, 2007.

[27] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[28] Red Hat. KVM kernel based virtual machine. pages 1–11, 2009.

[29] HP. Hp 3800-48g-poe+-4sfp+ switch. `http://h30094.www3.hp.com/product/sku/10383928`, 2014.

[30] IBM. OpenFlow: The next generation in networking interoperability. Technical report, 2011.

[31] IBM. Ibm system networking rackswitch g8264. `http://www.redbooks.ibm.com/abstracts/tips0815.html`, 2014.

[32] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: Transparent moving target defense using software defined networking. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 127–132, New York, NY, USA, 2012. ACM.

[33] Xuxian Jiang and Xinyuan Wang. "Out-of-the-box" monitoring of VM-based high-interaction honeypots. In *Recent Advances in Intrusion Detection*, pages 198–218. Springer, 2007.

[34] A.J. Kalafut, C.A. Shue, and M. Gupta. Malicious hubs: detecting abnormally malicious autonomous systems. In *IEEE INFOCOM mini-conference*, pages 1–5. IEEE, 2010.

[35] Keith Kirkpatrick. Software-defined networking. *Communications of the ACM*, 56(9):16–19, September 2013.

[36] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[37] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[38] Jonathan Lemon. Resisting SYN Flood DoS Attacks with a SYN Cache.

[39] Huan Liu and Sewook Wee. Web server farm in the cloud: Performance evaluation and dynamic architecture. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 369–380, Berlin, Heidelberg, 2009. Springer-Verlag.

[40] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *IEEE Conference on Cloud Computing (CLOUD)*, pages 423–430, 2012.

[41] Zhuoqing Morley Mao, Charles D. Cranor, Fred Douglis, Michael Rabinovich, Oliver Spatscheck, and Jia Wang. A precise and efficient evaluation of the proximity between web clients and their local dns servers. In *Proceedings of the USENIX Annual Technical Conference*, ATEC '02, pages 229–242, Berkeley, CA, USA, 2002. USENIX Association.

[42] William W. Martin. Honey pots and honey nets - security through deception. `http://www.sans.org/reading-room/whitepapers/attacking/honey-pots-honey-nets-security-deception-41`, 2001.

[43] Niels Provos Panayiotis Mavrommatis and Moheeb Abu Rajab Fabian Monrose. All your iframes point to us.

[44] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[45] Peter Mell and Karen Kent. *Guide to malware incident prevention and handling*. 2005.

[46] Robert Meushaw and Donald Simard. Nettop: Commercial technology in high assurance applications, 2000.

[47] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report.

[48] Big Switch Networks. Project floodlight. `http://www.projectfloodlight.org`, 2014.

[49] Juniper Networks. The QFabric architecture. `http://www.enpointe.com/images/pdf/The-Qfabric-Architecture.pdf`, 2013.

[50] NIST. Assessment of access control systems. `http://csrc.nist.gov/publications/nistir/7316/NISTIR-7316.pdf`, 2006.

[51] Bryan Parno, Jonathan M McCune, Dan Wendlandt, David G Andersen, and Adrian Perrig. Clamp: Practical prevention of large-scale data leaks. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 154–169. IEEE, 2009.

[52] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, pages 233–247. IEEE, 2008.

[53] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. Extending networking into the virtualization layer. In *In: 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII).New York City,NY(October 2009)*.

[54] Hangwei Qian, Elliot Miller, Wei Zhang, Michael Rabinovich, and Craig E. Wills. Agility in virtualized utility computing. In *Proceedings of the 2Nd International Workshop on Virtualization Technology in Distributed Computing*, VTDC '07, 2007.

[55] C.A. Shue and A. Kalafut. Resolvers revealed: Characterizing DNS resolvers and their clients. *ACM Transactions on Internet Technology (TOIT)*, in press.

[56] Craig A. Shue, Andrew J. Kalafut, Mark Allman, and Curtis R. Taylor. On building inexpensive network capabilities. *ACM SIGCOMM Computer Communication Review*, April 2012.

[57] Craig A. Shue, Andrew J. Kalafut, Mark Allman, and Curtis R. Taylor. On building inexpensive network capabilities. *SIGCOMM Comput. Commun. Rev.*, 42(2):72–79, March 2012.

[58] Lance Spitzner. Honeypots: Catching the insider threat. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 170–179. IEEE, 2003.

[59] Stuart Staniford, Vern Paxson, Nicholas Weaver, et al. How to own the internet in your spare time.

[60] Curtis Taylor and Craig Shue. Implementing a "moving target" system to protect servers. In *Proceedings of the Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '11, pages 81:1–81:1, New York, NY, USA, 2011. ACM.

[61] D Thaler, M Talwar, and C Patel. Neighbor discovery proxies (ND Proxy). *Request for Comments*, 4389, 2006.

[62] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[63] S.J. Vaughan-Nichols. Openflow: The next generation of the network? *Computer*, 44(8):13–15, 2011.

[64] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C Snoeren, Geoffrey M Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review*, 39(5):148–162, 2005.

[65] Ashish Vulimiri, Oliver Michel, P. Brighten Godfrey, and Scott Shenker. More is less: reducing latency via redundancy. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 13–18, New York, NY, USA, 2012. ACM.

[66] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger Peng Yu, and Martin Abadi. Host fingerprinting and tracking on the web:privacy and security implications. 2012.