Timing and Latency Characteristics in Disaggregated Systems

by

Anurag Dubey

B.Tech., Birla Institute of Technology, Mesra, India 2013

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Electrical, Computer, and Energy Engineering

2017

This thesis entitled:
Timing and Latency Characteristics in
Disaggregated Systems
written by Anurag Dubey
has been approved for the Department of Electrical, Computer, and Energy Engineering

_____

Prof. Eric Keller

_____

Prof. Eric Wustrow

_____

Prof. Shivakant Mishra

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Anurag Dubey, (M.S., Electrical and Computer Engineering)

Timing and Latency Characteristics in

Disaggregated Systems

Thesis directed by Prof. Eric Keller

In this dissertation, we evaluated two disaggregated systems - Software Defined Networks (SDNs) and Network Function Virtualization (NFVs) and explored the effects of disaggregation. The disaggregation in SDNs leads to timing side-channel information leaks, the result of which exposes the network configuration and flow information to the adversary. We evaluated this attack on real hardware and developed a countermeasure proxy which normalizes the network response time thereby plugging this side channel. Disaggregation in NFVs on the other hand leads to a very flexible and scalable architecture. The only caveat in the new design of NFVs is introduction of extra latency which is introduced because of disaggregation. To address this issue, we introduce a new directory based architecture which, while maintaining logical disaggregation of data, brings the data closer to the nodes which need access to their data.

Dedication

To my parents and brother and all my friends.

Acknowledgements

I would like to express my gratitude to my thesis advisor Prof. Eric Keller for his guidance and untiring support and for keeping me motivated throughout my two years at CU. I also wish to thank Prof. Eric Wustrow and Prof. Shivakant Mishra and for serving on my committee and for their valuable feedback on my work. I am also grateful to John Sonchack and Murad Kablan for helping me out with resources and showing me the ways of research.

I express my deepest appreciation for the faculty at CU who have helped me develop a strong knowledge base and acquire skills and mindset required for research throughput the coursework. I also thank my family, who have always been a source of encouragement and have inspired me to follow my dreams. Last but not the least, I would like to thank all my friends, who have always encouraged me with their support.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Disaggregation means breaking into pieces. A disaggregated system is thus what we get when we take a whole system and redesign it in such a way that its constituent components become independent entities connected by interconnects. A disaggregated system connects these entities in such a way that the overall system still behaves like the original system with many benefits. Separation of entities means separation of concerns. With the separation, our design also becomes more modular. If one entity is much more critical to a system compared to others, then it can be handled independently of other parts of system. This also brings a lot more flexibility into the disaggregated design as compared to traditional designs.

In terms of network based architectures, there are two primary systems that utilize the principle of disaggregation:

- Software Defined Networks
- Network Function virtualization

Software Defined Networks Software Defined Networking (SDN)[7] has radically changed network design, replacing fixed functionality elements (e.g. switches, routers, firewalls, address translators) with generic network elements (i.e. the data plane) that forward packets at high speeds by matching them against simple forwarding rules. A centralized control server (the control plane) manages and updates the rules on switches based on the functionality required by the network. Controllers, in particular, may perform more advanced packet processing as needed, es-

pecially when packets arrive at a switch that does not have a matching rule installed. In those situations, the packet information is forwarded from the data plane switch to the control plane controller for further processing. The controller makes a forwarding decision for the packet and optionally assigns new rules to the switch to handle further packets of the same pedigree based on specific source-destination-port information or on pattern matching, such as on sub-net information. This design offers tremendous benefits and flexibility to network operators because now

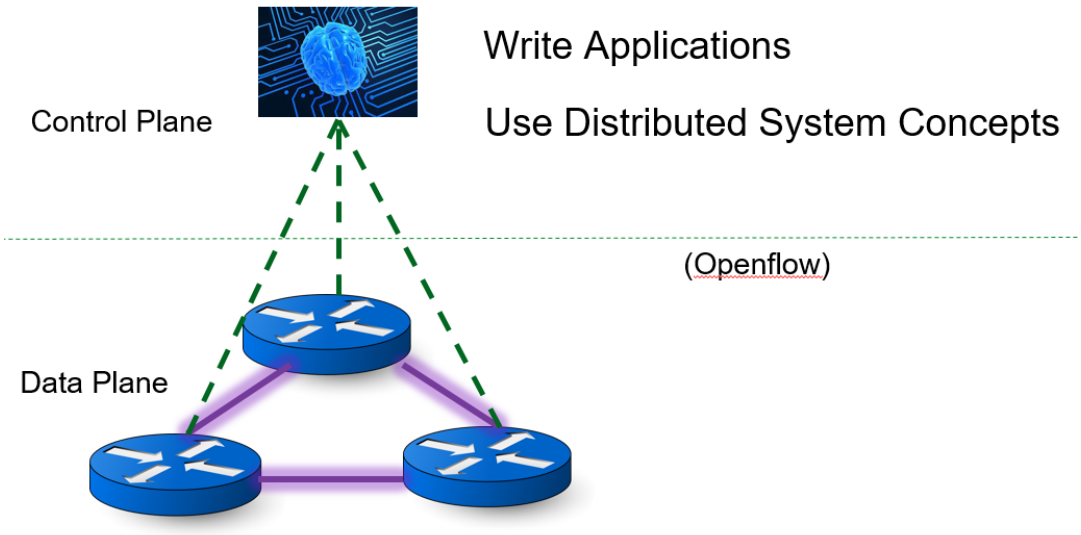


Figure 1.1.0: SDN architecture

we can implement higher level logic at controller level while not affecting the data plane. We can use all the benefits that come with cloud based systems to our advantage and seamlessly scale and manage the network all from one place. However, disaggregation in these networks lead to new security challenges. One important challenge, is ensuring that the elements of an SDN do not leak sensitive network configuration or usage information. Analyzing such behaviors is one of the focus of this research. In this research, we developed a sophisticated timing based side channel attack that can be launched by an adversary with access to only a single machine on the target network. Using the attack, the adversary can learn many more details about a network configuration than reported in prior work, without initiating connections to other hosts in a network. Much of the

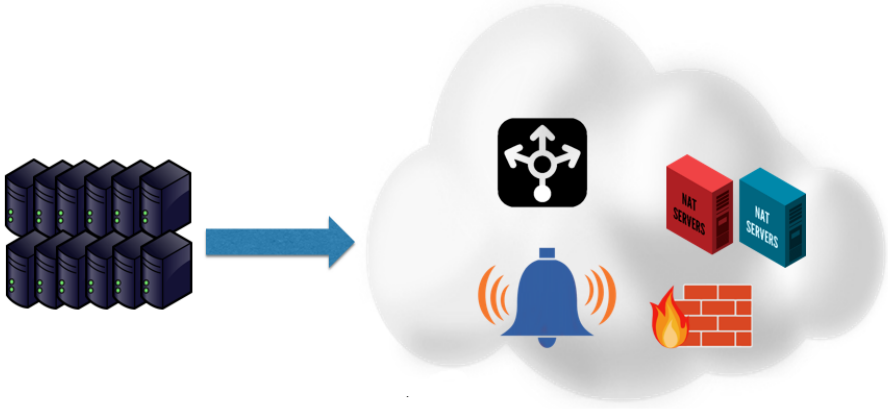
revealed information would be considered highly sensitive, including host communication records, network access control configurations, and network monitoring policies. At a high level, the attack works by estimating a control planes load by injecting timing pings into the controlled network. These pings are specially crafted such that the switch must invoke the control plane to forward them. At the same time, the attacker sends a second stream of test packets into the network. By comparing the turnaround time of the pings to previously collected baseline samples, the attacker can deduce whether the test packets were processed by the controller or simply forwarded by the data plane, and further, determine whether the controller installed new rules in response to the test packets. With a few trials using different test streams, an attacker can learn which flow rules are installed in the data plane and which sequences of packets cause the control plane to install new rules. Depending on the SDN application, this attack may also reveal security sensitive details about the network, such as:

(1) The host communication graph of which devices in the network communicate with each other (if the network is running a common layer 2 MAC learning application)

(2) the access control lists of the switch specifying which traffic flows the switch is configured to drop (if the network is running an access control application)

(3) and the usage of monitoring and packet counting rules to collect flow records on the switch (if the network is running a monitoring application)

Network Function Virtualization Network Function virtualization (NFV) [42] is often associated with software designed networks as another aspect of networks which has seen a dramatic departure from traditional approaches. NFV allows us to deploy network devices like load-balancers, firewalls, IDS, NATboxes etc on traditional commodity servers as virtual machines or containers. This obviates the need to buy dedicated network hardware making it very cheap and effective to deploy these systems in software. It also allows us to employ the benefits of virtualization and cloud based systems for network devices. StatelessNF System [33] proposes, a new architecture for network functions virtualization, where it decouples the existing design of network functions (e.g.,



[!p]

Figure 1.2.0: Network Function Virtualization

firewalls, NAT, load balancers) into a stateless processing component along with a data store layer. In breaking this coupling between state and processing, StatelessNF allows network functions to achieve greater elasticity and failure resiliency. The, StatelessNF system considers the data store as a remote node. Thus, an added latency is naturally introduced: reading from remote memory versus local will always be slower. To achieve acceptable performance, StatelessNF leverages advances in low-latency systems such as RAMCloud [36] where read and write operations are less than 100us. However, in applications where state needs to be updated for every network packets (e.g., traffic counters, timers) such delays can greatly affect network applications. As a part of this research we explored an integration of the data store nodes and the network function nodes, or more generally, a rethinking of the concept that the data store is separate from and independent of the nodes accessing the data store. This has two main advantages that must be considered, along with the potential for negative impacts such as reduced fault tolerance. First, similar to a cache within each node, it can reduce latency and increase bandwidth to access data. Unlike a cache, the data store nodes functionality is to replicate data for resilience, but not provide consistency across all accessing nodes. That is, with a caching architecture, each node accesses data and caches it locally. This, in turn, requires mechanism to maintain cache coherency. With an integrated data store, access to data goes to the nodes actually storing that data (which may be replicated among a few nodes, and coherency needs to be maintained between that small subset of nodes). This subtle

difference makes this more scalable.

Second, if we do not use replication for fault tolerance and have a one-to-one mapping of nodes and data-store, this effectively reproduces the architectures that use migration of data from within the network functions to other instances. It does so, however, with a general data store, moving the burden from every network function implementation to a common data store (which, of course, would require the data store to include the ability to control data placement). However, such integration will not be that simple as multiple questions will arise to identify what and where to place data store nodes. Such questions are:

(1) Should there be a data store instance on every server that hosts a network function?

(2) Can we efficiently replicate data in a common infrastructure?

(3) What is the performance benefit of such an approach and what is the penalty in terms of failure resilience in such an approach?

(4) How do we place data to ensure optimize access across multiple instances?

To summarize, Our research had two primary objectives:

- Study the effect of timing attacks which arise because of disaggregation in Software Defined Networks. Figure out the weaknesses in the architecture and propose their solutions.
- Study the disaggregated Network Function Architecture and propose and evaluate a solution for high latency problems.

Chapter 2

Related Work

There has been a ton of work already in the fields of both Software Defined Networking[7] and Network Function Virtualization[42]. Here we analyze other research work which is close to our focus of research in this dissertation.

2.1 SDN

Side channel attacks. There is a large body of work on using side-channels to leak secret information from computer systems, most prominently for recovering secret cryptographic keys [25, 23, 44]. Many side-channel attacks are based on **timing**; an adversary analyzes the timing of execution [26] or caches [44] to learn details about the code that is executing and what data it is operating on. Our timing-based side channel attack is similar to previous work that times **execution**, although execution timing has not previously been applied to the SDN domain.

Previous work has demonstrated that timing-based side channel attacks can be used remotely [11], to recover details about a host's operating system [27], expose private web pages [9], and to recover cryptographic keys [10] across local and wide area networks. Our attack is designed to work across a local network, though based on these previous studies and our tests, it seems likely that with more advanced techniques, future SDN timing attacks could also be effective against wide area SDN deployments, as they become more widespread. [21, 18].

OpenFlow side channel attacks. A SDN network is topologically similar to a traditional computer network, but behaviorally much more complex, with many different components that

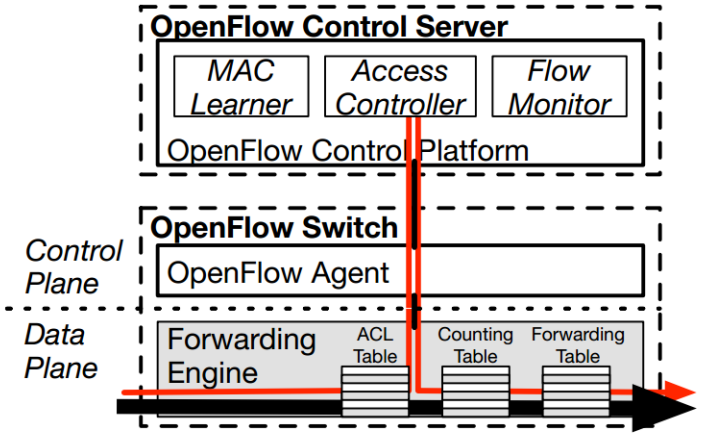


Figure 2.1.0: OpenFlow divides a network into a **data plane**, that forwards packets quickly using simple tables, and a **control plane**, that manages the data plane using more complex actions.

could be targeted with a side channel attack. OpenFlow [31] is the de-facto standard for SDN: it defines a packet processing model, API, and remote management protocol for switches. Figure 2.1.0 summarizes the architecture of an OpenFlow Network, which can be divided into a **data plane** and a **control plane**. The data plane processes most of the packets that a switch forwards by matching packet headers against rules in flow tables that define simple actions (the green line in Figure 2.1.0). The control plane processes packets that the switch does not know how to handle, and can install new flow rules into the data plane for future packets (the red line in Figure 2.1.0). The data plane is implemented as a highly optimized **forwarding engine** in the switch. For physical switches, this is implemented with specialized hardware. For virtual switches [35], the forwarding engine is often a kernel module. Forwarding engines do not directly implement the OpenFlow API. Instead, an OpenFlow Agent, which runs as software on the switch (even physical switches), translates OpenFlow messages from the control server into a lower level forwarding engine API. The control server implements more complex logic, such as MAC learning when a new device is connected to the network, managing the access control policies of all switches in the network, or configuring flow monitoring rules on switches There has been little research into applying timing-based side channel attacks to Software-defined networking. To our knowledge, there have only been three previous

works in this area. Kloti et al. [24] developed a simple attack which measures the set up time of sequential TCP connections to determine if an SDN is using forwarding rules that aggregate TCP connections. Shin and Gu [38] proposed a SDN scanner that measures the response time of pings to determine whether or not a network is using SDN. More recently, Leng et al. [30] designed an attack that measures the response time of requests to determine the approximate capacity of an OpenFlow switch's forwarding table (**i.e.** how many flow rules it can store). All of these attacks can be viewed as **cache-timing** attacks, where the switch's flow table is the cache and the goal is to determine if a legitimate request is processed by the cache or not. In contrast, our attack times **control plane execution**, where the goal is to determine if **arbitrary** packets crafted by the attacker get processed by the control plane. Our approach reveals additional, more security sensitive information about an OpenFlow network.

Additionally, the existing literature on SDN timing attacks uses software based networks, in the Mininet environment [29] with virtual switches [35]. In practice, hardware based OpenFlow switches manufactured by vendors such as Broadcom, Cisco, Juniper, Brocade, and others are widely deployed. These physical devices have vastly different architectures and performance characteristics compared to virtual OpenFlow switches: latencies and packet forwarding rates are orders of magnitude better [6]; but flow installation rate and flow table capacity are orders of magnitude worse [8]. In this work, we perform extensive testing with physical OpenFlow switches, and provide a first look at the effectiveness of timing attacks with a new degree of experimental realism.

attacks. Kloti et al. [24] speculate that there are likely to be detection or randomization based defenses against SDN timing attacks and Leng. et al. [30] briefly suggests further research into attack detection and automated flow table maintenance. We have taken the next step by designing a defense against the timing attack, implementing it as software that runs on a physical OpenFlow switch, and experimentally demonstrating its effectiveness.

2.2 NFV

The problem of network architecture design which is resilient in times of failures and scaling is not new and quite a few approaches have been proposed to tackle this problem. Three approaches proposed in the research community stand out.

First, pico replication [40], which is a high availability framework that frequently checkpoints the state in a network function such that upon failure, a new instance can be launched and the state restored. To guarantee consistency, packets are only released once the state that they impacted has been checkpointed leading to substantial per-packet latencies (e.g., 10ms for a system that checkpoints 1000 times per second, under the optimal conditions).

To reduce latency, another work proposes logging all inputs (i.e., packets) coupled with a deterministic replay mechanism for failure recovery. In this case, the per-packet latency is minimized (the time to log a single packet), but the recovery time is high (on the order of the time since last check point). In both cases, there is a substantial penalty and neither deals with scalability or the asymmetric routing problem.

The third approach which is the focus of our research proposes making the NF architecture stateless [33] by breaking the coupling between NFs and their dynamic state into separate components connected via a high throughput and low latency interface. As shown in Figure 2.2.0, this results in separations of concerns and a network function thus only needs to process network traffic and not worry about things like state replication, high availability etc. On the other hand a data store provides the residence of state. Since the data store could become a bottleneck in packet processing, it must also provide low latency access to data. With this approach, they picked a distributed key-value store (RAMCloud) to store the data in DRAM for low latency access. The data store acts as common storage for all the network functions. So in times of NF failure, no state is lost and scaling up or down becomes as simple as starting/stopping containerized network function instances.

RAMCloud [36] is a high speed in-memory database and it can be interfaced with state of the art

network interfaces like Infiniband and RDMA to provide sub 20us latencies. Avg remote read latency being 6us and 15us for durable writes. Another optimization which reduced remote read/write latencies was batching. Batching helps to achieve high throughput by hiding latencies incurred for every individual request. This architecture provides impressive performance without significant latency penalty even without local caching of data. It is much more simple to comprehend and has built in support for asymmetric and multi-path routing. However, there are a few shortcomings that can be improved upon:

- The remote data store means data placement cannot be optimized and every database request will have the same performance.
- The read/write latency can be made even better if we moved the data store closer to network function instances.
- Timers are one thing that should not be reliant upon the network functions. The data store should be able to automatically manage key expires.

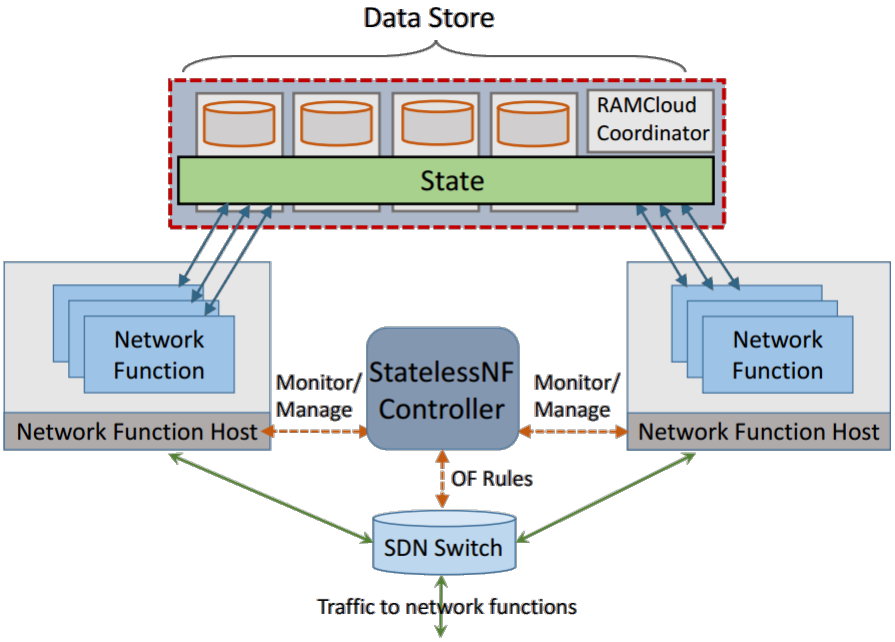


Figure 2.2.0: Architecture of Stateless Network Functions

Chapter 3

Software Defined Networks

In this chapter, we develop and study a timing side channel that can be exploited by an adversary to extract network information from within a network[1] .

3.1 Threat Model

The threat model assumes the adversary has control of a host in the network, and seeks to learn about the network without needing to compromise any of the network infrastructure or other hosts. This threat model matches two real world scenarios where an attacker may wish to learn information about the network configuration. First, an intelligent adversary performing a complex, multi-staged attack who has just gained access to one host in the target network through malware or social engineering and now wishes to plan subsequent stages. For example, consider the recent Target data breach [22], where attackers installed malware on point of sale terminals to collect credit card numbers; or the more recent breach at Juniper [20], where adversaries installed back doors into source code stored deep within Junipers network. In scenarios like these, the ability to learn sensitive information about the network in a reconnaissance could greatly benefit attackers. A second scenario that fits this model is a malicious user of a shared network (such as at a data center or cloud provider) that may wish to learn about other users of the network via the network configuration settings of the switch. As we will show, the timing attack is able to deduce host communication pairs as well as collect information about SDN monitoring applications.

[1] The work in this chapter has been accepted and presented at a security conference (ACSAC) in 2016

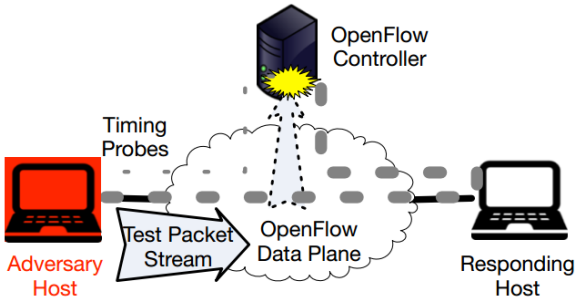


Figure 3.1.0: Summary of our control plane timing attack: an adversary times the control planes execution while sending test packets into the network. If the test packets put load on the control plane, the attacker will observe longer response times with the timing probes.

3.2 Attack

Figure 3.1 summarizes our control plane timing attack. The attacker learns about the network by performing trials in which they send a stream of timing probes and a stream of testing packets into the target network, and then comparing the timing probe RTTs to a previously collected baseline sample to learn some details about the networks configuration. We assume that the attacker has control of a single host on the network and can inject packets into the network at will, but does not want to disrupt network performance in an end-user detectable way. An attacker performs a series of trials. In a trial, an attacker: (1) collects a baseline round-trip- time (RTT) measurement while transmitting a stream that has a known impact on the control plane; (2) transmits a test stream to get a sample RTT measurement; and then, (3) compares the two RTT measurements. Based on the comparison (using a simple t-test) of the two distributions of RTT measurements, the attacker then attempts to deduce whether the control plane installs new flow rules into the data plane in response to the test packets or if other control level actions were taken. The timing probes are request packets to hosts or devices on the network that cannot be forwarded without invoking the control plane. Timing probes essentially act as control plane pings, and their RTT value informs the attacker of how long the controller takes to process packets at particular points in time. The test packets are spoofed packets that all have the same value for one or more packet

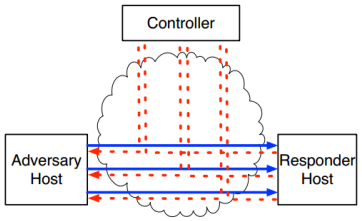


Figure 3.2.1: Timing the control plane with spoofed ARP requests to a host

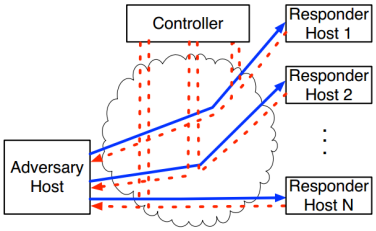


Figure 3.3.1: Timing the control plane with legitimate ARP requests to multiple hosts

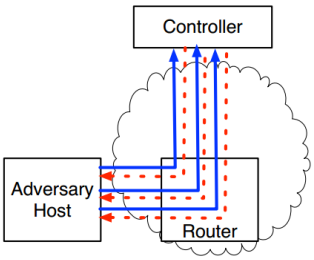


Figure 3.4.1: Timing the control plane with low TTL IP Packets

header fields to control for the kind of inference the attacker wishes to perform. Depending on how the timing probe RTT changes when transmitting a testing stream, the attacker can infer the role that the control plane plays in handling packets with those header values. By performing repeated trials with different test packet streams, the adversary can discover more and more information about the network configuration.

3.2.1 Timing Probes

Timing probes are specially crafted request packets that the adversary sends into the network to learn how long the control plane takes to process packets at specific points in time. Timing probes must have three properties: first, they must evoke a response from a device in the network, so that the adversary can compute a RTT for each timing probe; second, they network must not be able to forward the packet without invoking the control plane, so that controller processing time is a factor in the RTTs of the timing probes. Many types of requests can potentially act as timing probes, depending on the configuration of the targeted network. For example, in a naive network that sends all packets to the control plane, a TCP syn packet to any legitimate host in the network would be a timing probe. Below, we describe three types of timing probes: two timing probes that can be used on a Layer 2 network (i.e., where switches forward based on MAC address), and one that can be used on a Layer 3 network (i.e., where switches forward based on IP address).

Layer 2 Timing Probe: Spoofed ARP Requests Layer 2 networks, both traditional and SDN,

typically forward packets using MAC learning, a simple algorithm to associate MAC addresses with physical ports on a switch. Whenever a switch receives a packet from a device with a MAC address that is not present in its forwarding table, the switch sends the packet to the controller, which saves the (input port, MAC address) pair. When the controller receives a subsequent packet destined for that device, it instructs the switch to forward the packet out of the port where the device was observed, and installs a corresponding rule into the switchs forwarding engine. Traditional networking equipment used flow rules that mapped a destination address to an output port on the switch. Most OpenFlow implementations use finer grained flow rules that specify both the source and destination address because it provides the controller with greater visibility and allows additional path optimization techniques [2] . 1 . In networks that use MAC learning, an adversary can use ARP requests as timing probes. ARP is widely used protocol for mapping IP addresses to MAC addresses. The adversary selects a host on the network that uses ARP, then sends it ARP requests with spoofed source MAC addresses. 3.2.1 depicts what will happen when the adversary sends the ARP request packets into the network. The ARP requests themselves will not invoke the control, since ARP requests are broadcast, which a forwarding engine can do without invoking the control plane. However, the ARP reply, which will have the randomly generated address as its destination MAC, will need to invoke the control plane because the switchs forwarding engine will not have a rule mapping the randomly generated address to a port. It is important to not overload the network with spoofed MAC addresses as this might be noticeable (and detectable) to a network monitor. It may also degrade the performance of the network, something the attacker wishes to avoid as it may affect the ability to infer information from the network. Fortunately, timing probes are useful even at extremely low rates, because each probe provides a useful measurement and just a few is sufficient to notice variations in RTT rates when a testing stream is introduced. For example, in all of our experiments in Section 3.3 a timing probe at a rate of 10 per second was ample for inferring sensitive information. On a large network with many Layer 2 connected devices (as in a data center), there should be a reasonable amount of ARP traffic such that timing probes

[2] In a traditional, pre SDN network, the switch's CPU runs the controller level MAC learning logic.

will cause minimal interference with normal network activity.

Layer 2 Timing Probe: Legitimate ARP Requests In some layer 2 networks, it is not possible to send packets into the network with random MAC source addresses due to network access control systems that either drop packets from MAC addresses that do not belong to pre-authorized devices or limit the number of devices that can be connected to each physical port on a switch or router [13]. Figure 3.3.1 illustrates a technique to generate ARP based timing probes that works around this defense by taking advantage of forwarding rule timeouts: switch forwarding tables have limited memory and, as a result, are usually programmed to delete forwarding rules if they have not been used for more than a threshold period of time. To use this approach, the attacker selects a set of N hosts to act as ARP responders and sends them each one legitimate ARP request, in sequence. Assuming all the rules that forward packets from the ARP responders to the attacker have timed out, each of the ARP replies will need to be processed by the control plane so it can re-install the forwarding rules. An adversary would not need a large number of responder hosts to use this approach because, as our evaluation demonstrates in Section 3.3 , timing probes only need to be sent at a very low rate and for only a short amount of time (i.e., 10 per second for approximately 5 seconds).

Layer 3 Timing Probe: Low TTL Packets SDNs can also be configured to forward at the IP level (e.g., programming OpenFlow switches to act as routers [5]). This allows an adversary to use IP packets with low TTL values as timing probes. OpenFlow switch cannot generate ICMP packets in their forwarding engines and must send packets with expiring TTL values to the controller so that it can generate the correct response. As Figure 3.4.1 depicts, when an adversary sends IP packets with TTl=0 into the network, the first network element configured to act as a router will send the packet to the controller, which will generate the appropriate ICMP response and send it back to the adversarys host.

3.2.2 Test Packet Streams

A test packet stream is a sequence of packets with a small invariant that the attacker sends into the network at a constant rate to determine if the data plane needs to invoke the control plane for the given invariant. If the control plane is invoked that provides information for some configuration of the network. We define a test packet stream with a template for an invariant selection that specifies either a constant or randomly selected value for the Ethernet, IP, and TCP/UDP header fields of each packet. The attacker can determine the role that the controller plays in forwarding the test packets by analyzing the RTTs of timing probes that are ongoing during the testing stream. Based on the measurements from the RTT the following deductions could be made:

(1) If the data plane contains a rule that matches the packets in the test stream, it will place no additional load on the controller, and the timing probe RTTs will be low during the test stream.

(2) If the data plane does not contain a rule that matches packets in the test stream, the packet will be forwarded to the controller placing a moderate load on the control plane, and introducing some delay to the timing probes.

(3) If the data plane does not contain a rule that matches the forwarding packets in the test stream, and the control plane installs new forwarding rules into the data plane in response to each packet in a test stream, there will be a heavy load placed on the control plane as flow rule installation is an expensive operation [3] , adding much more latency to the timing probes than the previous scenario.

3.2.3 Detecting RTT changes

An attacker could learn what effect a test stream has on the network by comparing the distribution of probe RTTs observed while transmitting the stream with a sample of baseline RTTs

[3] Many OpenFlow devices only support a flow installation rate of ¡100 per second [8].

representing a known effect. A simple method for this that we found effective is a straightforward application of the Students t-test [43], a statistical test that compares if two samples are drawn from the same distribution. A t-test produces a t-statistic that measures the size of the difference between the samples relative to variation in the sampled data. A t-statistic can be converted into a p-value, which speaks to the likelihood that the two samples share the same distribution. Typically, a p value less than 0.05 is considered an indicator of statistically significant difference between the samples in that they are truly measuring different distributions. One baseline stream is simply a list of Ethernet packets with source and destination addresses that are chosen at random, without repetition. The controller will process each packet in this stream, to perform MAC learning, but will not install a rule since it will never receive a packet destined for that address. By using a t-test to compare the RTTs observed while transmitting a test stream to the RTTs observed while transmitting this baseline stream, an attacker can conclude:

(1) If the p-value is high, the timing probe RTTs likely share the same distribution as the baseline sample, so like the baseline stream, the test stream is likely processed by the control plane, but does not cause additional flow rule installations.

(2) If the p-value is low and the t-statistic is negative, then the timing probe RTTs likely have a different distribution than the baseline samples and a much lower mean, so the test stream packets are likely not processed by the control plane.

(3) If the p-value is low and the t-statistic is positive, then the timing probe RTTs likely have a different distribution than the baseline samples and a much higher mean, so the test stream packets likely caused additional flow rule installations.

3.2.4 Example Attack Scenarios

Learning host communication patterns. An attacker can learn if two devices (A and B) have recently communicated with each other by performing a trial that tests the presence of a forwarding rule that handles traffic between A and B. To do so is a straightforward application of

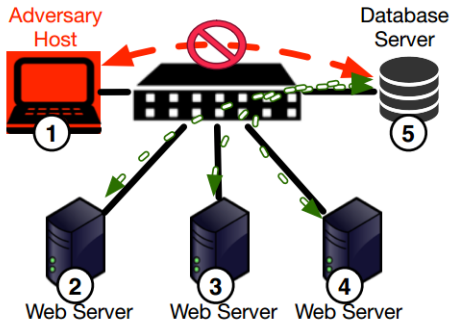


Figure 3.5.4: An example application of the timing attack: an adversary can learn which devices communicate with a back-end database server that the adversary cannot directly connect to, due to firewalling.

the procedures described previously: the attacker records a baseline RTT and then measures the RTT after injecting a test stream with spoofed host address for A with destination B. If there is not a significant change in timing information, then controller must not be involved in forwarding, and there is likely a rule installed. Learning host communication pairs may reveal broader information about the network that is especially useful in a multi-staged attack. It can help the adversary understand the purpose of devices that cannot be directly probed (e.g. due to firewalling). Consider the scenario depicted in Figure 3.5.4. The adversary's goal is to deduce which device is the database server for the networks web server. The adversary can collect the MAC addresses of all devices on the network by monitoring ARP requests, and open TCP connections with the web servers (#s 2 through 4 in Figure 3.5.4) in the network. However, the adversary cannot open TCP connections with the database server (#5 in Figure 3.5.4 because the switch drops all TCP packets destined for the database unless they come from a web server. The adversary can determine that server 5 may be a database server by using our timing attack to learn that there are rules installed on the switch that forward traffic from each of the web servers to server 5.

Learning ACL entries. One common application of OpenFlow is to provide access control, similar to a traditional firewall. This is generally implemented by configuring a switch to sequentially match packets against two tables: first, an access control table that can drop packets by

Priority	Source IP	Destination IP	Policy
10	1.1.1.2	1.1.1.5	**ALLOW**
10	1.1.1.3	1.1.1.5	**ALLOW**
10	1.1.1.4	1.1.1.5	**ALLOW**
1	*	1.1.1.5	**DROP**
DEFAULT	*	*	**ALLOW**

Table 3.1: An example ACL policy installed into the switch in Figure 3.5.4, which blocks connections to the database from all hosts besides the web servers.

matching them against entries specifying Layer 2 through 4 headers; second, a forwarding table that uses a standard algorithm, such as MAC learning, to select which port to send packets out of. For example, the ACL policy depicted in Table 3.1 could be used in the example scenario in Figure 3.5.4 to block connections to the database server unless they originate from one of the networks web servers. An adversary can learn that these entries exist in the ACL by performing trials of the attack with test packet flows that use spoofed addresses, as follows:

(1) First, the adversary chooses a MAC and IP source and destination addresses that do not exist in the network and performs an attack trial using test packets with those addresses. The packets will not match any entries in the ACL or forwarding table, and the switch will send them to the control plane for a forwarding decision, causing a measurable high RTT skew.

(2) Next, the adversary performs trials with the same MAC addresses but different IP address pairs. If the IP address pair is blocked by the ACL, the adversary will observe a measurably low RTT in that trial (compared to the previous) because the ACL table will drop the packets before they reach the MAC table. If the IP address pair is blocked by the ACL, the adversary will continue to observe the high RTT values from the first trial, as the packets will pass through the ACL to the MAC learning table that sends them to the controller.

Learning about monitoring controllers. OpenFlow networks are capable of running security monitoring applications alongside their forwarding logic, as proposed in [39], [41] and [17]. These applications are generally implemented with a **counting table** that simply counts the number of

packets and bytes from each IP address or flow. When a packet matches a flow in the counting table, the flow's counter is incremented and the packet is sent to the next table in the forwarding engine's pipeline (**e.g.** the forwarding able). When a packet does not match an entry in the forwarding table, the data plane sends the packet up to the controller which installs the appropriate counting rule. Periodically, the controller polls the forwarding engines for statistics about all of the flows in the counting table and performs analysis on the aggregated data.

An adversary can learn at least two important details about an OpenFlow monitoring application using our timing attack: whether the network is monitoring at the host or flow level and how frequently the controller receives updates from the switch.

To determine if a network is monitoring at the host or flow level, an adversary can build an **all pairs test stream** by generating a list of N random IP addresses then adding one packet to the test stream for each pair of IP addresses (**i.e.** N^2 packets). The adversary runs an attack trial with this test stream. If the network is not monitoring, the timing probe RTT will not shift when the all pairs test stream is transmitted because the control plane will not install any new rules due in response to the test stream. If the network is monitoring at the **per host** level, the probe RTTs will shift proportionally to $\frac{N}{test\_stream\_rate}$ because the controller will install a counting rule for each new IP address. Finally, if the network is running a **per flow** monitoring application, the probe RTTs will shift proportionally to $\frac{N^2}{test\_stream\_rate}$, the number of unique **flows** the adversary sent into the network per second, because the controller will install a counting rule for each new IP flow.

To determine how frequently the controller polls the switch for updates, the adversary can simply measure the timing probe RTTs over a longer period of time after running the trial described above. Whenever the controller polls the switch for statistics about the counting flows, it will place load on both the switch and controller, temporarily increasing the probe RTT.

3.3 Evaluation

To evaluate the effectiveness of the timing attack, we performed experiments on a physical OpenFlow testbed with real OpenFlow equipment and background traffic. The evaluation is divided

into three parts. First, we measured the accuracy of our attack at inferring changes in the network performance rates. Next, we performed benchmarks to understand the low level details of our attack: the effects of different attack rates, the impacts of background traffic, and determine bottlenecks in the control plane. Finally, we tested the techniques described in Section 3.2 as they are used to learn security sensitive details about a network.

3.3.1 Testbed and Background Traffic

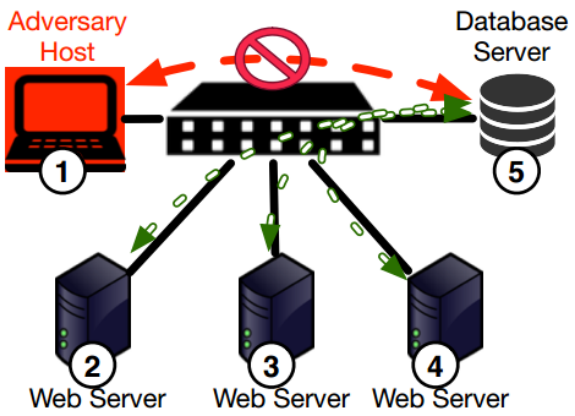


Figure 3.6.1: Testbed Setup

The testbed. Figure 3.3.1 illustrates our testbed network. It contains: a hardware OpenFlow switch, a Pica8 3290 with a Broadcom Firebolt-3 forwarding engine that processes packets in hardware according to OpenFlow rules, a 825 Mhz PowerPC CPU, and 512MB of memory, and runs Debian 7; a control server, a quad- core Intel i7 machine with 4GB of RAM, running Ubuntu 14.04 server LTS and the Ryu OpenFlow controller [4]. We connected three hosts to the switch: h 1 , the adversary controlled host; h 2 , the host that the timing probe sends ARP requests to; and h 3 , a host that replays background traffic into the testbed. Each host has a dual-core Intel Core-2-Duo machines with 2GB RAM. All network connections (i.e. switch to controller and switch to host) were via gigabit ethernet. Background traffic. To model real network conditions, we replayed

a background trace from NCCDC 2015 [1], a three day cyber-defense competition in which 10 teams compete by defend ing networks composed of real devices against human adversaries. Each teams network contained 8 servers and 6 workstations running a mix of Windows, Linux, and BSD, 1 VoIP phone, 1 Juniper EX2200 switch, and 1 Juniper SRX210 gateway. The 10 teams were all connected to a core switch, also a Juniper EX2200. The Juniper switches are an equivalent class of hardware as our Pica 8 3290 switch, but do not use OpenFlow. The trace contains all the full, unanonymized packets that passed through the core switch. On average, the trace had a throughput of approximately 80Mbps and 10,000 packets per second. The traces are publicly available [3].

Attack parameters. Unless otherwise noted, we used the spoofed ARP timing probes described in Section 3.2.1 with a rate of 10 probes per second, a test stream rate of 500 packets per second, and a test stream duration of 5 seconds.

3.3.2 Attack Effectiveness

To measure the accuracy of the attack in trials we used the t-test technique described in Section 3.2 to determine whether a test stream was processed by the dataplane, processed by the control plane, or caused new flow installations. The experiment was executed over 300 trials in which each scenario was equally represented by test streams and compared against a baseline sample of RTTs collected while transmitting a test stream that caused control plane processing, collected prior to the trials. Figure 3.7.2 shows the attacks accuracy in these trials, as we vary the p-value threshold used to distinguish between statistically significant RTT samples. At a test flow rate of 500 packets per second, the attack correctly classified all test streams with >99% accuracy for a large range of thresholds ranging from .1 to .000001. At a much lower test flow rate of 50 packets per second, the attack had difficulty distinguishing between data plane processing and control plane processing. However, due to the high load that flow installation places on the control plane, it was still possible to correctly identify whether or not a test stream caused new flow installations with a p-value threshold in the .1 to .000001 range. This suggests that if an attacker wishes to minimize

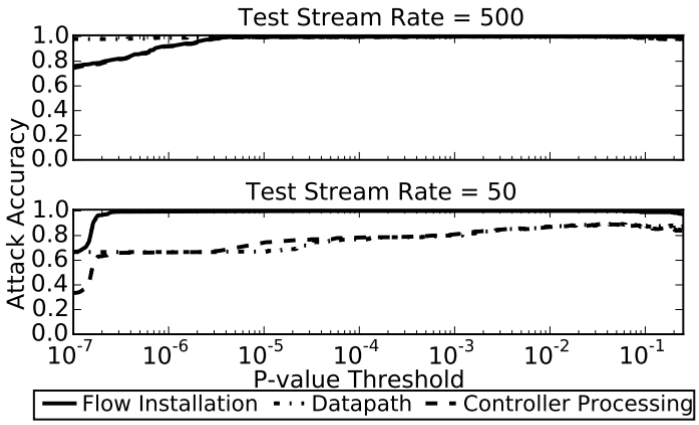


Figure 3.7.2: Attacker accuracy in predicting the control planes role as p-value threshold varies.

their impact on the network, they can first send a test stream into the network at a low rate, to determine if it causes flow installations, and then when they are confident that it does not, send test streams at a higher rate to learn if it is processed by the control plane or passes through the data plane without controller interaction. Given this wide range of thresholds, it would likely be straightforward for an adversary to calibrate the attack to a target network. A common p-value to indicate significance in many domains is .05, which was well within the range of thresholds that had very high accuracy in our trials. An adversary could also estimate a range of thresholds that yield high accuracy by measuring the difference, in terms of p-value, between RTT samples taken while transmitting a test stream known to be processed by the data plane vs. RTT samples taken while transmitting a test stream known to be processed by the control plane.

3.3.3 Attack Fundamentals

In this Section, we study at the attack at a lower level to better understand the factors that make it work.

Test Stream Rate Figure 3.8.2 shows probe RTTs as the rate of the test stream varied in trials where the controller was configured to make a forwarding decision for each test stream packet. Figure 3.9.2 shows RTTs in trials where the controller also installed a rule in response to each

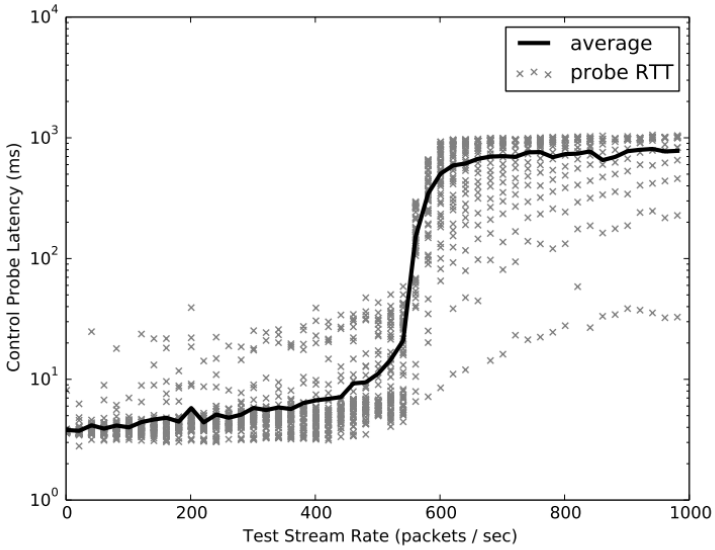


Figure 3.8.2: Timing probe RTT as test stream packet rate varies, for test streams that are processed by the control plane.

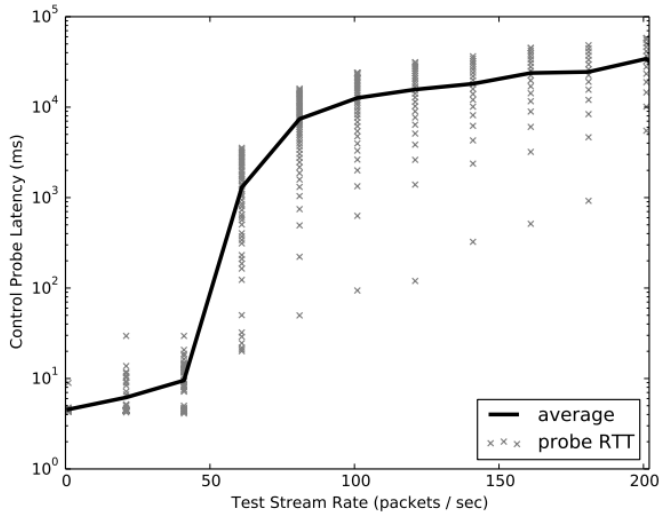


Figure 3.9.2: Timing Probe RTT as test stream packet rate varies, for a test stream where each packet invokes a rule installation.

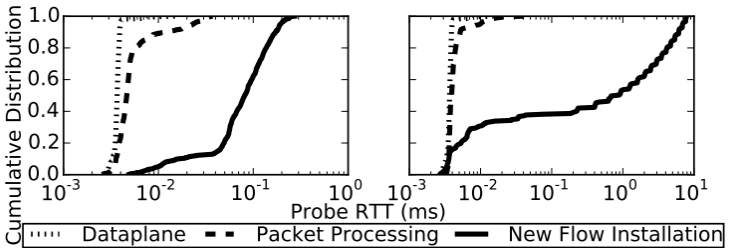


Figure 3.10.3: Empirical Timing Probe RTT distributions while sending test streams at 50 and 500 packets per second.

packet in the test stream. Figure 3.10.3 shows an alternate view, empirical distributions of timing probe RTTs in different scenarios for test stream rates of 50 (left) and 500 (right) packets per second. Both control plane operations had a significant effect on the distribution of timing probe RTT. However, flow installation had a much larger impact. When the controller processed 500 packets per second, the RTT approximately doubled. When the control plane installed a flow rule for each packet, the RTTs doubled at approximately 50 packets per second. We also observed that the distribution of RTTs changed very quickly when transmitting test flows that placed load on the control plane, with average RTTs increasing in under 1 second.

	Packet Processing						**Flow Installations**					
Rate	**10**	**30**	**50**	**70**	**90**	**200**	**10**	**30**	**50**	**70**	**90**	**200**
Switch CPU	11.0%	15.7%	25.0%	26.0%	32.0%	65.0%	5.9%	16.7%	28.7%	60.0%	87.0%	99.0%
Ryu CPU	2.6%	6.0%	11.8%	15.0%	19.0%	38.0%	3.3%	10.0%	10.0%	13.0%	10.0%	10.0%
Switch Memory	8.5%	8.5%	8.5%	8.5%	8.5%	8.5%	9.5%	9.7%	9.7%	9.9%	10.0%	10.4%
Ryu Memory	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%
Switch-Ryu BW	0.20%	0.65%	1.05%	1.55%	1.85%	4.20%	0.17%	0.70%	0.72%	1.03%	1.40%	0.35%

Table 3.2: Resource usage of control plane components for packet processing and flow installation at different rates. The switch's CPU is the primary bottleneck, especially for flow installation.

Control Plane Bottlenecks In both experiments, we observed a significant nonlinear jump: from <10ms to >500ms when the stream rate went above approximately 550 packets per second, in the forwarding decision trials; and from <10ms to >1000ms when the stream rate went above approximately 50 packets per second in the flow installation trials. We measured the resource usage of the physical components of the control plane in trials with 5 second long test flows, shown

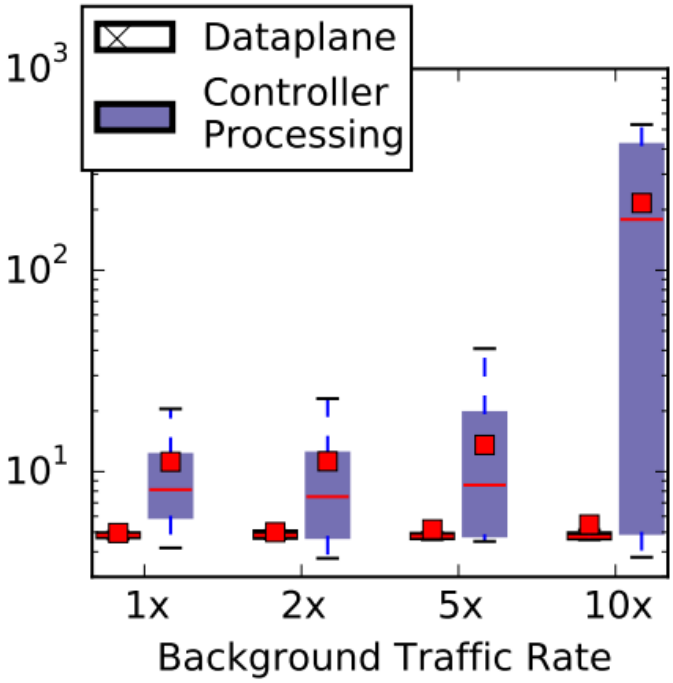


Figure 3.11.3: Probe RTT distributions as background traffic varies, for test streams that cause the controller to process 500 packets per second.

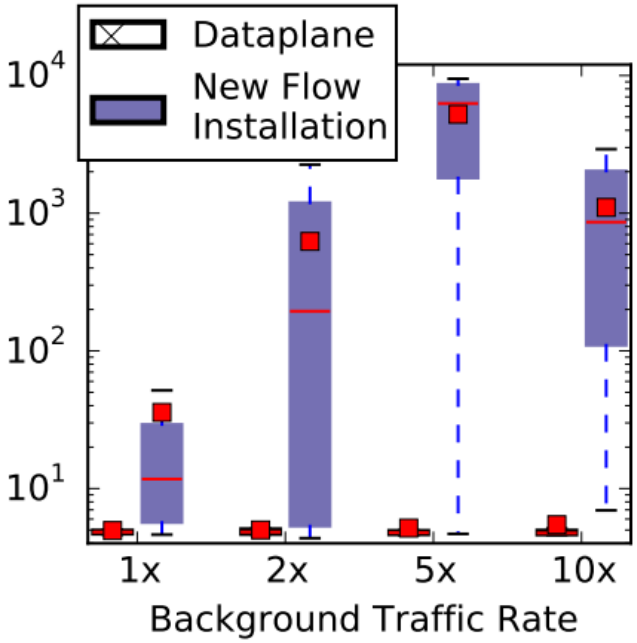


Figure 3.12.3: Probe RTT distributions as background traffic varies, for test streams that cause the controller to install 50 flow rules per second.

in Table 3.2, and concluded that the bottleneck was the switch CPU, in agreement with previous benchmarks [15, 41]. We found that the low level software interconnecting the forwarding engine to OpenFlow agent running on the switch was unoptimized and put extremely high load on the switch's CPU: the forwarding engine driver did not support packet coalescing or polling, as a result each time a packet needed to travel between the forwarding engine and controller, there were many expensive context switch and copy operations.

The impact of background traffic To measure the effect of background traffic on the timing attack, we ran trials of the timing attack while replaying our background trace at different rates. As Figures 3.11.3 and 3.12.3 show, higher rates of background traffic did not weaken the effectiveness of our attack. Conversely, higher background traffic rates actually amplified the effect of the test streams on timing probe RTT the difference between the RTT distributions during dataplane and control plane processing of test streams increased with background traffic rate, making it easier for an adversary to figure out whether the control plane was processing test packets. This effect occurred because timing probes RTTs increased non- linearly with control plane load, and the background traffic placed a small baseline load on it. We also observed that when the background traffic put a heavier load on the switch, the switch began to drop a large fraction of flow installation requests, causing the reduction in probe RTT depicted in Figure 3.12.3 when moving from a 5X to 10X background traffic replay rate. Even in these scenarios, transmitting a test flow still caused a statistically significant shift to timing probe RTTs.

3.3.4 Attack Evaluation Summary

In summary, our results demonstrated that the control plane timing attack was highly effective in a testbed with physical OpenFlow equipment and realistic background traffic:

(1) The attack is highly effective. In trials with test flows with rates of 500 packets per second, we were able to correctly predict what role the control plane played in processing the test flow with >99% accuracy.

(2) Control plane load has a high impact on timing probe RTT: processing only 500 packets per second in the control plane more than doubled timing probe RTT, on average; flow rule installation had approximately 10X greater of an effect. The primary bottlenecks were inefficient software and hardware between the switchs forwarding engine and its OpenFlow software agent that connects to the control server.

(3) Increased background traffic load increased the difference between RTTs observed when the control plane played different roles, benefiting the attack.

(4) With repeated attack trials, adversaries could learn which pairs of devices communicated, infer the entries in a switchs ACL, and uncover the monitoring behavior of a controller with nearly perfect accuracy.

3.4 Defending the Control Plane

The attack proposed in this research is challenging to defend against because of its generality: existing defenses can only protect against specific instantiations of the attack because they do not address the underlying issue of correlation between control plane load and control plane processing time. For example, deploying a MAC based access control system [13] to drops packets from MAC addresses that are not pre-authorized can stop the attack if an adversary uses spoofed ARP timing probes. However, it does not stop adversaries that use other types of timing probes. In this section, we describe a robust and general software based defense to control plane timing attacks that can run on existing physical OpenFlow switches and evaluate it on our testbed.

3.4.1 An OpenFlow Timeout Proxy

The core idea of our defense, depicted in Figure 3.13.0, is to normalize the RTT of attack timing probes to prevent the adversary from learning about control plane load. We implemented this solution using a timeout proxy that is interposed between the switchs forwarding engine and the control server. The timeout proxy tracks each packet that the forwarding engine sends to the

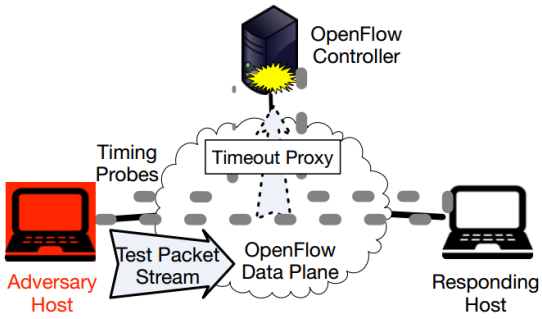


Figure 3.13.0: The timeout proxy sends a default packet forwarding instruction to the switch if the controller doesn't respond within a threshold period of time, normalizing the RTT of any potential adversary timing probes.

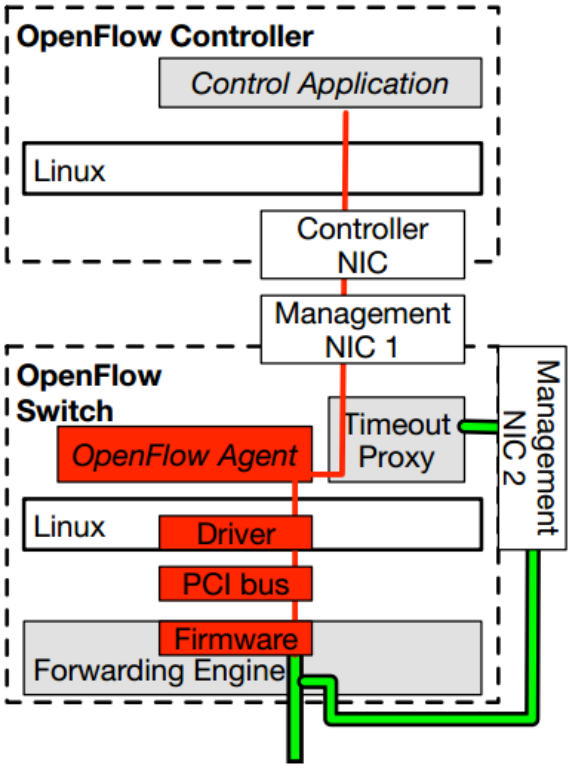


Figure 3.14.0: The timeout proxy avoids the standard bottlenecks between the data and control planes, and quickly sends default instructions to the switch when control requests time out

controller. If the controller does not respond within a threshold period of time, the proxy makes a forwarding decision for a packet by matching it against a table containing default flow rules and sends it to the forwarding engine. Later, if the controller sends a forwarding instruction for the packet, the proxy drops it to avoid duplicate packets. On the other hand, if the controller responds too soon, (i.e. before the threshold period of time has passed), the proxy queues the response until the threshold amount of time has elapsed. The proxy does not interfere with any of the other OpenFlow interactions between the switch and the controller, such as flow installation or polling. The controller installs the default flow rules to the proxy, and sets the timeout threshold that determines how long the proxy waits before considering a request timed-out. Increasing the interval gives the control plane longer to respond, and decreases the number of timeouts that will occur. However, it also increases the round trip time of all packets processed by the control plane. Since the control plane is usually only invoked at flow set up, the threshold can be set quite high without causing a user-noticeable impact. In our experiments, we used a threshold of 11 ms, approximately 3 times the average control plane RTT.

Implementation. Figure 3.14.0 shows how the timeout proxy interconnects with the standard components of an OpenFlow switch. The forwarding engine sends a copy of each controller-bound packet to the proxy via the switchs second management interface. This connection avoids all of the standard bottlenecks between the controller and the switch that we discussed in Section 3.3. The proxy also monitors each packet transmitted between the switch and the controller to determine which requests the controller has responded to. Our implementation encodes forwarding instructions as IP-ToS tags. It maps each possible default action to a unique tag. To send an instruction to the switch for a packet, it places the appropriate tag onto the packet and then sends it to the forwarding engine, where it is matched against a special proxy managed table that takes the appropriate action based on the ToS tag. This switch behavior conforms to all OpenFlow specifications, as OpenFlow switches do not guarantee that they will always respect the controllers forwarding instructions. The proxy is implemented as an efficient C++ application with two threads: a proxy thread that forwards packets between the switch and controller, and maintains a list of outstanding

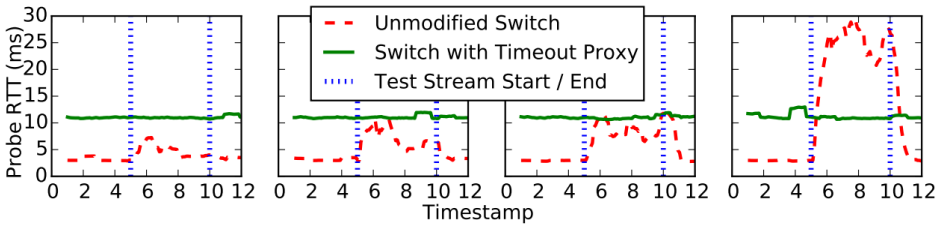


Figure 3.15.1: Probe RTTs while sending test packets that are processed by the control plane, with and without the timeout proxy defense, for trials with test packet rates of 400, 600, 800, and 1000 packets per second

requests to the control plane; and a timeout thread that checks the list for requests that have timed out.

3.4.2 Defense Evaluation

We evaluated our timeout proxy implementation in the OpenFlow testbed described in Section 3.3. The timeout proxy ran directly on our Pica 8 3290 switch, configured to send controller responses to the switch within 10 - 11 ms of the original request. We focused on answering three questions: (1) what effect does the timeout proxy have on timing probe RTTs; (2) does the timeout proxy impact an adversarys ability to accurately predict what actions the control plane is taking in response to test packet streams; and (3) what are the upper limits of the timeout proxy, with respect to how many packets per second it can handle?

Timing probe RTT. Figure 3.15.1 shows the effect of the timeout proxy on timing probe RTTs while an adversary sends test packets into the network that require control plane processing. Without the proxy, latency was low (3ms) during the pre-attack period when control plane load was low, but drastically increased when the adversary host sent a test stream that increased control plane load. The timeout proxy normalized the control plane delay: during the period of light load, before the test stream was sent, it queued control plane responses for 7ms; during the period of higher load, while the test stream was transmitting, it sent a default action to the switch whenever the control plane did not respond within 11 ms. Figure 3.16.1 plots timing probe RTTs while the

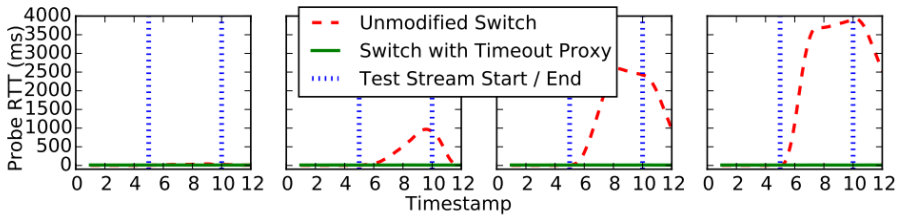


Figure 3.16.1: Probe RTTs while sending test packets that cause the control plane to install new rules onto the switch, with and without the timeout proxy defense, for trials with test packet rates of 400, 600, 800, and 1000 packets per second.

adversary host sent test packets into the network that caused the control plane to install a new flow rule for each test stream packet. Each flow installation puts significant load on the control plane, thus without the timeout proxy there was a large increase to probe RTT when the adversary transmitted the test stream. The timeout proxy again normalized the amount of delay added to the probe RTTs, even in extreme load scenarios that would have added >1000 ms of latency to the timing probes.

Impact on timing accuracy. Figure 3.17.2 shows our attack's accuracy in 300 trials where we used the t-test technique described in Section 3.2 to predict whether a test stream transmitted at a rate of 500 packets per second was processed by the dataplane, processed by the control plane, or caused new flow installations. While using the proxy, there was no p-value threshold that resulted in a high accuracy. This is in contrast to the identical trials that we did without the proxy, where we saw that there was a wide range of p-values that yielded perfect accuracy (Figure 3.7.2). The proxy had this impact on the timing attacks accuracy because it kept the distribution of the timing probe RTTs nearly identical, regardless of what role the controller played in processing the test stream packets, as Figure 3.18.2, a plot of the average ECDFs of each class of test flow with and without the proxy, shows. The similarity between these timing probe RTT distributions suggests that the proxy would be effective against any statistical or machine learning based approach to learn the control planes role by analyzing timing probe RTT differences.

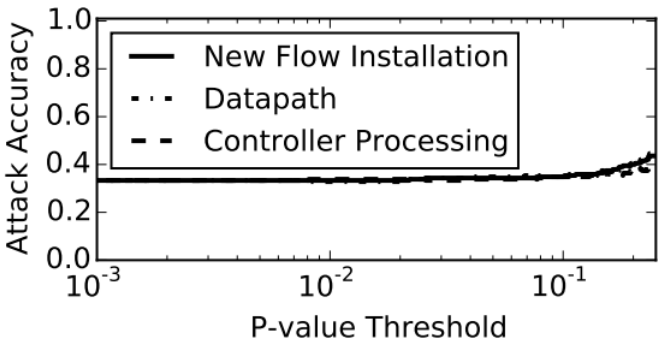


Figure 3.17.2: Attack accuracy while running the proxy.

Rate	100	500	1000	5000	10000
Without Timeout Proxy					
25th percentile RTT	2.83	3.84	18.78	4276.2	–
50th percentile RTT	2.94	4.56	22.38	4460.1	–
75th percentile RTT	3.25	5.22	29.38	4637.8	–
With Timeout Proxy					
25th percentile RTT	10.83	10.63	10.55	10.84	11.97
50th percentile RTT	11.08	10.86	10.85	11.16	15.05
75th percentile RTT	11.27	11.12	11.17	13.35	20.46

Table 3.3: timing probe RTT statistics as attack rate changes.

Upper limits. In our testbed, the current implementation of the timeout proxy was effective against test streams with rates of up to approximately 10, 000 packets per second. Table 3.4.2 summarizes statistics for probe RTTs while transmitting test streams that caused the control plane to process packets. When using the timeout proxy, a test stream of 5, 000 packets per second shifted the timing probe RTT distribution less than a test stream of 100 packets per second did on an undefended network. The timeout proxy performed even better in trials where test streams caused the control plane to install rules. On an undefended network, we found that a test stream that caused 10 flow rule installation requests per second had as much of an impact on probe RTTs as test streams that caused 5, 000 flow rule installation requests when the timeout proxy was in use. Test streams with rates above 10, 000 packets per second were a significant strain on several components of our testbed, even with the timing proxy, including both the switchs and controllers

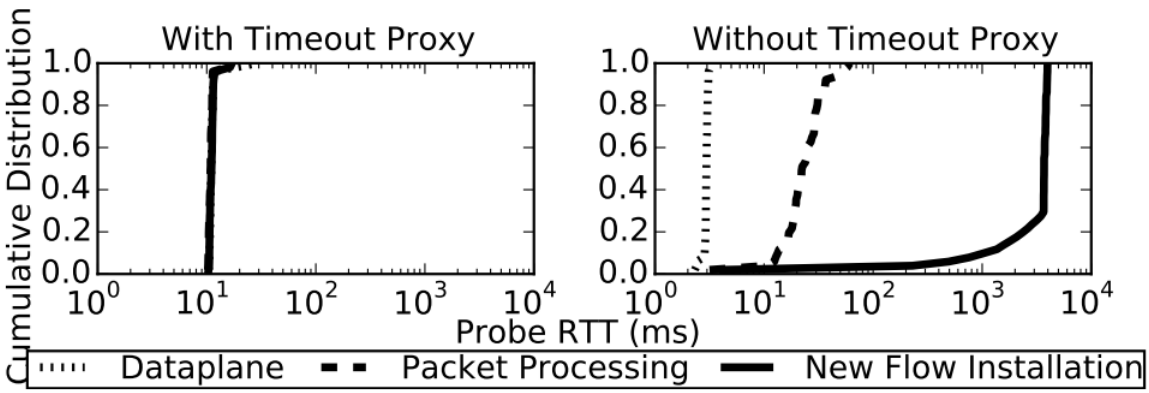


Figure 3.18.2: Probe RTT distributions for test streams that invoked the control plane.

CPU. Two simple improvements would provide protection against attacks with much higher rates: moving the timeout proxy application off the switch, to a dedicated server with a stronger CPU; replacing Ryu, the Python based control platform we used in our testbed, with a control platform designed for performance, such as [28, 16, 34].

3.5 Conclusion

Control plane timing attacks allow adversaries to learn about Software-defined networks without needing to compromise their infrastructures. We developed a more refined timing attack that reveals new kinds of sensitive information about an OpenFlow network and demonstrated that it has high accuracy against real OpenFlow hardware. We also proposed and evaluated a robust software based defense, capable of running on existing physical OpenFlow switches. As SDN adoption grows and networks deploy increasingly more advanced control plane applications, it is likely that timing attacks will be observed in the wild. Our work provides a comprehensive first look at their potential, and offers a deployable and effective defense.

Chapter 4

Network Function Virtualization

Network function virtualization (NFV) [42] forms the core of next generation networks. A lot of research has gone into designing highly scalable NFV architectures using software. The designs described in "Related Work" chapter [33] makes use of disaggregation to resign network functions (NFs). The core idea is the decoupling of state from network function itself. The makes it much easier to manage state independent of NFs and we also achieve highly resilient networks. However, this disaggregation of state means every network function now has to make a remote query if it needs to access its dynamic state. The remote query is very expensive and unless we use specialized hardware, the latency becomes too much.

In this chapter, we have tried to answer the question - Is it possible to maintain disaggregation in NFVs while bringing the remote data even closer to network functions? This may allow is to eliminate the overheads of remote latencies. The idea is to setup data placement in such a way that >90% of data can be accessed locally. As shown in the Figure 4.1.0 the resulting architecture will look something of in midway between traditional and completely disaggregated architectures. Having the data locally means much lower latencies in processing without relying on specialized hardware like infiniband or RDMA.

One solution for this would be to use caching of remote data. However, caching brings unnecessary complications into the picture, not to mention issues of data coherency. This is why we do no want to rely on caching techniques. The other approach would be co-locating the data-store itself alongside the Network Functions and distribute the state among these co-located instances.

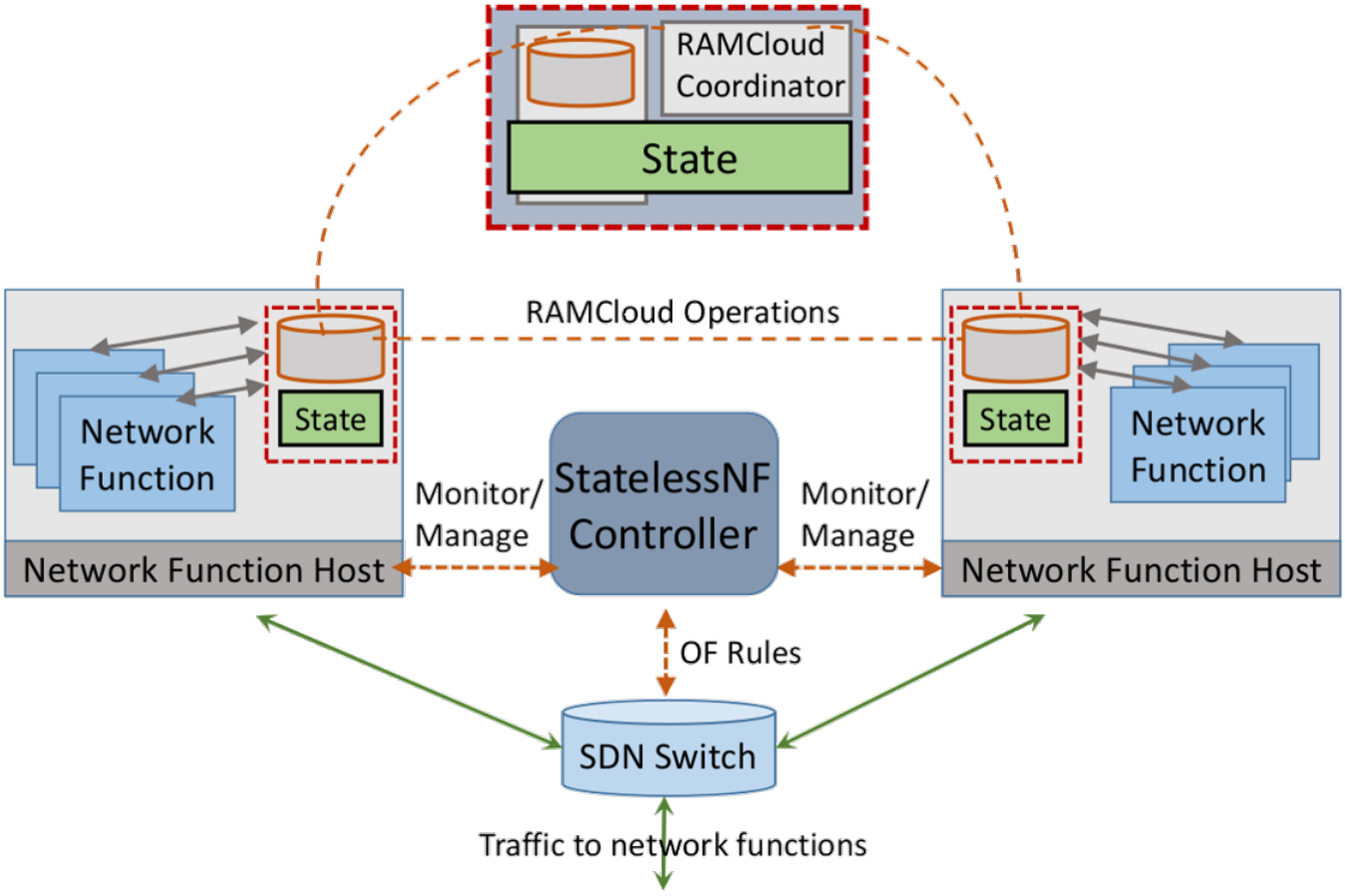


Figure 4.1.0: A representation of ideal NFV architecture

This approach will work, provided we figure out a way to optimize data placement. In short, there are two issues that are needed to be addressed in this scenario:

(1) How do we optimize the Data Placement?

(2) How do we make sure that any Network Function can access data at any location while still maintaining the logical disaggregation of data from the NFs themselves?

Another objective of our research has to do with key expires. In stateless NF, a central orchestration component is needed to initiate a timer for each data element that is set to be expired. It then picks one of the active NF instances and asks it to delete the key. This approach to key deletion is not very scalable. Some databases support in-built key expiry mechanisms. These come at the cost of additional complexity inside the data-stores themselves as they need to run periodic cleanup operations. These cleanup operations could lead to little periodic spikes in latency. As long as these spikes are not significant, it makes more sense to let the data-store itself handle the key expires.

Nature of Data: The state data is usually stored in a key-value pair format for the purposes of easy manageability. The key is a 5-tuple which identifies a specific flow. It is extracted from the packet headers. The value is network function dependent. For example, a load-balancer might need to store need to store backend server IP corresponding to every flow. Or an intrusion detection system might need to store automata state in the value.

The most obvious method of distributed data placement is "hash based partitioning". However, we cannot simply use it in our design because of the reasons explained in the next section.

4.1 Challenge : Data Placement

Our design aims towards keeping state data as close to the network functions as possible, while still maintaining logical dissociation of state from the network functions. Most of the distributed systems today use some kind of hash based sharding [32] of data (as shown in Figure 4.2.0) in order to distribute the data among several nodes. For this purpose a hash based algorithm is selected and

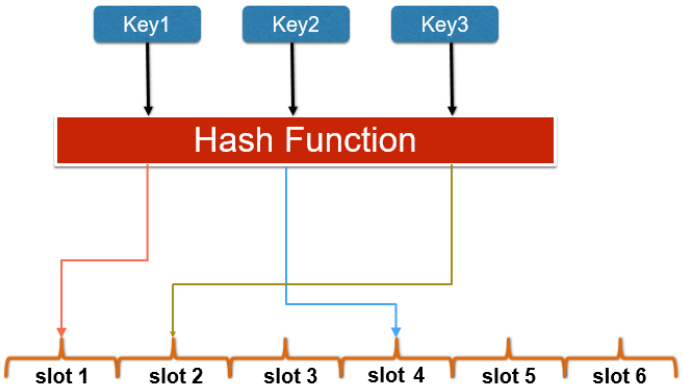


Figure 4.2.0: Traditional hash-based approach to data sharding

the hash space is partitioned into different ranges. Each range acts like a slot and slots are assigned to different nodes based on some predetermined method. When we want to insert a key-value pair into this database, the database uses the key to calculate the hash and the corresponding key-value pair is stored into the node which is assigned that particular slot. While this algorithm works really well in distributing load over multiple nodes, there is no way to optimize the data placement. This is because hash functions are designed to be non-reversible we do not have any control over the keys.

4.2 Directory Data Store

Our proposed solution to the data placement problem is using a directory based data storage. Basically, instead of distributing the state data, we instead maintain a distributed directory store to keep track of the state data. The directory store holds pointers to the actual data locations. Pointers can be simply a socket of the data-store where the data is stored. The data is kept locally (separate from NFs) to speedup access and the directory store is basically a hash based distributed store which provides remote lookup capabilities. This is show in the Figure. The NF operations in this scenario work as follows:

(1) **Read Operation:** NF first checks local data-store and if it cannot find the data, it checks if a directory entry for the given key exists. If it does, then it uses the directory pointer

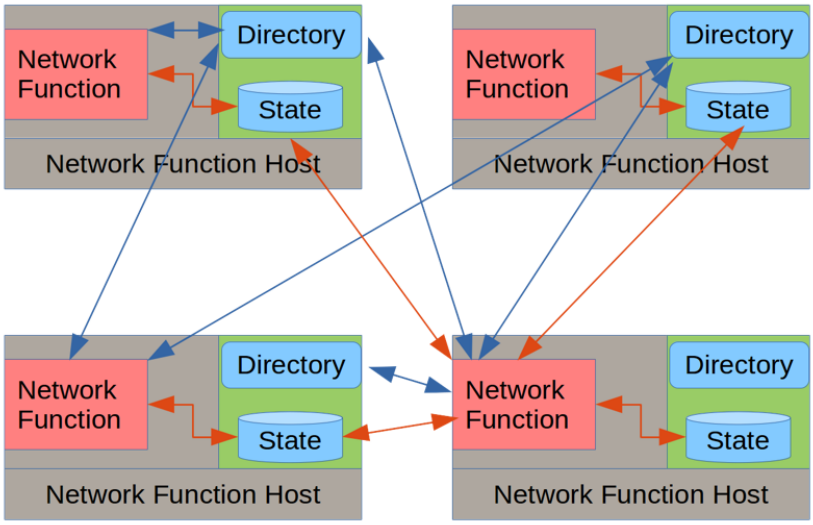


Figure 4.3.0: A logical representation of directory based data-store

to get access to the data. This means in the worst case scenario, we might end up making two accesses for one read operation. We get around this limitation by implementing a LRU cache inside every Network Function. Anytime a NF makes a request for a directory entry, it caches the entry in its local cache so that next time it doesn't have to make an extra remote read. Since this cache is LRU based, we do not need to deal with cache management. Since we do not support key migration yet, there are no coherency issues with the directory entries.

(2) **Write Operation:** NFs write any new entry locally and add the corresponding directory entry in the directory store. The second operation is done asynchronously so it doesn't impact the latency in any way.

(3) **Update/modify Operation:** If a NF is not writing a new data entry, then it needs to update the existing local or remote data entry.

This architecture brings with it the advantages to low latency local read/writes while still benefiting from disaggregation.

4.2.1 Design Considerations

Consistency and Coherency. Coherency means all the network functions which are using the directory store have the same view of the data. Since we decided not to use data caching, we do not face this issue in our design. Directory caches, however could lead to coherency issues in case there is migration of state across nodes. Another problem is consistent writes. There is no guarantee that if a NF has read some data and done some processing on it, the data on the data-store will remain the same during this operation. This could happen, for example in an Intrusion Detection System where the state needs to be modified for every packet. To ensure consistency in these scenarios, we implemented an atomic "Test and Set" instruction inside the data-store. This adds some latency overhead but ensures that the state data is updated correctly. Another factor which brings up consistency issues is data-store replication. We might get some performance improvements with replication, but the effects of potential loss of consistency are data-store dependent. We plan to analyze these in future.

State Migration. Migration of remote data to the local instance might bring huge speedups in performance but it adds a lot of complexities. If a flow spans multiple NF instances, then it does not help to migrate state. Also, the data migration has to be an atomic operation and all the network functions should be made aware of it somehow, else they might not find the state and assume that the flow and state has expired. Because of these reasons, we have decided not to implement migration in our initial system. One scenario where migration would be much more simpler and make sense if in a system which has a constraint that at ant point in time, all the packets belonging to a single flow will only be processed by a single NF.

4.3 Evaluation

We were able to implement a firewall with directory access client built in. We tested it over 10 Gigabit network interface. We used 6/12 cores DELL servers running Intel's Dataplane Development Kit (DPDK)[14] to run all the applications. Netmap traffic generator was used to act

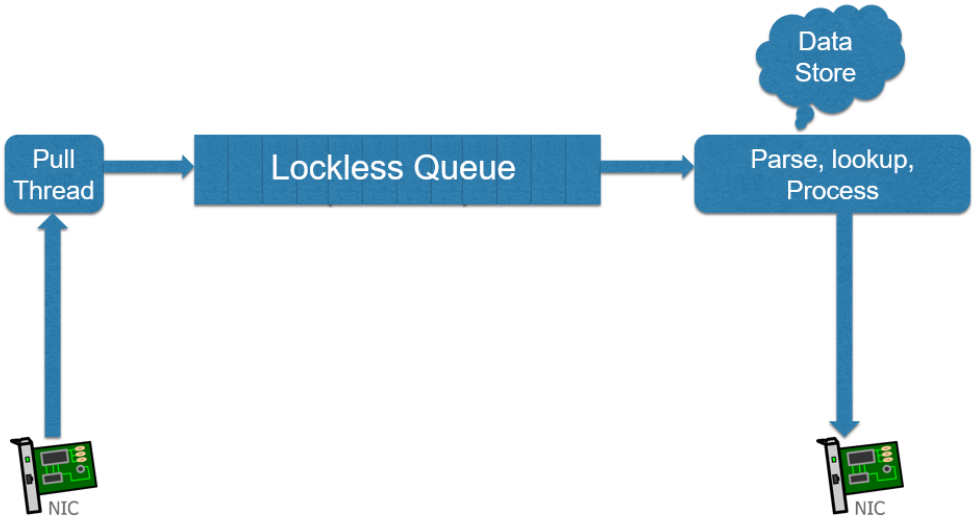


Figure 4.4.0: Firewall architecture

as a packet source/sink. The 'Ryu' [4] SDN controller was used to control and direct traffic.

4.3.1 Firewall Architecture

Our firewall had four main components as shown in Figure -

(1) **Producer Thread:** This is a userspace thread which runs network card driver and polls the Rx network interface for any received packets. After it receives packet(s), it simply enqueues them into one or more lockless queues.

(2) **Lockless Queue:** A high performance queue which holds the packets before it is processed.

(3) **Process and Dispatch Thread:** This thread dequeues the packets from the queue, extracts the 5-tuple from them and retrieves the corresponding state from the data-store. It then processes the packet based upon the state and then it either forwards to the Tx interface or drops it.

(4) **Data-Store:** Finally, the database which is used as the primary data-store for all the state data. We chose 'redis'[37] database for our application because of many benefits it

provided:

(a) In-memory storage

(b) Open Source and can be modified

(c) Support for shared memory module for super low latencies

(d) Server side scripting which allows implementing custom commands.

(e) Replication and cluster support

However, the remote reads on redis are purely on TCP which is one disadvantage.
Also, to make a fair comparison to StatelessNF, we added two more queues and consumer threads over same pair of network interfaces. Every thread was running on a physical core because hyper-threading has a bad impact on thread performance.

4.3.2 Latency Measurements

First we ran tests to measure how much latency gain we can achieve by running the data-store locally vs remotely. Also, we wanted to compute latency variation as a function of percentage co-location. The StatelessNF [33] was able to achieve 6 microseconds read and 16 microseconds write latencies with a remote data-store running over RDMA. Their choice of data-store was RAMCloud [36] because of its RDMA support. But, since we're using redis, we will compare redis latencies. In the initial testing over TCP with localhost or unix sockets, it achieved around 9-14 microseconds read/write latencies. Localhost access added 2 microseconds overhead on average over unix socket access. This is decent local access latency but we could do better. We found a shared memory module for redis which brought down access latency down to 2 microseconds for both read and write operations. Then we tested purely remote read/write operations over a 1 Gigabit interface. We got around 80 microseconds remote read/write latency. With the help of query pipelining, we were able to do 100 remote queries in range of 100-200us. The Figure 4.5.2 shows a graph of the latency improvement factor as compared to percent co-location in our design. The latency improvement

factor is defined as the ratio of remote to local access latencies. We see that as more data becomes local, we gain a significant improvement in the latency.

4.3.3 Throughput

Our netmap traffic generator is able to generate a maximum of 14 million packets per second (Mpps) over a 10 Gigabit network interface. So, we tested how many packets per second our design is able to handle. If the incoming packet rate is higher than the processing capabilities of NFs then they start dropping excess packets. For this test, we tried to simulate the worst case scenario. Which means we generated minimum size packets (64 bytes) and forced one data-store operation per packet. We then measured and compared the rate at which packets were received at the sink to the rate of packet generation at source. We then plotted this data versus the percentage co-location of data as shown in Figure 4.6.2. As seen from the figure, the total throughput is proportional to co-location percentage. At 100 percent co-location, all the reads were local and we were able to achieve 4-4.8 Mpps processing rate. As this percentage decreased, the total throughput also decreased as we expected. At 0 percent co-location, all of our requests goto remote data-stores via TCP and we get the worst performance at around 1.3 Mpps.
To conduct these experiments, we optimized our data-store operations using combination of following techniques -

- Multi-get Operations : This operation allows us to fetch multiple values in a single command.
- Pipelining with Batching : We can queue multiple commands without blocking on the connection and then retrieve a bulk reply from the data-store. We used a batch size of 128 for our experiments as that provided the best balance between latency and throughput.

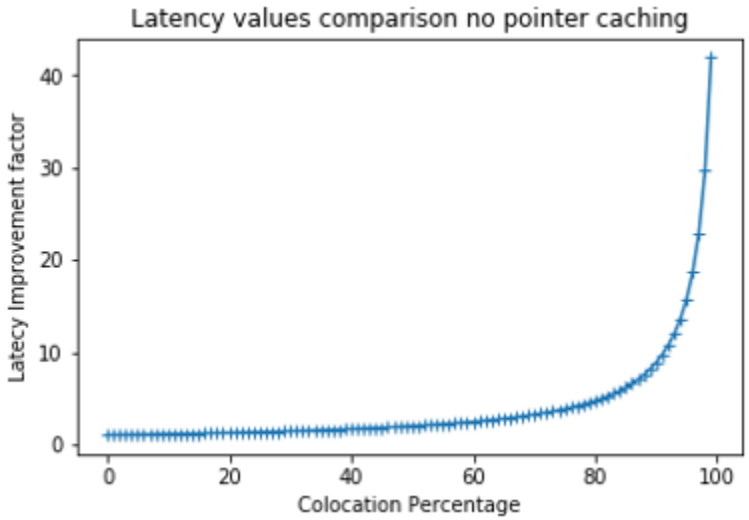


Figure 4.5.2: Latency Factor improvement versus percent co-location

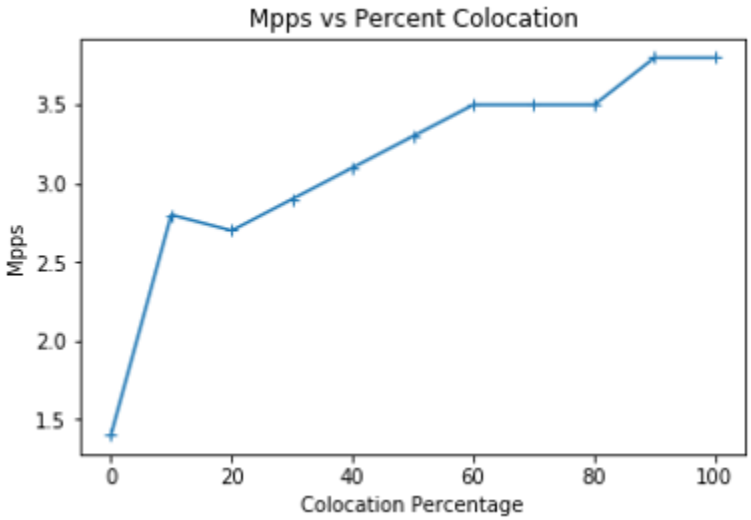


Figure 4.6.2: Network Function throughput versus percent co-location

4.4 Conclusion and Future Work

This research is still a work in progress. In conclusion, we were able to match the performance of an RDMA based disaggregated data-store without making sacrifices on latency.

- We were able to come up with a co-location algorithm that maintained logical disaggregation of state data, while bringing state closer to NF instances.
- Our data-store took care of key expires with small overhead in latency.

In future, we plan to explore the following avenues to improve upon our initial design-

- Replication? We need to study if replication can improve our performance and if it affects the system consistency in any way.
- State Migration? We need to study the possibility and effects of state migration in our system.
- We need to quantify LRU cache effects on the directory lookup.
- Possible use of DPDK to improve remote access.

Chapter 5

Research Summary and Conclusions

In this research, we studied the effects of dissociation in two network architectures, namely 'Software Defined Networks' and 'Network Function Virtualization'. Initially we analyzed how disaggregation leads to a more flexible and resilient design but introduces potential flaws at the interfaces. One such flaw is introduction of timing side-channel in Software Defined Networks. Another flaw was introduction of latency at the interfaces in Network Functions.

First Chapter talks about basics of disaggregation and we introduce Software Defined Networks and Network Function Virtualization. We also describe briefly each of these systems.

Second Chapter describes some of the related work happening in research community with similar themes. This also highlights that while, people have been developing and studying these advanced network systems for years, not much research has gone into studying the effects of disaggregation and providing and evaluating solutions for problems.

In the third chapter, we go into details of why the timing channel exists. It is because of flow processing happening at the control plane. The control plane load introduces significant skew in timing distribution of probe packets. Using a specifically crafted packet test stream, we could verify whether it was being processed at the control plane or not. If the test stream has a rule installed in switch, it will not goto control plane, else it gets processed by the control plane. This test method could be used to extract information such as host-communication patterns and rules, firewall acl rules or presence of a monitor in the network. We evaluated this attack with real hardware and realistic network conditions like background traffic. We were able to find bottlenecks in control

plane that make such attacks so easy. We then proposed a solution to this attack - a timeout proxy in control plane. This proxy sits between the switch and the controller. It's aim is to normalize the control plane timing characteristics by sending a reply to the switch if the SDN controller did not reply in time. We evaluated this proxy on a real switch and were able to see that it becomes nearly impossible to carry out the timing attack with our proxy in the control plane. The effects of the proxy were such that it did not affect normal functionality of the switch.

In chapter four, we shift our focus to Network Functions. StatelessNF[33] proposed decoupling of network state and network functions to achieve a highly scalable and resilient network design. They leveraged recent growth of low latency technologies like RDMA to move the state into a remote data-store while, while still maintaining reasonable latencies. Although, remote read latency can be brought down to reasonable numbers with specialized interfaces, we try to rethink the idea of decoupling. We think that we can further improve this design if we brought the state data closer to the network functions. We propose a 'Directory Data Store' which aims towards keeping as much data local as possible, while still maintaining decoupling. For this purpose, we define the term co-location as the percentage of data (and queries thus) which are local to a NF instance. We implemented a basic directory store with a Firewall and evaluated latency and throughput. We were able to match the throughput of an RDMA based system with 100 percent co-location and got 40x improvement over remote TCP remote store. This is still a work in progress but we're seeing encouraging results in our initial testing.

Bibliography

[1] National collegiate cyber defense competition. http://www.nationalccdc.org.

[2] Nsa uses openflow for tracking... its network. http://www.networkworld.com/article/2937787/sdn/nsa-uses-openflow-for-tracking-its-network.html.

[3] Protected repository for the defense of infrastructure against cyber threats. http://predict.org.

[4] Ryu. http://osrg.github.io/ryu/.

[5] Ryu 1.0 documentation: Router. https://osrg.github.io/ryu-book/en/html/rest_router.html.

[6] Pica 8. Pica 8 3290 datasheet. http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3290-p3295.pdf.

[7] Alexey Andreyev. Introducing data center fabric, the next-generation facebook data center network. https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, 2014.

[8] Michiel Appelman. Performance analysis of openflow hardware. 2012.

[9] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In Proceedings of the 16th international conference on World Wide Web, pages 621–628. ACM, 2007.

[10] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Computer Security–ESORICS 2011, pages 355–371. Springer, 2011.

[11] David Brumley and Dan Boneh. Remote timing attacks are practical. Computer Networks, 48(5):701–716, 2005.

[12] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. ACM SIGCOMM Computer Communication Review, 37(4):1–12, 2007.

[13] Cisco. Configuring port security. http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst6500/ios/12-2SX/configuration/guide/book/port_sec.pdf.

[14] Intel Corp. http://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html.

[15] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: scaling flow management for high-performance networks. In ACM SIGCOMM Computer Communication Review, volume 41, pages 254–265. ACM, 2011.

[16] David Erickson. The beacon openflow controller. In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, pages 13–18. ACM, 2013.

[17] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In ACM SIGPLAN Notices, volume 46, pages 279–291. ACM, 2011.

[18] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. Sdx: A software defined internet exchange. In Proceedings of the 2014 ACM conference on SIGCOMM, pages 551–562. ACM, 2014.

[19] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In NSDI, volume 10, pages 249–264, 2010.

[20] Will Hurd. The data breach you haven't heard about. http://www.wsj.com/articles/the-data-breach-you-havent-heard-about-1453853742, 2016.

[21] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM, pages 3–14. ACM, 2013.

[22] Keith Jarvis, Jason Milletary, and Threat Intelligence. Inside a targeted point-of-sale data breach, 2014.

[23] Marc Joye, Pascal Paillier, and Berry Schoenmakers. On second-order differential power analysis. In Cryptographic Hardware and Embedded Systems–CHES 2005, pages 293–308. Springer, 2005.

[24] Rowan Kloti, Vasileios Kotronis, and Paul Smith. Openflow: A security analysis. In Network Protocols (ICNP), 2013 21st IEEE International Conference on, pages 1–6. IEEE, 2013.

[25] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Advances in Cryptology—CRYPTO'99, pages 388–397. Springer, 1999.

[26] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Advances in Cryptology—CRYPTO'96, pages 104–113. Springer, 1996.

[27] Tadayoshi Kohno, Andre Broido, and KC Claffy. Remote physical device fingerprinting. Computing, 2:2, 2005.

[28] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In OSDI, volume 10, pages 1–6, 2010.

[29] 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, page 19. ACM, 2010.

[30] Junyuan Leng, Yadong Zhou, Junjie Zhang, and Chengchen Hu. An inference attack model for flow table capacity and usage: Exploiting the vulnerability of flow table overflow in software-defined network. arXiv preprint arXiv:1504.03095, 2015.

[31] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2):69–74, 2008.

[32] MongoDB. Hashed sharding. `https://docs.mongodb.com/manual/core/hashed-sharding/`.

[33] Azzam Alsudais Murad Kablan and Eric Keller. Stateless network functions: Breaking the tight coupling of state and processing. NSDI, 2016.

[34] Big Switch Networks. Project floodlight. `http://www.projectfloodlight.org/floodlight/`.

[35] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pages 117–130, 2015.

[36] RamcloudWiki. RAMCloud. `https://ramcloud.atlassian.net/wiki/display/RAM/RAMCloud`.

[37] Redis. Redis – an opensource in-memory datastore. `https://redis.io/`.

[38] Seungwon Shin and Guofei Gu. Attacking software-defined networks: A first feasibility study. In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, pages 165–166. ACM, 2013.

[39] Seungwon Shin, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, Guofei Gu, and Mabry Tyson. Fresco: Modular composable security services for software-defined networks. In NDSS, 2013.

[40] Dan Williams Shriram Rajagopalan and Hani Jamjoom. Pico replication: A high availability framework for middleboxes. ACM, 2013.

[41] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Enabling practical software-defined networking security applications with ofx. In Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS), 2016.

[42] Wikipedia. Network Function Virtualization — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Network_function_virtualization`.

[43] Wikipedia. Student's t-test — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Student's%20t-test&oldid=723048300`.

[44] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In Proceedings of the 2012 ACM conference on Computer and communications security, pages 305–316. ACM, 2012.