Network Function Capacity Reconnaissance

Aidan Walsh

Advisors: Mary Apostolaki and Aqsa Kashaf (CMU)

Independent Work Report and Documentation, Spring 2024

Abstract

Network Function (NF) capacity reconnaissance (NFCR) serves to estimate the processing speed of devices that monitor packets entering and exiting the network. This reconnaissance can be helpful for adversaries attempting to attack a network and network operators attempting to configure a network. There exists several methods that attempt to predict NF processing capacity, such as Self Loading Periodic Streams (SLoPS) implemented by Dovrolis et al. [3] , but most methods are detectable, slow, and yield low accuracies. My independent work attempts to understand a new method, NF capaciTY estimation (NFTY) that uses packet dispersion values and binary segmentation to produce improved accuracies of NF capacity estimation and is faster and more undetectable compared to contemporary methods. I test NFTY when packets are sent across the Internet and compare its effectiveness to SLoPS.

1. Introduction

Network Function Capacity Reconnaissance (NFCR) can be helpful for both an adversary and network operator. Consider the case where an attacker is attempting to conduct a DDoS attack on a network. As an attacker, it is in their interest to understand the capabilities and limitations of a network involving processing capacity (in packets per second or bytes per second). For example, uncovering that a server can handle at most 450,000 packets per second can prove as useful information because if an attacker can only send at most 250,000 packets per second, then they will learn that their attack will prove to be almost fruitless. As a result,

NFCR can help an adversary plan a DDoS attack and will be more effective if their NFCR is undetectable. NFCR is also helpful for network operators because it can reveal to them the capacity of servers in their network. For example, if there are two servers in a network, and the network operator uses NFCR to reveal that one server is faster than the other, then they may route more traffic to the faster of the two servers. Thus, NFCR can be used to optimize routing paths to minimize round trip times (RTTs).

NFCR is already being used and several different methods have been proposed that involve the following: iterative techniques[3], flooding[1], and dispersion[2]. These methods, however, involve a significant amount of overhead: they are slow, inaccurate, undetectable, and send many packets. As a result, we propose NFTY which aims to minimize these issues by modifying contemporary methods to produce results that improve accuracy, increase efficiency, and reduce detectability.

The method Network Function capaciTY estimation (NFTY) was originally proposed by Maria Apostolaki and Aqsa Kashaf and my contribution to their paper is confirmation of their results, as well as experiments in the Internet environment. The end of this paper has the official paper attached at the end. My contribution to the official paper is section 6.4 (Internet Experiments). In this paper after section 2, the rest of this report mostly deals with the methodology and produced results so that future research may reference this paper as a guide to continuing experimentation on NFTY or as a general guide for network and firewall configuration. Frequently throughout this report, I will be referencing my Github which has all the required code and files to run the experiments and setup the network environments. To properly replicate my work, please clone my github repository found at https://github.com/Aidan-Walsh/NFTY-Internet/tree/main .

2. Related Work

Other research focuses on network reconnaissance that infers the topology of the network and rules governing firewalls. For example, Samak et al. [6] infers rules of a remote firewall by sending probes. Rather than inferring the packet processing rate like this research, they infer the rules. Their work finds three techniques that an attacker can use to reconstruct a firewall's policy by probing the firewall with specially tailored packets and can do so in "feasible time with acceptable accuracy". Ramamurthy et al. [4] estimates other values such as bandwidth, CPU utilization, and memory usage of a server. Salehin et al. [5] estimates the delay in the networking stack of a server. Their estimation is based on measuring the capacity of the link connected to the host under test. Thus, NFTY continues this area of research that attempts to infer a capability of a NF.

Research also contains methods alternative to NFTY: Self Loading Periodic Streams (SLoPS) [3] and other iterative based methods. This paper will focus on SLoPS which uses one-way delay to iteratively probe and send bursts of packets to determine the processing capacity. This paper will also focus on another iterative method (this paper will call it Iterative Drops) which sends a burst of packets and if any packets are dropped, then it will send at a lower rate, otherwise, it will send at a higher rate. This is done using a binary search approach that probes until a certain limit has been reached. SLoPS and Iterative Drops will be further discussed in section 3.2.

3. Background

For all instructions given hereafter, all files should be inspected and changed by an individual attempting to reconstruct these experiments or further them. All changes that will need to be made involve: changing constants in the files that define the IP addresses, MAC addresses, interfaces names, passwords of the nodes, and paths used. This is because every time an experiment is begun, all of these values change.

3.0 Remote Servers and Two-Way Model

All of these experiments that tested for packet processing capacity were done on remote servers. **These servers were established either on Cloudlab Wisconsin c220g2 nodes or a Vultr Los-Angeles node. The Vultr Los-Angeles node was given the following configuration: Bare Metal, Los-Angeles, Ubuntu 23.10, Intel E-2288G, RAID 1.**

To perform these experiments, there are two kinds of models that can be used: a two-way model and a one way model. The ultimate goal is to detect the processing capacity of Network Functions, and in the two-way model, we use three devices to do so. A diagram of the topology that I used for my results is given below.

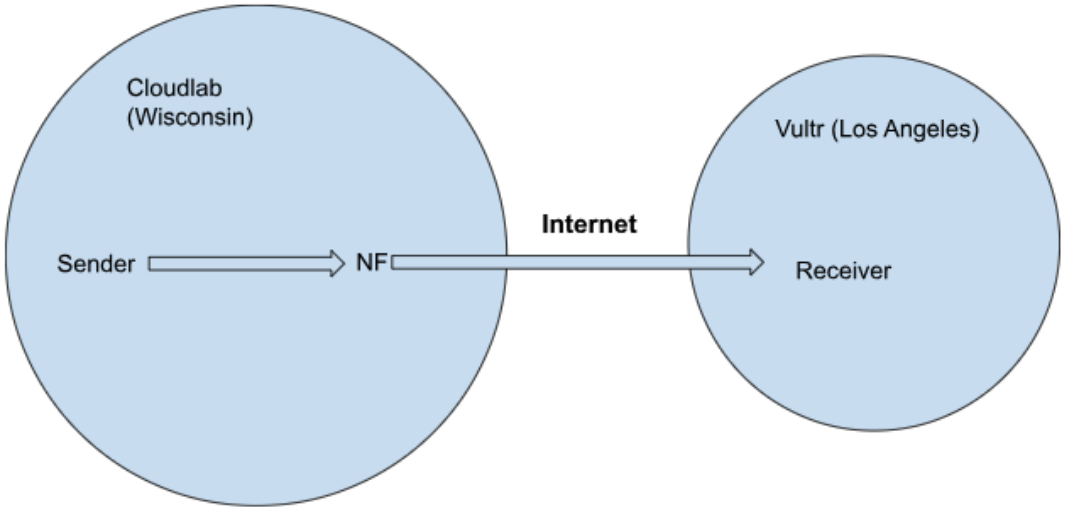


Figure 0: Topology of Experiments that send packets across Internet

We see that we have a Sender machine and NF that are established in the Cloudlab-Wisconsin cluster, and a Receiver machine is established in Vultr Los Angeles. The sender sends packets to the receiver, but they are all seen and processed by the NF. After processing, the packets are then sent across the Internet and so could experience many hops.

NFTY utilizes packet arrival times to predict the processing capacity of the NF. As a result, when they arrive at the receiver, there could be lots of noise in their arrival times due to the Internet. In the two way model, the network operators or adversaries have access to both the sender and receiver to attempt to infer the processing capacity of the NF. On the other hand, in the one way model, the network operator/adversary only has access to the sender. This latter model is a more realistic model for the adversarial setting. This paper only utilizes the two-way model. As a result, we can SSH into both the sender and receiver to conduct our tests.

When establishing the Cloudlab nodes, we must provide a source file that describes the topology and gives the nodes the required networking interfaces. For the topology in figure 0, use the file Internet.py as the profile for the Cloudlab experiment.

3.1 Limitations to NFCR

3.10 DVFS

Dynamic Voltage Frequency Scaling (DVFS) is where the frequency of a CPU does not remain constant. That is, as load on a CPU increases, so does its frequency. For example, a CPU will operate at a reduced frequency at the beginning of a workload, and as the voltage increases from the workload, so will the CPU frequency. It is dangerous for a CPU to handle a small workload (requiring a low voltage) with a high frequency for an extended period of time. It is also energy inefficient for a CPU to handle a heavy workload (requiring a high voltage) with a low frequency. Since DVFS is present on modern systems, this means that the processing capacity of a NF will change with workload. NFTY contains an optimization that mitigates any errors caused by CPU fluctuations, but for my experiments, I deactivated DVFS on both the sender, receiver, and NF. Since the sender had DVFS deactivated, this means that it can send packets at a high rate, and since the NF had DVFS deactivated, this means that it will immediately process packets at its max capacity. The purpose of this is to see the viability of

NFTY when packets are sent across the internet. If the results would have shown NFTY as ineffective, then it would be ineffective with an NF with DVFS activated. To deactivate DVFS, run cpu.sh. Also run the command “ifconfig” on all devices to view all interfaces. For all interfaces, we want the packet receiving and sending delays to be 0, so run the commands “sudo ethtool -C <interface> rx-usecs 0” and “sudo ethtool -C <interface> tx-usecs 0”. We also want to maximize the size of the buffers. To do this, run “sudo ethtool -g <interface>” to view the max tx and rx values for the system. Then run “sudo ethtool -G <interface> tx <max_value>” and “sudo ethtool -G <interface> rx <max_value>” to make the adjustment. The nodes are now all ready for the experiments.

3.11 Multithreading

Although DVFS is deactivated, multithreading is still possible if software allocates multiple CPUs to a process under the hood. Multithreading may not be detected by some probing techniques since multithreading can be triggered by load or other circumstances that probing techniques may not bring about. For example, an NF may be multithreaded and assign packets to threads based on flow. Thus, if an adversary were sending packets to an NF from only one IP that assigned packets to queues based on flow, then all the packets would only go to one thread. As a result, the adversary would only be probing one thread, not all. Our experiments utilize multithreaded-ness and later sections describe how we overcome this.

3.2 Firewalls and NFs

Network Functions can be defined as devices that monitor packets entering and exiting a network. A simple example of one would be a firewall. This means that packets must be processed in some way when they enter the NF. A simple router that forwards packets somewhere that does no processing would not fit this criteria. To estimate the processing capacity of a NF, we used five NFs for our experiments and tested the performance of various

NFCR methods. These five NFs were run on servers that used Ubuntu 18.04 and are called SNORT-RL, SUR-BL, SUR-RL, SUR-BL-mt, and SUR-RL-mt. They are given their names from the type of software they were using for packet processing, their rules files, and whether or not they were multi-threaded. “SNORT” means that the NF was running snort software, “SUR” means that the NF was running Suricata software, “RL” means that a rate-limiting firewall was used (suricata-rl.rules), and “BL” means that a blacklist firewall was used. The rules file is given in blacklist-suricata.rules in the repository (located in the Suricata directory). This rules file consists of a large set of IP rules. Finally, the “mt” means that the NFs were multi-threaded (only run with 2 threads). The absence of “mt” means that the NF was single-threaded.

The github comes with Suricata (version 6.0.2) on the NF, so run suricata.sh in the Github repository to properly configure. You may also run into the issue where Suricata cannot be properly installed with a later version of rust. To fix this issue, you must downgrade to rust 1.52.0 using “rustup default 1.52.0”. If this does not immediately work, and you get “rustup: command not found”, perform the command “curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh” and follow the on-screen instructions. Finally, run suricata2.sh to configure all IP settings. Ensure that the last command in suricata2.sh is executed. To check that it is, run “sudo iptables -L”. If no rules are listed, run the command “sudo iptables -I FORWARD -i <rcv_interface> -o <out_interface> -j NFQUEUE” where you give the interfaces that receive and output packets for the NF. This will forward all packets to NFQUEUE where packets will be processed by Suricata (and Snort too). Make sure that the rcv_interface is not the receive interface for all packets to the NF. If that is the case, then all the SSH packets that are letting you communicate with the NF will be sent to NFQUEUE and you will no longer be able to SSH. To bypass this, add “-d <ip_of_receiver>” to the iptables command. To run Suricata with a certain NF, just run run_<NF>.sh. Also, note that the directories given in the config files will be incorrect since your machine will have different paths. Be sure to edit the paths in all .yaml files so that the paths are correct.

When sending packets through Suricata, be sure to use sendpfast() in python. I initially used hping3, and Suricata dropped all TCP packets. Even after changing a configuration setting in the .yaml file that prevents the dropping of malformed packets, hping3's TCP packets were still dropped.

We also experiment on Suricata's multithreaded NFs by changing configurations in the .yaml file. We only experiment with 2 threads. Suricata assigns packets to threads (where each thread has a queue that processes packets) based on flow. Section 4.0 describes how NFTY is able to account for this and detect the processing rate for all threads. To properly configure the NF to be able to handle multithreaded processes for Suricata, run prepare_mt.sh. Note that once you run this, if you want to switch back to single threaded, run prepare_st.sh. These files allocate packets to the proper queues depending on the threading. Be sure to add an extra option to the bash code in these files if you do not mean to be sending all packets on the receive interface to NFQUEUE.

Snort will also have been downloaded by cloning the repository located in snort_src. To configure Snort, run snort.sh. Note that you may have to edit snort.sh to fit your directory structure. To run the NF with the rate-limiting rules file, just run run_snort.sh. Note that Snort is also running with NFQUEUE, not Afpacket. This is because with Afpacket, the receiving interface and sending interface will be bridged, which will cause strange behavior when the NF forwards packets. Packets will seem as if they are duplicated, but packets are actually likely getting sent back and forth. Also, with NFQUEUE, the buffer will not be large enough for Snort. To increase the size of the queue to prevent packet drops, navigate to this website (https://blog.inliniac.net/2008/01/23/improving-snort_inlines-nfq-performance/) and run the commands.

As Snort ran in NFQUEUE mode, I noticed functionality that differed from Suricata. As Snort ran for an extended period of time, its processing capacity changed. So, after sending many bursts of packets through the Snort NF, its processing rate changed. Since my

experiments only produced one ground truth for each firewall (I could have used multiple for Snort, but this would have overcomplicated things), I overcame this by restarting Snort every packet burst. This affects the testing that is described throughout the rest of this paper. Furthermore, Snort must be run using the “taskset” command, which designates a CPU to the process. If this is not done, then multiple CPUs will be allocated to the Snort process, thus causing DVFS-like results. Refer to the figure below to see how not performing “taskset” can affect the dispersion times of packets when processed at the NF.

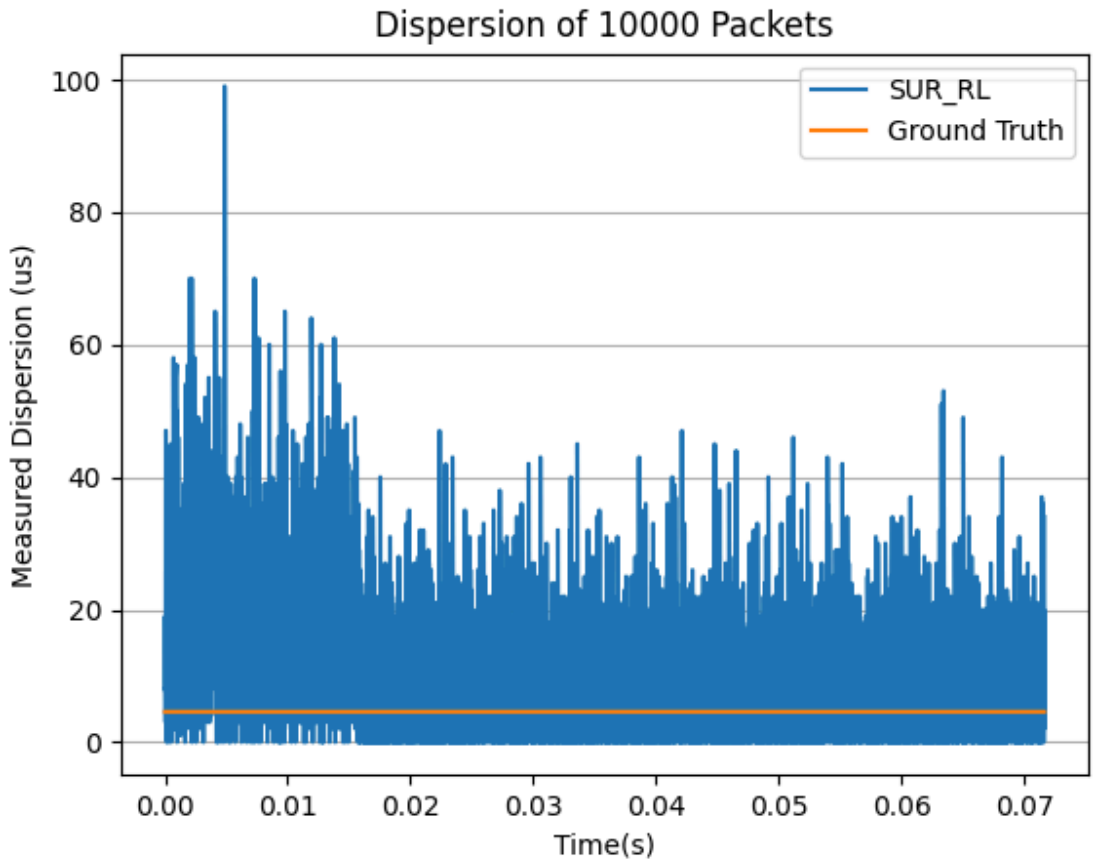


Figure 1: At approximately time t = 0.015, the dispersion values drop due to another CPU being allocated to the process that is running SUR-RL

3.2 Competing Methods

All methods to detect the NF's processing capacity were either run locally or on the sender (client) machine. To have the sender machine properly configured, run the file packages.sh. All python files run on the client must be run using Python 3.9 due to package inconsistencies in earlier versions and some later versions. After trial and error, I found Python 3.9 to be the best.

3.20 Iterative Drops

This method uses a binary search to find the packet processing rate of the NF. So it will have a minimum (0 pps) and maximum bound (500,000 pps), and will begin by sending packets at 250,000 packets per second for a second. If packets are dropped at the NF, then it sent too fast and will send at a slower rate. Otherwise, it will then send at a faster rate. This process is repeated until the binary search approaches a limit of a higher bound - lower bound < 100. If the detected rate is the maximum bound, then this is repeated with a minimum bound of 500,000 pps and a maximum bound of 1,000,000 pps. This method is just one method of many that use an iterative approach to uncover the packet processing rate of NFs. This method is very accurate, and can serve as the ground truth for our experiments, but is impractical as a method for network operators and adversaries for the following reasons: it takes a long time to run and it leads to packet drops which makes it detectable and diminishes the NF's performance. To run this method, run benchmark_<NF_name>.sh located in the NFTY/setups directory. Note that you will have to change the values of constants in internet3_benchmark.py (and any other benchmark python files) so that your experiment may run properly.

3.21 SLoPS

Self-loading periodic streams (SLoPS) is a method implemented in a measurement tool called Pathload that uses one-way delays to predict the processing capacity of an NF. Like Iterative Drops, it also uses an iterative approach, where it initially sends at a rate 250,000 pps and calculates two values Spdt and Spct. If these values satisfy certain conditions, then the trend is said to be “increasing”. If this is the case then they sent packets too fast and so an upper bound of 250,000 is set and the rate is halved. If the trend is “decreasing” then they sent packets at too slow of a rate and so a minimum bound of 250,000 pps is set and the next rate is set to 375,000. One nuance is that, unlike Iterative drops that actually tried to get drops to occur (by sending at a fast rate for a full second), SLoPS sends a smaller amount of packets for each iteration. That is, SLoPS-100 sends only 100 packets per iteration and SLoPS-5K sends 5,000 packets per iteration. The one-way delay values are still telling us whether or not we have an “increasing” or “decreasing” trend.

To run SLoPS, this required a significant amount of time creating the file that automates the process. Firstly, the file that automates the process, run_slops.sh must have the correct value for slops-send.py. This value (x) will determine which file stores the results for SLoPS and will run SLoPS-x. Also, this bash file must be run externally from the client, NF, and sender, so I ran it locally. This is because SLoPS involves sending packets from the client, using tcpdump to capture on both the client (capture timing of outgoing packets) and server. Doing all of this from my local machine was easier. This is because I decided that I could write a python program that could SSH into both the server and client and start multiple processes to get this all done. Doing so from the client would have been too difficult. As a result, slops-send.py contains a significant amount of code that SSH’s into the client and server, and runs code from those SSH sessions. Thus, to make this work, the github repository must be cloned on all three nodes. All results will be placed into a txt file and can be parsed by running “python3 slops-read.py <txt_file> <ground_truth_value>”. The <ground_truth_value> was obtained by the Iterative Drops method

mentioned earlier. Also note that for this to work, the correct firewall must be running on the NF node so that packets are processed (except for SNORT-RL where that is automated by my code). Make sure that the client is sending the correct type of packets (UDP for blacklist rules and TCP for all others), by editing scapy_stream.py in the NFTY directory.

Be sure to run run_slops_snort.sh to test SLoPS on SNORT-RL. The difference here is that sun_slops_snort.sh SSHs into the firewall and restarts Snort every time. This is done due to the functionality described in section 3.2. Since the ground truth that we obtain for SNORT-RL (using benchmark_SNORT-RL.sh) gives the processing rate for the beginning of Snort's process, we have to conduct all tests at this portion of Snort's lifetime.

When running SLoPS, the files client_read_check.py and scapy_stream.py are run on the client machine. Scapy_stream.py may need to be edited because that will either send UDP or TCP packets, so that will need to be configured. Also if the firewall is SNORT-RL, then the destination MAC may need to be the server's MAC, not the NF's MAC. Only perform this last change if no packets end up getting forwarded to the server when running SLoPS.

4. Methodology

4.0 NFTY

The paper's proposed method, Network Function capaciTY estimation (NFTY) uses a dispersion-based approach for both the two-way and one-way models. My research focuses only on the two-way setting. NFTY sends a burst of X packets that are received at the NF, then forwarded to the receiver. If the packets are sent at a rate that is fast enough (faster than the processing rate of the NF), then the packets are queued and processed. Since the NF will process one packet at a time (per thread), this processing will disperse the packets at time intervals that are equivalent to the packet processing rate. Thus, when the packets are received at the receiver, we can inspect their arrival times, and their dispersion from one another should

indicate how long it took the NF to process one packet at a time. From this, we can infer the processing capacity of the NF. With DVFS on, these dispersion values will fluctuate because the CPU's frequency will change. To account for this, NFTY segments the dispersion values using binary segmentation and uses only the smallest dispersion values (when the CPU was at its highest frequency and so this is the NF's absolute highest processing capacity). Figure 2 shows these changes in CPU frequency and with NFTY, we use the segment of dispersion values with the 17.3us above. To select this segment, NFTY takes the mean dispersion value for all segments and chooses the smallest of these mean dispersion values as the predicted NF processing rate. In figure 2, 17.3us is NFTY's predicted processing rate per packet. NFTY sends bursts of either 100 packets for NFTY-100 or 5,000 packets for NFTY-5K.

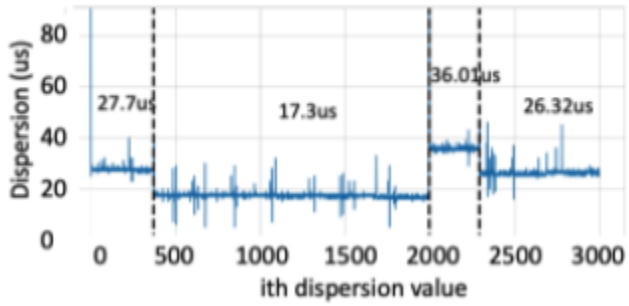


Figure 2: NFTY uses binary segmentation then takes the minimum of the mean dispersion values for each segment.

To perform 100 experiments of NFTY, as I did, you must send 100 bursts of packets that are spaced apart by at least 2 seconds. This is because the code that performs the experiments starts a new analysis for probes that are spaced more than 2 seconds apart. To send these bursts, run send_traffic.sh such that send_traffic.py is called with the correct arguments. The

arguments determine the size of the burst and whether or not the NF is multithreaded. When these bursts are being sent, the NF must be running with the correct firewall and software, and the server should have tcpdump, recording the received probes into a pcap file. After sending the probes, I copied over the pcap file to my local machine to run “python3 nfty_estimate.py <pcap_file> <nf_type> <setup>”. Check ground_truths.py to see how to store the ground truths of the NFs and in there, you can define “nf_type” and “setup”. The median of “mean SCI” is the Median Absolute Percentage Error (MdAPE) for the 100 experiments of NFTY. Be sure to record the array of error values beside the MdAPE to record the distribution of errors. You can also tune the binary segmentation by changing the penalty in nfty_estimate.py. The lower the penalty, the more segments are created, and the higher it is, the fewer the segments are created. Finally, if nfty_estimate.py is being run on SNORT-RL, the firewall will have to be restarted, so use the code that SSH’s into the firewall and starts up SNORT-RL for every probe.

As discussed earlier, there will be multithreaded queues NFs that assign packets based on flow. To counter this, NFTY spoofs packets using a random function that assigns IPs from a range of valid IP addresses that have similar paths. As a result, these packets are not dropped and, due to the random distribution that is distributed evenly, all packets will be distributed among all threads evenly.

4.1 Preliminary Experiments and Verification

The beginning of my experiments first began with properly establishing a network environment that would allow me to properly send packets from a client to server such that all packets are processed at the NF. The NFTY paper that I contribute to already had experiments and results for the controlled environment (see figure 4) and the Internet2 environment (see figure 3). Thus, my goal was to replicate their results for both environments. If successful, I would be verifying their results and establish the fact that my settings were all configured

properly and the only changes that I would need to make would be changes to the topology to conduct the Internet experiments.

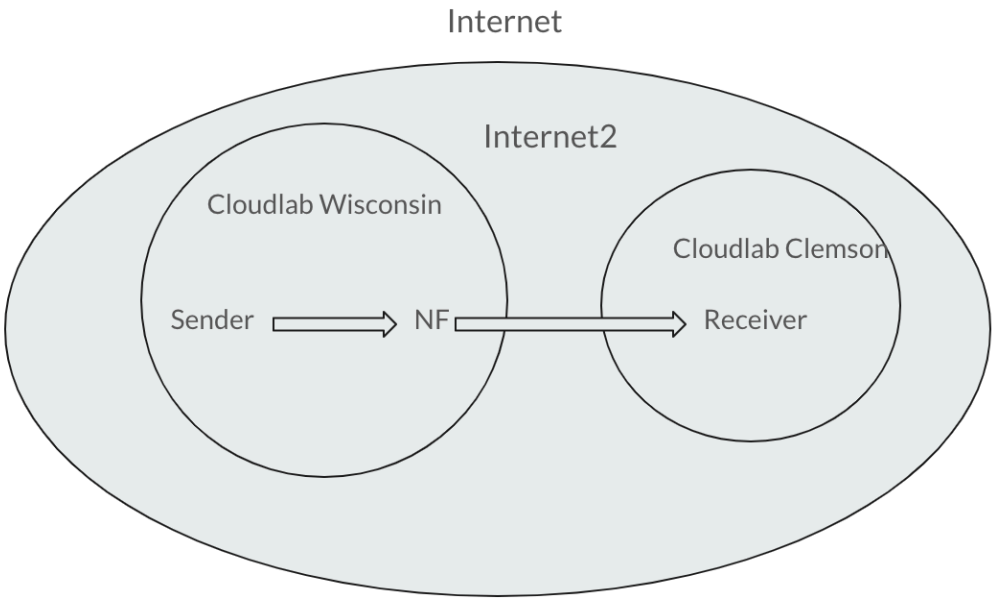


Figure 3: Topology for Internet 2 Experiments. Internet2 is different from Internet because Cloudlab uses its own network services to let nodes communicate with each other. This is like a University network.

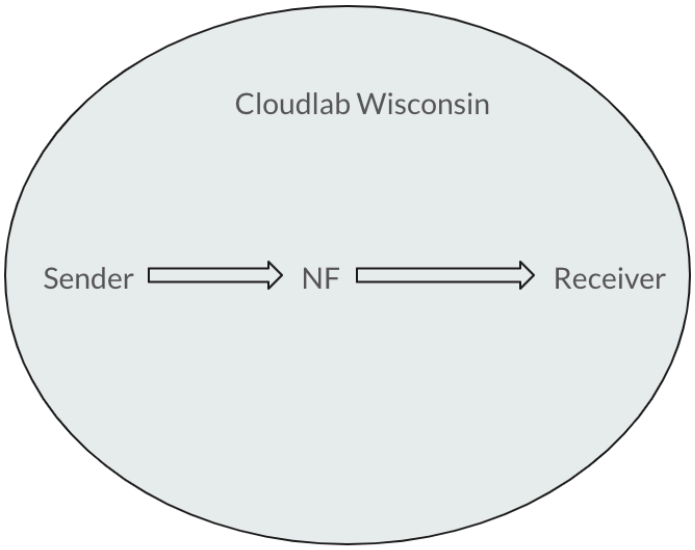


Figure 4: Topology for Controlled Experiments

4.11 Controlled Experiments

First, I verified the paper's ground truth results in the controlled environment (figure 4). Here, all nodes are in Cloudlab Wisconsin and are c220g2. The source file is given in Controlled.py. This is controlled because the NF is directly forwarding the packets to the receiver with no hops between. To make sure that the nodes were properly configured, I had to have them ping each other, and then run "sudo sysctl -w net.ipv4.ip_forward=0" for the NF, then ping the receiver from the sender, and ensure that no response was found.

To verify the experiments, I first obtained the ground truth using benchmark_<nf_name>.sh (where the IPs were in the private IP address range) for just SUR-RL and SUR-BL. The paper gives 142,000 pps and 200,880 pps as the ground truth packet processing rates for SUR-BL and SUR-RL, respectively. My experiments ended up

obtaining 148,684, and 212,381. Not all nodes in the Wisconsin Cloudlab cluster that are c220g2 have the same performance, so these slight differences are not unexpected.

After obtaining these ground truths, I knew that my configuration was correct for Suricata and the nodes, and so moved on to verifying the Internet2 experiments.

4.12 Internet2 Experiments

The topology for my Internet2 experiments is given in figure 3. I used Internet.py as my profile for the Wisconsin nodes. Here, the sender and NF are in Cloudlab Wisconsin (c220g2 again), but the receiver is in Cloudlab Clemson (c6320 node which has the closest configuration and performance as the c220g2 node). Again, I obtained the ground truth for SUR-BL and SUR-RL as 145,463 pps, and 218,065 pps, where the original NFTY paper found them to be 134,640 and 193,750 pps. These values are relatively close - close enough to confirm that I could move on to the next step: experiments on the Internet.

4.2 Internet Experiments

The crux of my contribution is my experiments that use the topology conveyed in figure 0. These experiments utilize a server that is a Vultr node in Los-Angeles, where the sender and NF are, again, in Cloudlab Wisconsin. Refer to section 3.0 for an in-depth description of this topology. As before, my first goal was to obtain the ground truth for all of the NFs using the benchmark bash file. Since I had only familiarized myself with SUR-RL and SUR-BL, using SNORT for the first time took a while. It also took time configuring the multithreaded NFs properly. After obtaining the ground truth, I then ran the 100 experiments for NFTY-100 and NFTY-5K at different times of day: 10am, 4pm, and 9pm. This was done for all NFs. Afterwards, I then ran the SLoPS experiments such that I obtained 50 results for SLoPS-100 and SLoPS-5K. I did not run SLoPS at different times of day because it took a while to run just 50

experiments. Approximately, each experiment for SLoPS-5K sent 10 bursts of packets where it took (safely) 1.25 minutes to process each batch of packets and determine whether or not the trend was increasing or decreasing. Thus, it took around 13 minutes for each experiment, which resulted in a total runtime of almost 11 hours to run all 50. NFTY-5K, on the other hand, only took 5 seconds per probe when sending packets through the NF. This is approximately 8 minutes to send all probes. When running nfty_estimate.py for all 100 experiments, it took on average 30 minutes to output 100 predictions. Thus, it took at most 40 minutes to run 100 NFTY experiments. Comparing NFTY-5K to SLoPS-5K, accounting for the difference in experiment number, NFTY is 32 times faster! After all of the results for SLoPS were placed into a file, I then ran slops-read.py to give me the MdAPE of SLoPS and the distribution of errors.

4.2 Challenges

There were many challenges that I had to overcome throughout this process. The first challenges were adapting the code in the original NFTY repository to fit my environment with new IPs and MACs. The next challenge was properly configuring the network topology for the Internet2 and Internet experiments. Initially, I set up the NF as the default gateway for the sender, but this made things very difficult since this forced all packets (including SSH packets) to go through the NF and through the sender's private interface. Using sendpfast() where I specified the output interface solved this since, in Cloudlab, the sender and NF already had a link at their private interfaces.

When dealing with Snort, it brought many challenges: Afpacket had odd functionality, it showed signs of multithreading, the ground-truth was adaptive, and there were odd drops happening in NFQUEUE mode when Suricata was not dropping any packets. All of this was mitigated by my process given in section 3.2.

The multithreaded-ness was also difficult to handle for the multithreaded NFs. At first I noticed that all packets were going to one queue, but NFTY is supposed to spread packets among all threads. Through experimentation, I then found that spoofing the IPs using a random function was the best way to evenly distribute the packets. The spoofing had to be done properly though. I had to give valid IPs that would have a similar path, so that they would not be dropped on-path. Thus, I resorted to using CloudLab's public IP space range (the IP's of other nodes chosen randomly).

When moving to Internet experiments, I also ran into the issue of where the receiver node in Vultr was not fast enough to process the received packets, and so rather than having the NF as the bottleneck, the receiver was the bottleneck. This led to very odd ground-truth estimates for the faster NFs. As a result, I had to restart my experiments with a new and faster receiver Vultr node.

There were more minor challenges that were encountered throughout the early configuration stages. This includes downloading and configuring Suricata and Snort and ensuring that they worked properly. Many times, they would not process packets or would randomly drop just UDP or TCP packets. I had to go through the config files, log incoming and outgoing packets, and inspect their headers to ensure that everything was working properly. Many times, it was a very small error on my end. Also, there were version inconsistencies, I had the wrong version of Python or Rust, did not update "apt-get", or forgot to run a command with "sudo". One fatal mistake I made was that I sent UDP packets when I meant to have sent TCP packets and vice versa.

5. Results and Analysis

Network Function	Measured Ground Truth (pps)
Suricata-BL	144940
Suricata-RL	227726
Snort-RL	132388
Suricata-BL-Multithread	246679
Suricata-RL-Multithread	348795

Table 1: All ground truth values for the NFs when packets are sent across the Internet

Comparing Table 1 to the actual paper, the ground truth for all NFs is very similar to the controlled and Internet2 ground truths. Note that the ground truth does and is supposed to change depending on the environment that we are in. This is because the environment can have an influence on packet processing operations and timing of queueing. We are only different by at most approximately 10%, which is expected, except for SNORT-RL. This is potentially because the paper uses SNORT-RL in Afpacket mode, not NFQUEUE like we did. Afpacket is generally slower. Also, our receiver is very fast, and so this could make SNORT-RL much faster for us (as well as all of the other NFs).

Network Functions	NFTY-100 Performance (Absolute Percentage Error (%))					
	10am		4pm		9pm	
	Mean	Median	Mean	Median	Mean	Median
Suricata-BL	4.121	2.922	4.103	3.147	10.636	10.48
Suricata-RL	4.849	3.755	6.167	5.262	6.650	5.286
Snort-RL	6.848	5.928	6.521	5.452	5.942	5.028
Suricata-BL-ML	23.258	22.746	25.007	25.697	25.599	25.194
Suricata-RL-ML	32.314	32.016	31.771	30.433	32.561	31.934

Table 2: NFTY-100's performance at different times of day

Table 2 shows us that the performance of NFTY-100 at different times of day remains relatively constant. At 9pm, we do notice that the accuracy is reduced. This is likely caused by greater noise in the dispersion values caused by higher traffic in the evenings. Performance seems to be the best at 10am which is likely when traffic is at the lowest during the day. We also notice that the performance for the multithreaded NFs is quite poor. This can be explained

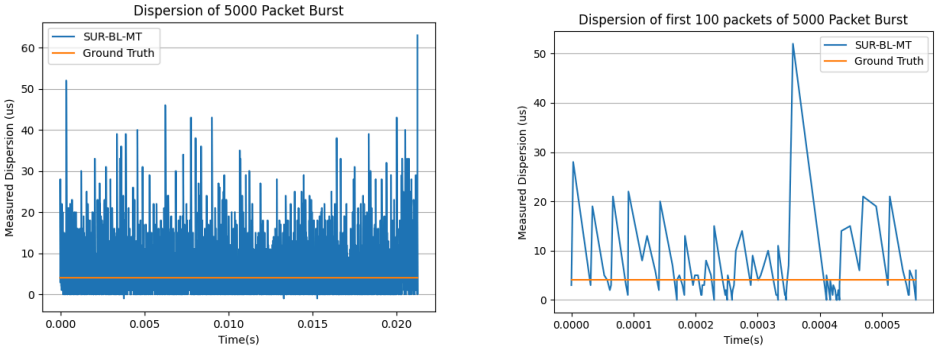


Fig 5 (left) + 6 (right): Dispersion of 5000 packets sent through SUR-BL-mt (Fig 5) and Dispersion of its first 100 packets (Fig 6)

By looking at dispersion values. Looking at Figure 5, we can see a very slight gradient at the bottom left corner - the absence of any 0 dispersion values. Looking more closely at the first 100 packets (given in Figure 6), we can see this gradient more clearly. This absence of low dispersion values leads to an underestimation of packet processing capacity which results in the high MdAPE. An explanation for why this happens is given in my work in the official paper (at the end of this report) in section 6.4.

Network Functions	NFTY-5000 Performance (Absolute Percentage Error (%))					
	10am		4pm		9pm	
	Mean	Median	Mean	Median	Mean	Median
Suricata-BL	1.413	1.291	3.607	3.504	0.707	0.334
Suricata-RL	2.998	2.92	3.093	3.067	1.628	1.570
Snort-RL	3.718	2.573	3.437	1.813	4.648	4.332
Suricata-BL-ML	2.861	2.863	3.385	3.124	3.32	3.167
Suricata-RL-ML	3.922	3.911	2.775	2.856	4.495	4.506

Table 3: NFTY-5K's performance at different times of day

Table 3 indicates that, like for NFTY-100, the performance of NFTY-5K remains constant throughout the day. Interestingly, the performance of NFTY-5K improves at 9pm for SUR-BL and SUR-RL. This could actually indicate that the ground truth does not remain constant throughout the entire day, and so there are very small fluctuations caused by Internet network traffic. An alternative explanation would be that at the beginning of a probe (the first 100 packets, for example) experiences most of the noise which results in the larger change in performance for NFTY-100 when higher traffic is seen. So, when we send a burst of 5,000 packets, the noise at the beginning of the probe has little to no effect on our results. This follows the explanation for the poor performance of NFTY-100 on the multithreaded NFs, thus this reasoning is the most likely explanation for the variation in performance at 9pm for NFTY-5k.

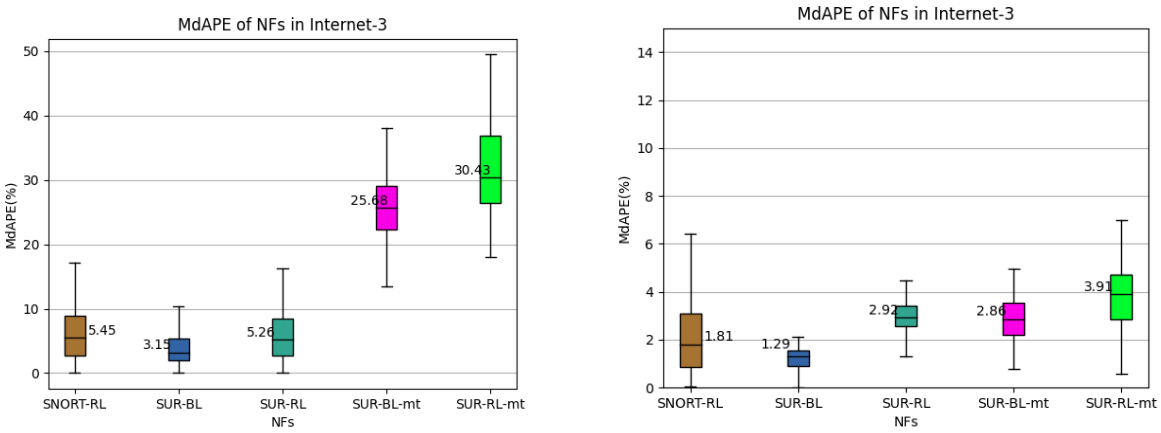


Figure 7 (left) , 8 (right): Distribution of errors of NFTY-100's performance at 4pm (Fig 7) and distribution of errors of NFTY-5K's performance at 10am (Fig 8)

Figure 7 indicates that as the processing rate increases, the MdAPE and spread of errors also increase. Figure 9 also shows this trend. This trend is described and found in the original paper for the controlled and Internet2 experiments, showing that my results are aligned with the original paper's. The one outlier is SNORT-RL. Interestingly, SNORT-RL's MdAPE is slightly higher than SUR-BL's, but its spread of errors is surprisingly large. This could be due to a couple reasons: the variant ground truth was accidentally captured during the probing, or SNORT's packet processing rate is noisier when packets are sent across the Internet. The latter reason implies that SNORT's ground truth is noisy and does not stay as constant at the beginning of its process lifetime as my paper assumes.

Network Function	SLOPS-100	
	Mean Abs. Perc. Error (%)	Median Abs. Perc. Error (%)
Suricata-BL	16.554	11.191
Suricata-RL	31.262	12.452
Snort-RL	20.933	10.551
Suricata-BL-MT	46.119	42.375
Suricata-RL-MT	49.841	44.545

Table 4: Performance of SLoPS-100 throughout the entire day

Network Function	SLOPS-5000	
	Mean Abs. Perc. Error (%)	Median Abs. Perc. Error (%)
Suricata-BL	5.383	2.219
Suricata-RL	4.195	2.594
Snort-RL	8.951	5.166

Suricata-BL-MT	49.874	49.113
Suricata-RL-MT	47.298	45.506

Table 5: Performance of SLoPS-5K throughout the entire day

Tables 5 and 6 show the performance of both SLoPS-100 and SLoPS-5K. Upon initial inspection of Table 5, we can see the poor performance of SLoPS-100 where its MdAPE exceeds 10% and its Mean Absolute Percentage Errors (MAPE) drastically differ from the MdAPE. This implies that there is massive variation in all experiments and so the results are very noisy and inconsistent. Table 5 indicates that SLoPS-5K is far more consistent and massively improves the performance of packet processing capacity estimation. Both figures also show that the performance of SLoPS for the multithreaded NFs was very poor. This is because SLoPS does not account for the multithreaded-ness of the NFs and so sends all packets to one queue (since all packets look like they are coming from the same source = one flow). Thus, SLoPS's estimation only estimates the processing capacity of one thread/queue.

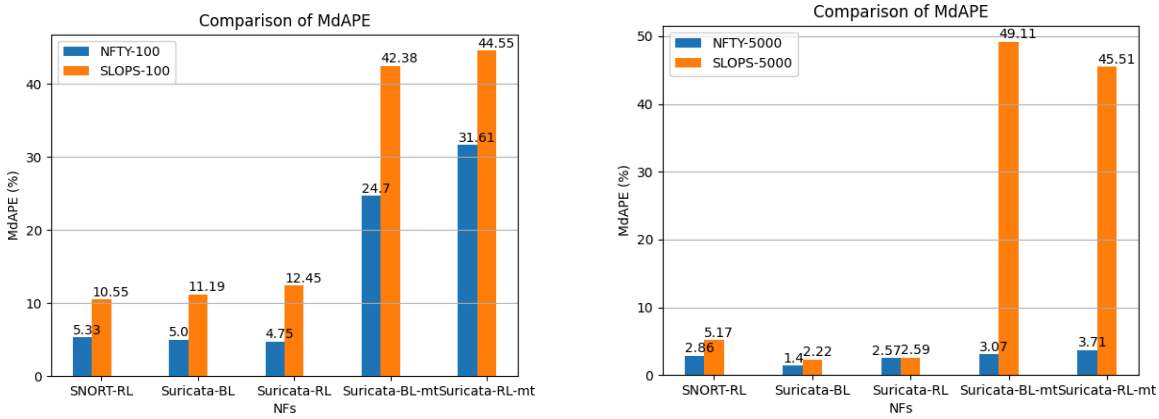


Figure 9 (left), 10 (right): Comparison of SLoPS-100 and NFTY-100 with all results from time of day aggregated (Fig 9), and comparison of SLoPS-5K and NFTY-5K with all results from time of day aggregated (Fig 10)

These two figures are included in the original NFTY paper that is appended to the end of this report in section 6.4. Refer to that section for an explanation of figures 9 and 10. One important thing to note is that SLoPS-5K sends bursts of 5,000 packets for each iteration. On average SLoPS-5K ended up sending 10 bursts, so 50,000 packets. Compared to NFTY-5K, SLoPS sent 10 times more packets. The same can be said about SLoPS-100 versus NFTY-100: SLoPS, on average, sent 1,000 packets in total, so 10 times more packets than NFTY-100. So, although SLoPS sent many more packets than NFTY, we found that NFTY always outperformed SLoPS. This large amount of packets that SLoPS sends iteratively makes it highly detectable compared to NFTY.

Conclusion and Next Steps

Network Function Capacity Reconnaissance can provide very helpful information for both adversaries and network operators. Current methods do not provide the undetectability, ease, accuracy, and quickness that should be characteristic of these methods. Network Function capaciTY estimation (NFTY) provides these characteristics by using a dispersion based algorithm with binary segmentation for the two-way model. My research focused on the effectiveness of NFTY when packets are sent across the Internet and found that NFTY remains undetectable, quick, and accurate. When compared to an iterative based approach, NFTY outperforms SLoPS in accuracy, quickness, and detectability. Future research and steps for NFTY could look at testing NFTY's effectiveness with cross traffic, as well as looking into refining the binary segmentation to remove any outliers in the dispersions of the chosen packet train.

Acknowledgements

I would like to thank my advisor Maria Apostolaki for the continued guidance, mentorship, and support that she has provided me with since the beginning of the year. I am grateful that she invited me to join her NFTY research group with Aqsa Kashaf. I would also like to thank Aqsa Kashaf for taking me through a lot of the process and helping me whenever I hit a roadblock.

Appendix

A Code

Please refer to my github repository https://github.com/Aidan-Walsh/NFTY-Internet/tree/main for all code that I reference throughout this documentation.

B Honor Code

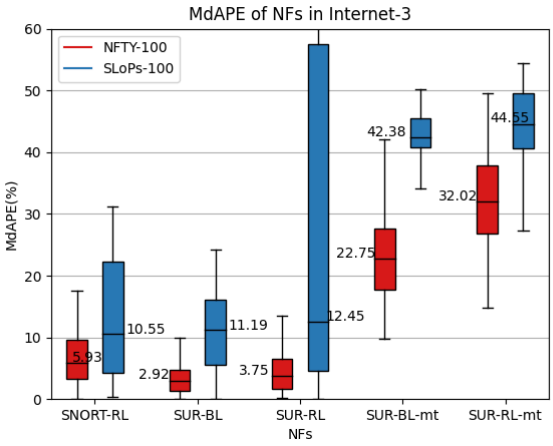
I pledge my honor that I have not violated the Honor Code.

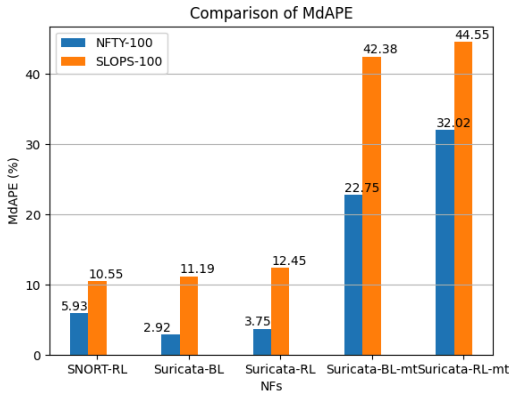
~Aidan Walsh

C Graphs

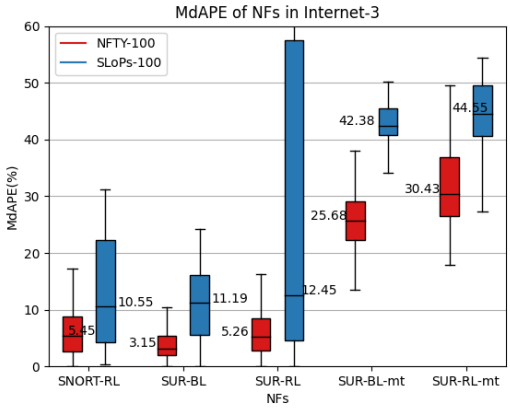
The Graphs below shows NFTY-100 at time X compared to SLOPS-100. The first graph is a boxplot showing the distribution of all MdAPE values. The second graph is a bar chart just comparing the median of the MdAPE values.

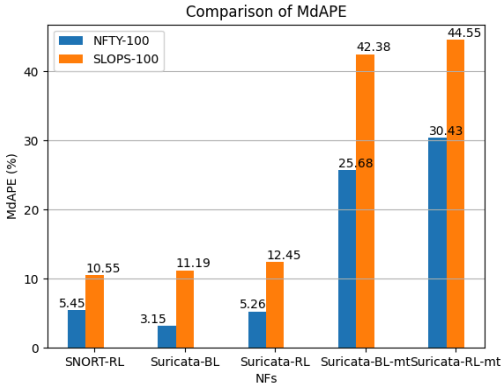
<u>X = 10am:</u>



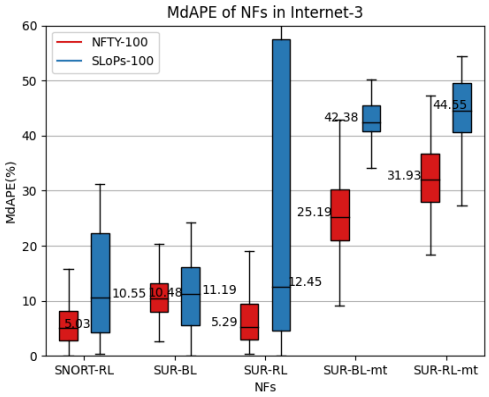


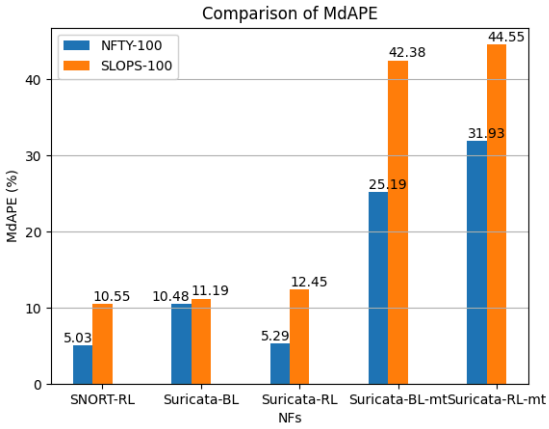
<u>X = 4pm:</u>



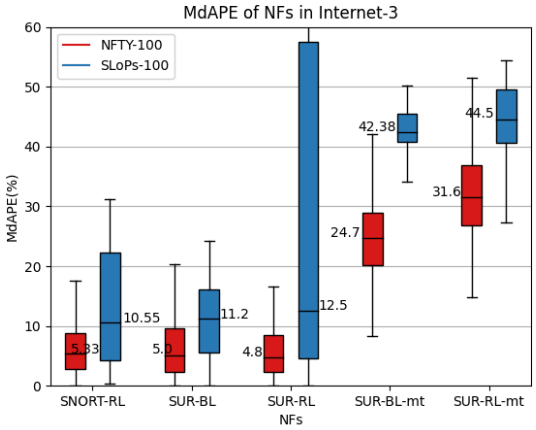


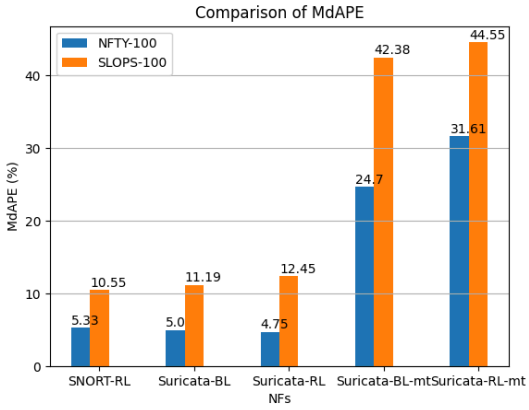
X = 9pm:





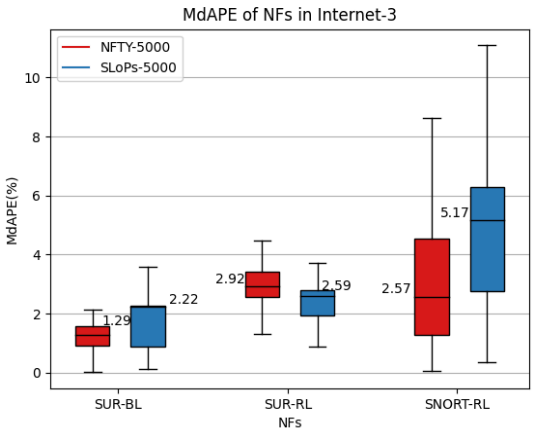
The graphs below show NFTY-100 (all times aggregated) compared to SLOPS-100. The first graph is a boxplot showing the distribution of all MdAPE values. The second graph is a bar chart just comparing the median of the MdAPE values.

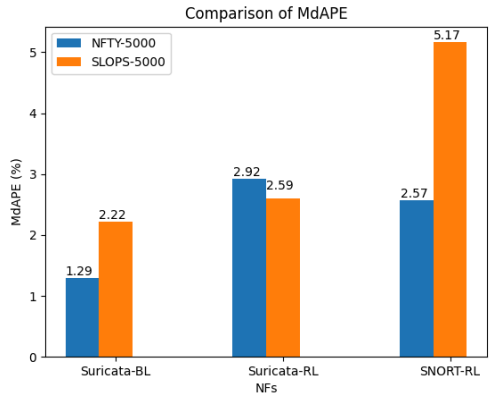


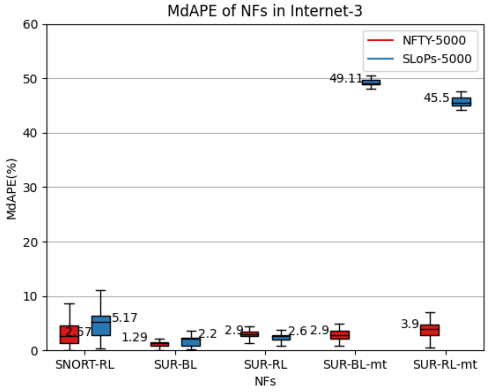


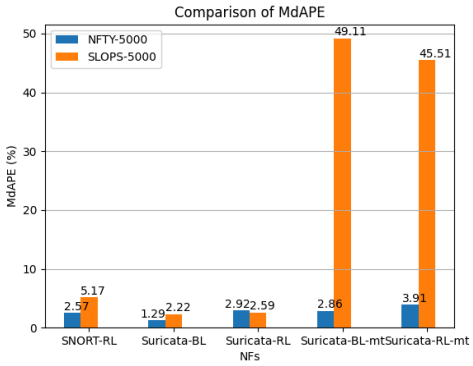
The Graphs below shows NFTY-5000 at time X compared to SLOPS-5000. The first pair of graphs is only of 3 of the 5 NFs (the multithreaded NFs cause the graphs to look odd due to the spread). The second pair is the results of all 5 NFs. In each pair, the first graph is a boxplot showing the distribution of all MdAPE values. The second graph is a bar chart just comparing the median of the MdAPE values.

<u>10am:</u>

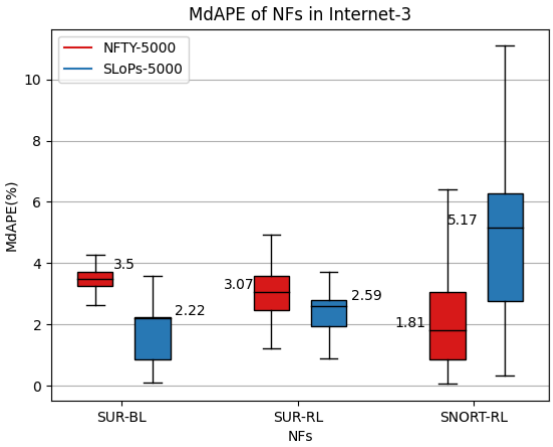


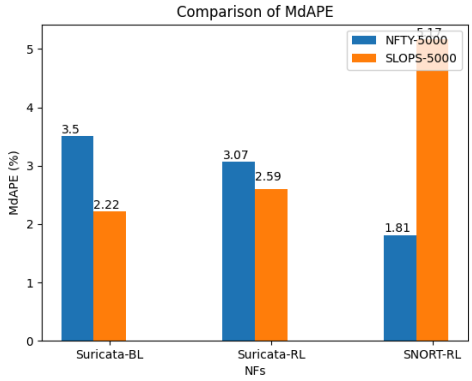


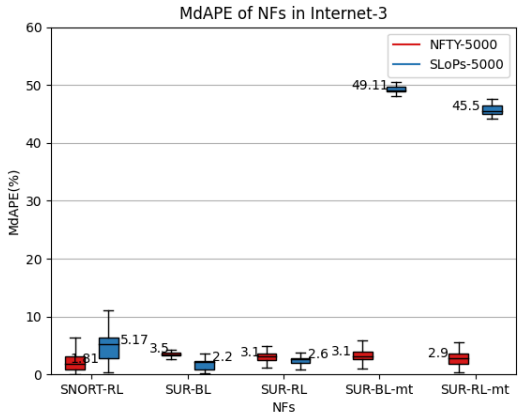

MdAPE of NFs in Internet-3
NFTY-5000
SLoPs-5000
MdAPE(%)
NFs
SNORT-RL
SUR-BL
SUR-RL
SUR-BL-mt
SUR-RL-mt
49.11
45.5
5.17
1.29
2.2
2.6
2.9
3.9

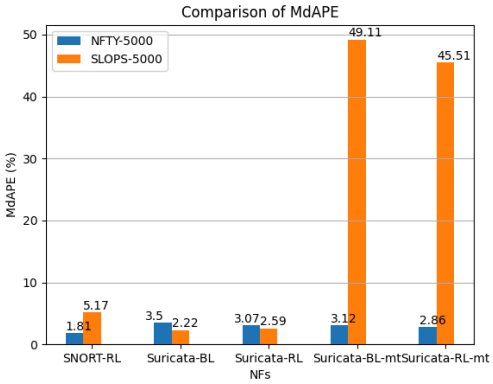

Comparison of MdAPE
NFTY-5000
SLOPS-5000
MdAPE (%)
NFs
SNORT-RL
Suricata-BL
Suricata-RL
Suricata-BL-mt
Suricata-RL-mt
49.11
45.51
5.17
2.57
1.29
2.22
2.92
2.59
2.86
3.91

4pm:

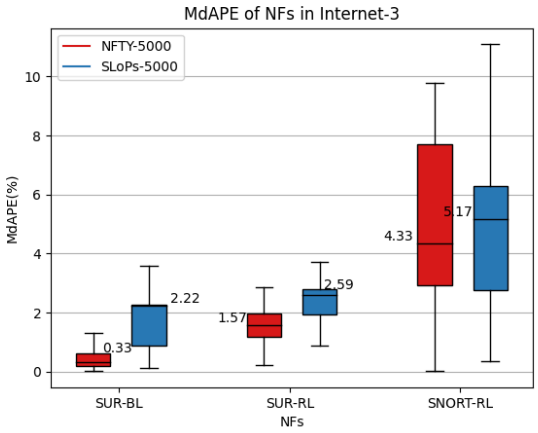

MdAPE of NFs in Internet-3
NFTY-5000
SLoPs-5000
MdAPE(%)
NFs
SUR-BL
SUR-RL
SNORT-RL
3.5
2.22
3.07
2.59
1.81
5.17

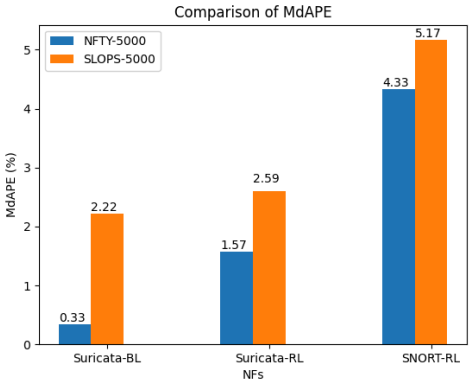

Comparison of MdAPE
NFTY-5000
SLOPS-5000
MdAPE (%)
NFs
Suricata-BL
Suricata-RL
SNORT-RL
3.5
2.22
3.07
2.59
1.81

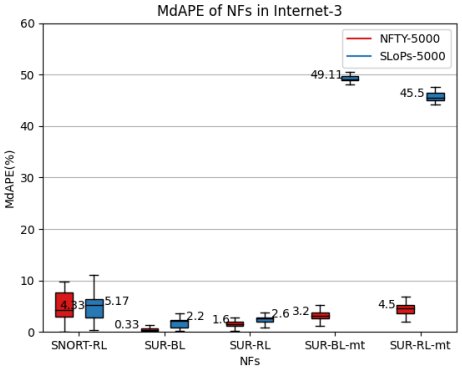

MdAPE of NFs in Internet-3
NFTY-5000
SLoPs-5000
MdAPE(%)
NFs
SNORT-RL
SUR-BL
SUR-RL
SUR-BL-mt
SUR-RL-mt
49.11
45.5
5.17
3.5
2.2
3.1
2.6
2.9

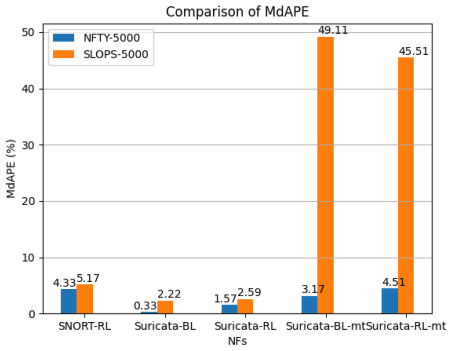

Comparison of MdAPE
NFTY-5000
SLOPS-5000
MdAPE (%)
NFs
SNORT-RL
Suricata-BL
Suricata-RL
Suricata-BL-mt
Suricata-RL-mt
49.11
45.51
5.17
1.81
3.5
2.22
3.07
2.59
3.12
2.86

9pm:









References

[1] ALLMAN, M. Measuring end-to-end bulk transfer capacity. Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement (2001), 139–143.

[2] CARTER, R., AND CROVELLA, M. Measuring bottleneck link speed in packet-switched networks. Performance Evaluation 27 (1996), 297–318.

[3] JAIN, M., AND DOVROLIS, C. End-to-end available bandwidth: measurement methodology, dynamics, and relation with tcp throughput.

ACM SIGCOMM Computer Communication Review 32 (2002), 295–308.

[4] RAMAMURTHY, P., SEKAR, V., AKELLA, A., KRISHNAMURTHY, B., AND SHAIKH, A. Remote profiling of resource constraints of web servers using mini-flash crowds.

[5] SALEHIN, K., ROJAS-CESSA, R., BI LIN, C., DONG, Z., AND KIJKAN-JANARAT, T. Scheme to measure packet processing time of a remote host through estimation of end-link capacity. IEEE Transactions on Computers 64, 1 (2013), 205–218.

[6] SAMAK, T., EL-ATAWAY, A., AND AL-SHAER, E. Firecracker: A framework for inferring firewall policies using smart probing. In 2007 IEEE International Conference on Network Protocols (2007), 294–303.

Network Function Capacity Reconnaissance by Remote Adversaries

Please do not distribute, Paper #341, 12 pages

Abstract

There is anecdotal evidence that attackers are using reconnaissance to learn the capacity of their victims prior to DDoS attacks to maximize their impact. We find, however, that there is little known about the feasibility of such capacity reconnaissance in the context of network functions (e.g., firewalls, NATs, DDoS filters). A first step to design mitigation strategies or reconnaissance-proof designs is understanding the feasibility and limits of capacity reconnaissance. To this end, we formulate the problem of network function capacity reconnaissance (*NFCR*), and explore the feasibility of inferring the processing capacity (*i.e.,* the maximum packet processing rate) of a network function while sending a small number of probes to avoid detection. We identify key factors that make *NFCR* challenging and analyze how these factors affect the trade-offs between accuracy of capacity estimates (measured as divergence from ground truth) and stealthiness (measured in packets sent). We propose a practical tool called *NFTY* that performs NFCR. Based on our experiments, we present two practical *NFTY* strategies representing two different points in the detection-accuracy trade-off space. We evaluate them both in a controlled lab environment as well as across real-world Internet settings with commercial NFs. Our results show that *NFTY* can accurately estimate the capacity of different *NF* deployments within 10% error in the controlled experiments and in the Internet. *NFTY* also estimates the capacity of a commercial *NF* deployed in cloud (AWS) within 7% error. Moreover, *NFTY* outperforms link-bandwidth estimation baselines by up to 30x. Finally, we propose and study various countermeasures.

1 Introduction

Attackers seek to optimize their impact by using reconnaissance probes to assess a target's resources before launching an actual attack. For example, prior work has shown attackers can use topology information to launch sophisticated denial-of-service (DoS) attacks and concentrate their efforts on important nodes or links of a targeted network [30, 41].

Particularly in attacks against critical network infrastructures, a significant aspect of reconnaissance involves estimating the capacity of network functions (NFs), which implement crucial operations, including DDoS mitigation, firewall management, and network translation. Anecdotal evidence suggests that attackers might be conducting such reconnaissance to estimate the capacity of NFs before initiating DDoS attacks against high-value targets (e.g., [8, 53, 56]). Accurately gauging the capacity of various network function enables an adversary to identify the most vulnerable targets and estimate the necessary resources (e.g., number of packets, attack rate) for a successful attack.

Despite the potential risks and impacts of such attacks, little is known about the feasibility of Network Function Capacity Reconnaissance (*NFCR*). This paper aims to raise awareness of *NF* reconnaissance attacks and guide potential targets towards practical countermeasures. To the best of our knowledge, we are the first to comprehensively define the problem of *NF* Capacity Reconnaissance (*NFCR*). Specifically, we are interested in answering the following question: Can an attacker infer the processing capacity of an *NF* remotely while remaining undetected (*e.g.,* without sending too many packets)?

At first glance, it appears that an attacker can leverage the significant prior work on link capacity estimation [15, 29, 31, 34] for *NFCR*. Yet, we find that link-capacity estimation techniques are ineffective in the *NFCR* context due to several unique challenges. First, unlike link-capacity estimation techniques, which often involve sending large volumes of traffic[38, 54], the number of packets sent for *NFCR* is crucial for its success. Intuitively, sending more traffic increases the likelihood of the target detecting the impending attack and bolstering capacity. Second, most link capacity estimation techniques [15, 52] require control over two vantage points before and after the link of interest, which is not always easy to achieve, as *NFs* can be deployed within private networks. Third, *NFCR* is heavily affected by optimization in the NF deployment such as multi-threading, CPU frequency scaling *etc.* Such optimizations are non-applicable for network links whose capacity is stable, hence the bandwidth estimation techniques [14, 15, 18, 26, 29, 31, 33, 42] do not work in those settings.

Despite the challenges, we demonstrate the practicality of *NFCR* by designing and implementing a functional attacker strategy, which we call *NF* capaciTY estimation (*NFTY*) that would allow an attacker to remotely and accurately estimate an *NF*'s capacity while sending a relatively small number of packets to evade detection. At a high level, *NFTY* operates by sending a burst of packets designed to momentarily stress the NF, revealing its true capacity despite hardware and software implementation. *NFTY* measures the dispersion of the packets, —a concept borrowed from link-capacity estimation techniques— but strategically processes the measurements to reduce the estimation error. *NFTY* is not only much more accurate than link-capacity estimation baselines (up to 30x) but also generalizes to an attacker controlling only a single vantage point by leveraging the TTL-exceeded mechanism in Internet routers.

We design *NFTY* to be configurable depending on the attacker's capability (*e.g.,* vantage points at sender and receiver), her knowledge of the target *NF*'s deployment (if any). Among all possible *NFTY* configurations, we identify two concrete representative strategies and evaluate them in lab settings and in the wild. The two attacks, namely *NFTY-100* and *NFTY-5K*, differ in the number of packets sent and thus their effectiveness and visibility. At a high level *NFTY-100* is stealthier but less accurate. Specifically, *NFTY-100* assumes an attacker that controls two vantage points and sends only 100 packets. *NFTY-100* can accurately estimate the capacity of simple NFs with 9% in the controlled environment and within 10% error in the Internet. We also evaluate *NFTY-100* against a commercial NF in AWS and find that it estimates the NF capacity within 7% of its true capacity while completely avoiding detection. *NFTY-5K*, in contrast, works with an attacker controlling one or two vantage points and sending 5K packets. We find that *NFTY-5K* estimates with up to 9% error in the controlled environment and within 9% error in the Internet. This is up to 30x more accurately than the bandwidth estimation baselines.

We use our insights gained while optimizing *NFTY* to propose practical countermeasures against *NFCR* attacks. Specifically, we consider countermeasures that aim at adding noise to probing traffic or obfuscating their infrastructure with rate-limiting and evaluating their effectiveness and performance overhead.

Contributions:

- We are the first to formally define and analyze the feasibility of *NF* Capacity Reconnaissance (*NFCR*).
- We investigate the applicability of bandwidth estimation techniques to *NFCR* and identify the unique challenges that differentiate *NF* capacity estimation from bandwidth estimation.
- We introduce an extendible methodology for conducting *NFCR*, namely *NFTY* that is accurate, robust, and practical under realistic scenarios for various NFs, and thread models, including the one-sided attacker scenario.
- We evaluate *NFTY* on both lab and Internet settings as well as against a commercial NF the cloud.
- We present a comprehensive suite of countermeasures designed to safeguard potential targets against NFCR attacks with limited overhead.

Ethical concerns: As with any "attack" paper in security, we run the risk that attackers can also benefit from our work. That being said, understanding the feasibility of attacks is a critical and important step for designing good systems and defenses. We have run our experiments in controlled settings on servers under our control, i.e., not running "attacks" against third parties. We have also taken precautions to ensure that we do not induce undue measurement load on shared testbeds and cloud networks.

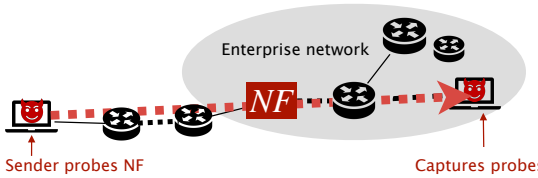


Figure 1: The *NF* processes the incoming traffic of the private enterprise. A two-sided attacker controls nodes on both sides of the *NF* (in and out of the enterprise), while a one-sided attacker controls nodes only outside the enterprise.

2 Problem Formulation

In this section, we introduce the general problem of *NF* Capacity Reconnaissance (*NFCR*) using a representative network setting and formally define our terminology. We start by describing the different entities that are part of an *NFCR* attack scenario. Next, we specify the threat model and formulate the attack strategy.

Setting: We assume an enterprise that deploys a network function (*NF*) (e.g. IPS, firewall) placed such that it observes all incoming traffic, as shown in Figure 1. network function comprises a set of processing tasks applied to a packet type t, with a well-defined goal, such as load balancing or intrusion detection. For example, a TCP SYN proxy NF is geared towards protecting against *TCP SYN* flooding attacks. An *NF* can be deployed on different hardware (*e.g.,* a server, a programmable switch) with resources configured statically or dynamically (*e.g.,* in a number of threads, cores, memory). We define capacity $\mathbb{C}_p$ of an *NF* as the maximum number of packets of type t (p^t) the *NF* can process per unit time. It can happen that an *NF* has different capacity for different packets of types.

Note that *NFCR* is not specific to a private enterprise setting. It generalizes to other settings, such as the cloud, in which the *NFs* are deployed by various third parties *e.g.,* the Azure Marketplace [9].

Threat Model: We consider two threat models depending on the attacker's capabilities as either: *(i)* one-sided; *(ii)* two-sided. In the two-sided threat model, the attacker controls a vantage point (VP) on both sides of the *NF*, which means that the attacker can send traffic from one VP (e.g., a host on the Internet) to the other (a host inside the enterprise network) via the *NF*. In the one-sided threat model, the attacker controls a single VP (e.g., a random Internet host). The attacker is unaware of the deployment of the *NF*; the topology of the target network and location of the *NF*; the congestion on the network path to the *NF* and the *NF* itself. The attacker aims to have a low network footprint to avoid detection. Beyond preventing being blacklisted, which can be circumvented, the attacker's low footprint prevents the victim from noticing the reconnaissance and ramping up resources in preparation for

the real attack. Intuitively, the more packets an attacker sends, the more visible and hence detectable they will be.

Problem Definition and Scope: Given this context, we formally define the *NFCR* problem as follows: Given an *NF* with unknown processing capacity $\mathbb{C}_p$ for packet type p^t and unknown deployment, construct an inference model that can remotely and accurately estimate the capacity $\mathbb{C}_p$ for packet type p^t by probing it with at most budget $\mathbb{B}$ packets.

In this paper, we restrict our focus to NFs for which the processing capacity does not depend on the packet history. The NF may maintain state but that does not affect its processing capacity. In fact, most Internet-facing NFs have this property to prevent an attacker from algorithmic attacks exploiting this dependency to deplete the NF's resources. We also only consider statically provisioned *NFs*, meaning that their allotted resources (cores, threads, memory, hardware) do not change with time. In fact, statically provisioned *NFs* are more common in practice, especially for small/medium enterprises, as dynamically provisioning an *NF* is complex. Automating the entire lifecycle of NFs, from provisioning to decommissioning, while handling workload fluctuations, real-time performance monitoring and decision-making, makes dynamic provisioning a highly complex and nuanced task.

We consider an attacker that knows the high-level packet types that the target *NF* processes (*e.g.,* TCP SYN or UDP). The capacity that the attacker will measure will correspond only to this packet type. Of course, the attacker can use packet types that correspond to commonly seen traffic e.g., TCP SYN, HTTP, DNS and observe whether the output changes to identify which are processed.

Finally, we implicitly assume that the *NF* is the bottleneck in the path (*i.e.,* its processing rate is lower than that of the links) which is the common case as Network Functions do more computation and are more complex than simply forwarding packets. In §**??** we describe a way of identify whether that is not the case.

3 Prior work and limitations

A natural starting point for an attacker is to use link-bandwidth estimation techniques for *NFCR*. Hence, in this section, we try such techniques against real NFs and show that they do not provide an accurate capacity estimate. Finally, we reveal the unique characteristics of *NFCR* that make traditional link-bandwidth estimation techniques ineffective.

3.1 Overview of Bandwidth Estimation :

Bandwidth estimation techniques can be grouped into three categories depending on the probing technique and the signal they use:

- Dispersion-based techniques typically send a sequence or pair of packets and measure the link-induced spacing at the receiver [15, 19, 27, 35].
- Rate-based techniques iteratively probe at varying rates until they find the minimum rate that causes the one-way delay of packets to increase signaling self-induced congestion [29, 47].
- Bulk-transfer techniques send a high volume of traffic for a longer duration and measure throughput (*i.e.,* after self-induced drops) at the receiver [12, 54]. Using the bulk transfer technique, an attacker might try to flood the *NF* by sending packets at a high rate until drops are observed.

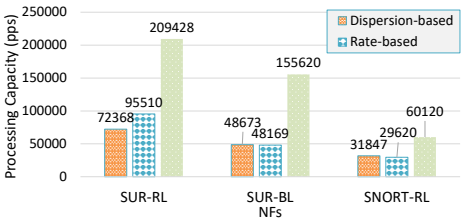


Figure 2: Traditional bandwidth estimation techniques underestimate the capacity of *NFs*.(maria: Green bar is not explained)

3.2 An Experimental Insight

We start by using each of these techniques to measure the capacity of realistic NFs built on top of Snort [5] and Suricata [6] and illustrate the results in Figure 2. We observe a huge discrepancy among the three approaches. Naturally, the bulk transfer is the most accurate: by offering excessive load, the NF runs at maximum capacity thus, the measured throughput is the capacity. While bulk transfer can serve as an experimental ground truth, it does not satisfy *NFCR*. As bulk transfer requires sending excessive traffic for an extended period to induce drops it will be extremely noticeable. Unfortunately, the less noticeable dispersion-based and rate-based techniques (orange and blue bars) are extremely inaccurate (maria: up to XX% error) [up to 65% and 54% respectively].

3.3 Unique Challenges of *NFCR*

Next, we describe the unique characteristics that make *NFCR* challenging and traditional bandwidth estimation techniques ineffective.

Optimized NF Deployments make NF capacity dynamic. Software and hardware optimizations introduce a level of variability in the immediate capacity of an NF, which can change based on current load conditions or the types of packets being processed. This fluctuation presents a significant challenge for Network Function Capacity Reconnaissance (NFCR) techniques, as they might not accurately capture the full potential of the NF's capacity. Importamtly, bandwidth estimation techniques do not deal with this problem as links have constant capacity. For example, to make better use of CPU power, modern OS and hardware support multi-threading. In our context multi-threading allows packets processed by the same Network Function (NF) to be distributed across various threads based on a specific scheduling algorithm. Depending on the efficiency of the scheduling algorithm, the aggregate process-

ing capacity of all threads can change. Similarly, to reduce power consumption, CPUs on servers can use Dynamic Voltage and Frequency Scaling (DVFS) to adjust frequency based on load. DVFS may cause the CPU on a machine running an *NF* to operate at a lower frequency, rather than the maximum, providing a lower estimated capacity.

Measurement infrastructure introduces noise. Intuitively, infrastructure is crucial for any measurement campaign. In the context of *NFCR*, batching in the receiver is particularly worrisome. Batching is used to reduce the number of kernel interrupts and increase the throughput of servers. Yet at the receiver, batching either in software (NAPI [50]) or hardware(IC [10]) will affect the packet dispersion measurements, leading to high error. Other potential confounding factors include the high sending rate, which is necessary for dispersion-based techniques, and fine-grained tuning of the sending rate in rate-based techniques. Notably, many existing capacity estimation techniques do not take these factors into account.

Two-sided estimation techniques do not generalize to one-sided. Traditional bandwidth estimation techniques typically assume two-sided control of the link. This is natural as bandwidth estimation techniques were designed as a benign tool for operators who could exert their control over the network. In this paper, we aim to investigate the effectiveness of an attacker performing *NFCR* who might only have one-sided power. Link bandwidth estimation techniques that work under the one-sided assumption typically rely on a particular protocol behavior involving specific packet types (*e.g.,* a TCP SYN on a closed port triggers a TCP RST [52] or an ICMP ECHO request triggers an ICMP reply [15]). packets.

4 Design

In this section, we begin with a high-level view of *NFTY*: our strategy for *NFCR*. *NFTY* relies on dispersion-based estimation to our problem setting. Then, we discuss how we can adapt this inference strategy in the setting when the attacker controls only one measurement endpoint. We conclude with an end-to-end view of how we configure and use the *NFTY* tool in practice.

4.1 High-level Overview

Consider an attacker who aims to estimate the maximum packet processing rate of an NF for a particular packet type (e.g. UDP). The NF is deployed at the edge of a victim network (as in Figure 1). The attacker might control only the sender (one-sided threat model) or both the sender and the receiver (two-sided threat model). If the attacker is two-sided *i.e.,* controls a receiver, the attacker measures the packet dispersion directly (Figure 3). If the attacker is one-sided, she engineers a subset of the sent packets to trigger an error message from an on-path router after the NF. Next, the one-sided attacker measures the dispersion between consecutively received error messages (§4.3).

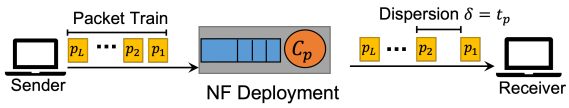


Figure 3: In the two-sided threat model, the time difference between consecutive packets increases as packets exit from the *NF*, revealing *NF*'s processing time ($\delta = t_p$), thus its capacity.

To this end, the attacker sends a set of UDP packets to a host such that they are processed by the NF. To remain stealthy, the attacker does not flood the NF or use rate-based iterative probing. Instead, the attacker measures the dispersion *i.e.,* the NF-induced time gap between consecutive packets, to infer the NF processing capacity. After being processed by the NF, the packets will be spaced in time depending on the NF processing time (Figure 3). The attacker manages to cause the *NF* to run at its maximum (true) rate while staying under the radar by sending a packet burst. The burst instantaneously stresses the *NF* and creates a clear step-wise pattern in the dispersion values that the attacker can recognize and use to find the true capacity. We elaborate on how the attacker adapts traditional dispersion-based techniques to the *NFCR* context in the next section.

4.2 Dispersion-based *NFCR*

We start with the basic background on dispersion-based estimation. We highlight unique challenges that our setting poses and how we tackle these in our design.

4.2.1 Background on dispersion-based estimation

Dispersion-based techniques seek to estimate the bandwidth of a bottleneck link by sending a probe of two (i.e., a pair) or more (i.e., a train) packets back to back and measuring their spacing at the receiver. Intuitively, for a packet pair, assuming that the link capacity of the bottleneck link is lower than the packets' inter-arrival rate, the second packet of a packet pair will be queued, waiting for the transmission of the first one. The time difference of the arrival between the two packets (measured at the destination) is called *dispersion* δ and provides the transmission time t_{tr} on the bottleneck link. We can then calculate the bottleneck link capacity C_l for a packet of size S as follows: $C_l = S/t_{tr} = S/\delta$.

This basic idea assumes that processing delays on the path are negligible compared to link transmission delays. Yet, if the pair traverses an *NF*, the measured dispersion depends on the *NF*'s processing t_p time, which is typically higher than transmission time since typical *NF*'s do more than just forwarding. Hence, to adopt dispersion-based bandwidth estimation for *NFCR* where the packet also traverses an *NF*, the dispersion is:

$$\delta = \max(t_{tr}, t_p) = \max(S/C_l, 1/\mathbb{C}_p) \quad (1)$$

Thus, if we assume that the *NF*'s processing time t_p is larger than the transmission time t_{tr}, then measured dispersion corresponds to the former. As an illustration, in Figure 3, the sender

forwards back-to-back packets, and they will be queued at the *NF*. For each p_i, the next packet p_{i+1} will only be processed after the *NF* is done processing p_i (as the *NF* is single-threaded). In other words, packets are dispersed by the time it takes to process a packet at the *NF*. Hence, the amount of dispersion between packets δ at the receiver gives the processing time t_p of the *NF*. From this dispersion δ between packets, the processing capacity $\mathbb{C}_p$ of the *NF* can be estimated as we discuss next.

4.2.2 Challenges in using dispersion-based estimation

Unlike link bandwidth estimation, estimating the processing capacity of an *NF* using dispersion requires careful consideration of several factors.

Challenge 1: Due to DVFS, not all dispersion values correspond to the true capacity: Consider an *NF* that runs on a CPU configured with DVFS. The CPU will operate at a lower frequency under a lighter load, transitioning to a higher frequency under a heavier load driven by the OS or kernel. In the context of *NFCR*, DVFS can cause the observed rate at which packets are processed to change, as a CPU might undergo multiple transitions to different frequencies before reaching its maximum frequency. As a result, the measured dispersion values do not all correspond to the true capacity. For example, Fig. 4a shows the dispersion values that will actually be observed if DVFS is enabled at the *NF*. The dispersion values go through multiple transitions. DVFS is not applicable to links and hence, link bandwidth estimation techniques cannot measure NF capacity accurately in the presence of DVFS as we show later in Section 6.

Challenge 2: Multi-threading and batching of packets introduce spikes in dispersion values: As a result of multi-threading at an *NF* or batching at receiver, there can be spikes in the dispersion values as shown in Figure 4b. While batching may result in periodic spikes, hence many link bandwidth estimation works have tried to detect the batching interval to correct for it in the measurements []. However, multi-threading in *NF* may result in irregular spikes depending on the distribution of load across threads, making it even harder to detect the number of threads. In the case of a multi-threaded *NF*, the capacity is the average capacity of the *NF*, including all threads over a period of time.

A straightforward approach for the estimation function F given a set of dispersion values $D = \{\delta_1..\delta_{L-1}\}$ would be to choose the minimum dispersion value. In this case $F(D) = min(\delta_1, \delta_2, ..., \delta_L)$ and the capacity $\mathbb{C}_p$ is given by $\mathbb{C}_p = 1/\min(\delta_1, \delta_2, ..., \delta_L)$. Unfortunately, this estimate could lead to inaccurate results. The minimum dispersion could represent a moment of (self-induced) congestion, which would cause two packets to wait in the same queue, effectively modifying their dispersion. Moreover, the presence of batching and multi-threading will also distort this value. An alternative approach is to calculate the mean or median dispersion value across the last N packets in the train since

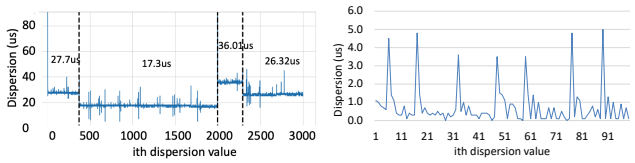


Figure 4: Effect of DVFS and packet batching on dispersion values, affecting capacity estimates. (a) Assuming that the last step in dispersion values corresponds to maximum frequency would be incorrect in this example due to the potential under-clocking. (b) Packet batching can cause spikes in the dispersion signature. Hence, choosing a probe length larger than the batch interval is important.

these packets are more likely to have observed the maximum frequency if the train itself has triggered it. In this case, $F(D) = mean(\delta_{L-N}, \delta_{L-N+1}, ..., \delta_L)$ and $\mathbb{C}_p$ will be $\mathbb{C}_p = 1/mean(\delta_{L-N}, \delta_{L-N+1}, ..., \delta_L)$. The number of packets, N, can be selected by finding the last step in the dispersion values using a step detection algorithm. This approach implicitly assumes that CPU frequency transitions from low to high and then remains there until the load subsides. However, we observed that in certain cases, frequency may transition back to a low frequency, as shown in Figure 4a.

4.2.3 Our approach in *NFTY*

To handle the challenges mentioned in the aforementioned section, *NFTY* first performs step detection [55] on the sequence of dispersion values $D = \delta_1, \delta_2, ..., \delta_L$ to detect the frequency transitions in the dispersion values. Second, it divides D into a set of segments S at the detected steps. Third, for each segment $\delta_{i\to j} \in S$ from δ_i to δ_j, it computes the mean dispersion value of the segment. We choose mean dispersion because it generalizes to the case of batching and multi-threading. After that, *NFTY* picks the minimum mean dispersion values as δ^*. The minimum mean dispersion will correspond to the segment of dispersion values when the frequency is maximum. Given segments $\delta_{i\to j} \in S$, we calculate δ^* as:

$$\delta^* = \min_{\delta_{i\to j} \in S} \frac{\sum_{n=i}^{j}(\delta_n)}{j-i+1} \tag{2}$$

For step detection, we consider algorithms that do not require the number of steps as input since the *NF* may transition to multiple steps, which will vary across deployments. Hence, we consider a different class of step detection algorithms [55], which uses penalty functions to separate noise from steps. Among these algorithms, we choose Binary Segmentation [13] as it is widely used [16, 22]. We use the Gaussian Kernel [23] as our cost function to measure variations in the data as it empirically works well. We use linear penalty [55] for step detection based on our empirical observations.

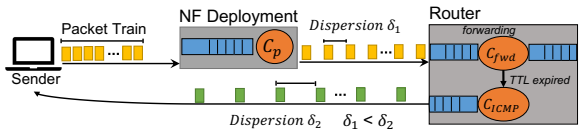


Figure 5: One-sided threat model: While a router can be used to echo the packets back to the receiver (triggering ICMP TTL time exceeded), its processing rate may change the spacing across packets (dispersion), effectively ruining the signature.

4.3 One-sided Control

NFTY, as we have described it so far, relies on the timestamped packets from the receiver that we assume that an attacker also has access to. In this subsection, we relax this assumption and explain how *NFTY* can be extended to a one-sided threat model as well. While prior works have proposed techniques that work with a one-sided threat model, they are not applicable to *NFCR* directly. For instance, Saroiu et al. [52] send TCP SYN requests aiming at triggering a TCP RST. An alternative approach from Carter et al. [15] sends an ICMP ECHO REQUEST aiming at triggering an ICMP REPLY. In both cases, the solution relies on the use of particular packet types that trigger protocol-specific behavior that might not trigger any processing from the *NF*.

4.3.1 High-level idea and challenges

NFTY exploits the *ICMP time exceeded* error response message sent by routers to hosts upon receiving an IP packet that has no time to reach its destination. Concretely, each IP packet carries a Time-To-Live (TTL) value in its header. As routers in the Internet path forward an IP packet, they decrement this value by one. If the TTL reaches zero (expires) before the packet reaches its destination, an *ICMP time exceeded* message is sent by the last router that decremented it to the sender. To trigger this behavior, the attacker: *(i)* sends packets towards a destination IP that is within the IP address space allocated to the targeted network;[1] and *(ii)* sets the TTL of the probing packets such that the TTL expires after the packet is processed by the *NF* but before the packet reaches its destination. The attacker can discover the appropriate TTL by using traceroute [7].[2] Traceroute is a standard measurement tool that relies on the TTL expiration mechanism and allows the sender to observe the path (the series of routers) that a packet takes to a destination. Importantly, TTL is part of a standard IP header and hence, it generalizes to various packet types. While one could, in principle, disable ICPM time-exceeded messages to protect from , that would deprive operators of a very useful debugging tool and would not necessarily protect the router's network as can use any router on the path between the NF and a host.

[1] IP address space allocation is public information.

[2] For example, the attacker can target the last router in the path that replies to traceroute.

While promising, relying on *ICMP time exceeded* to calculate *NF* dispersion is challenging. In fact, the use of ICMP replies has known limitations, which are well studied in the context of traceroute [17, 24]. We focus on three factors that are particularly impactful in our context, as they obfuscate the dispersion signals. These are related to the router processing time, contention at the router or potential rate-limiting of ICMP traffic:

1. *The router processing time might overwrite the NF dispersion.* In this case, the observed dispersion would correspond to the router's processing time *i.e.,* not the *NF*'s processing time. We illustrate this issue in Figure 5. This is not a problem in the case of two-sided, in which routers only forward the probing packets (instead of dropping them and generating a new packet).
2. *The router in which the TTL expires might be congested with regular traffic, thus adding random delays to ICMP replies*. In such cases, routers prioritize regular packet forwarding from control tasks, which are often best effort. The random delays added to the ICMP packets will make their dispersion useless for an attacker attempting *NFCR*.
3. *ICMP rate limiting may cause the routers to not respond at all.* Finally, an operator might rate-limit the ICMP traffic [2] for performance or security reasons. For example, in our experiments, we tried three routers in the university network and found that two of them rate-limited packets at five packets per second, as also observed by Ravaioli *et al.* [46] who found that 60% of the routers in the Internet implemented ICMP TTL exceeded rate limiting.

4.3.2 Our approach in *NFTY*

To effectively address the aforementioned limitations, we use two insights. First, while *NFTY* needs to send large trains to trigger frequency scaling, *NFTY* only needs the dispersion from a small number of packets to estimate $\mathbb{C}_p$. Second, to measure dispersion, *NFTY* does not need the arrival times of consecutive packets, instead *NFTY* can calculate the mean dispersion $\bar{\delta}$ between two packets p_i and p_j, which arrive at the receiver at t_i and t_j by $\delta = (t_j - t_i)/(j - i)$

Next, we explain how we use these insights to mitigate the effects of rate-limiting, router processing times, and router load. Consider that the attacker sends a packet train of length L at time τ^{send}. The attacker sets the TTL of packet p_i and some other non-consecutive packet p_j to expire at some router r. The packets arrive at the *NF*, which takes time t_p to process each packet. The first packet p_1 sees no queuing delay at the *NF*. The second packet is queued for time t_p and packet p_i is queued for time $(i-1)t_p$ because $i-1$ packets will be processed before the packet p_i. Then, the packet p_i arrives at the router, which will see that p_i has expired TTL, and it will generate an ICMP TTL time exceeded with processing delay t_r. Then, let τ_i^r be the time the reply for packet p_i arrives at the sender, and τ_j^r be the time the reply for packet p_j arrives at the sender.

If d_i is the total delay when packet p_i left the sender and its ICMP Reply arrived at the sender, then τ_i^r is:

$$\tau_i^r = \tau^{send} + d_i \quad \text{where } d_i = t_p + t_r + (i-1)t_p + c \tag{3}$$

c is for the constant transmission and propagation delay; t_p is the processing delay experienced by the packet p_i at the *NF*; t_r is the router processing delay in generating an ICMP reply; $(i-1)t_p$ is the queuing delay experienced by the packet. We choose packet p_j such that when it arrives at the router, packet p_i has already been processed. This ensures that packet p_j does not see any queuing at the router. Then for packet p_j, τ_j^r is given by:

$$\tau_j^r = \tau^{send} + d_j \quad \text{where } d_j = t_p + t_r + (j-1)t_p + c \tag{4}$$

At the sender, we can calculate the dispersion as follows:

$$\delta = \tau_j^r - \tau_i^r \tag{5}$$

Substituting Equation 3 and Equation 4 in Equation 5, we get:

$$\begin{aligned} \delta &= \tau^{send} + t_p + t_r + (j-1)t_p + c \\ &\quad - \tau^{send} - t_p - t_r - (i-1)t_p - c \\ \delta &= (j-i)t_p \end{aligned} \tag{6}$$

From Equation 6, the processing capacity $\mathbb{C}_p$ can be estimated as follows:

$$\mathbb{C}_p = 1/t_p = (j-i)/\delta \tag{7}$$

Having explained why we do not need to trigger an ICMP error in every packet of our probe, the next natural question is how to choose these packets. To decide that, we need to look at our constraints. We want to space the packets such that when packet p_j arrives at the router, packet p_i has already been processed. This requires an estimation of an *upper-bound* of router processing time. We can estimate processing time by sending packet pairs with the correct TTL. The spacing of the replies will give the router processing time. For example, in our experiments, we found that the router took 55us to process and generate ICMP replies.

Once we have an estimation of the router processing time, we can choose the packets such that when they arrive at the router, they are at least t_r apart. For example, for a router processing time of 100us, we can set the TTL after every 100 packets, assuming our send rate is 1Mpps. Concretely, for a router processing time of t_r, and the maximum capacity C_{max} that the attacker can send, we can set the TTL after a gap of g_{TTL} packets where g_{TTL} is chosen such that $g_{TTL} > t_r \times C_{max}$, $t_r \times C_{max}$ gives the number of packets that arrive at the router in time t_r. Observe that the *NF* processes packets before they arrive at the router. Thus, the actual duration between the packets will be much greater than t_r. For example, consider that we have an *NF*, which processes each packet in 5us, and a router takes 100us to generate an ICMP reply. If the sender sets the TTL of every 100th packet, then if the first packet arrives at the router at time t, the 101st packet will arrive at $((5us) + (100)(5us)) \times t$, which is much greater than 100us. Importantly, the estimation of the processing in the router need not be extremely accurate. In fact, we only need to make sure that the packets are spaced out more than the processing time.

4.4 End-to-end View

Next, we discuss how we can configure *NFTY* depending on various deployment settings and constraints. The key configuration parameter here is the ideal probe length for the attacker, *i.e.,* the minimum number of packets for a good estimation. This, in turn, depends on all the factors we have mentioned so far, including *NF*'s deployment, the receiver's existence, or hardware configuration. Figure 6 summarizes our insights into the various factors that play a role in this decision.

Starting from the bottom-most path in Figure 6, if the attacker is one-sided, meaning they only control the senders and rely on a router for receiving some form of reply, she would need to send $c_1 \times g_{TTL}$ where g_{TTL} is the gap between packets for which the attacker sets the TTL, and c_1 a constant which determines the number of TTL Exceeded packets the attacker will receive. Remember that in the one-sided case, to deal with routers with longer ICMP router processing times and with ICMP rate-limiting, *NFTY* spaces out the packets with the correct TTL. This gap determines the length of the probe. In our experiments, $30 \times g_{TTL}$, with g_{TTL} of 100 packets (total of 3000 packets) yields accurate capacity estimations. In the presence of DVFS, the attacker will again need to send packets in the order of thousands to first trigger DVFS to see the maximum capacity, as shown in Figure 7.

If the attacker is two-sided and the *NF* does not use DVFS (topmost branch), then the probe length will depend on the number of threads in the *NF* or/and the batch interval in case of packet batching. First, to capture the effect of multi-threading, the attacker will need to send packets $c \times k$ packets at the *NF* with k threads, where c is some constant. Intuitively, the value of c is proportional to the number of threads. Second, to handle the noise introduced by batching, attacker will need to send packets such that the total arrival time of packets at the receiver is larger than the batch interval. Batching is usually done in microseconds with values typically around 100us [36]. Hence, in this branch of the decision tree, a few hundred packets will give a good estimate, assuming that the number of threads is also in the hundreds.

If the attacker is two-sided and the *NF* uses DVFS, the packets needed to trigger DVFS will be in thousands and hence the number of threads and batching will not play a role in determining the probe length. Similarly, in the one-sided case, the number of packets typically needed would be in the

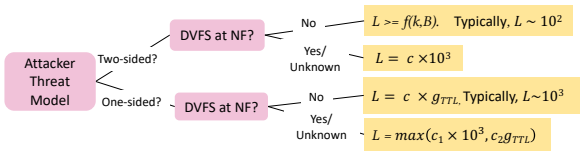


Figure 6: Probing an *NF* to estimate its capacity is challenging because of factors such as if the attacker controls nodes on both sides of the *NF* or not, if there is DVFS enabled in the *NF* deployment, batching of packets etc. Here, L is the probe length, k is the number of threads, B is the batch interval, g_{TTL} is the packet gap needed for the one-sided case, c, c_1, c_2 are constants.

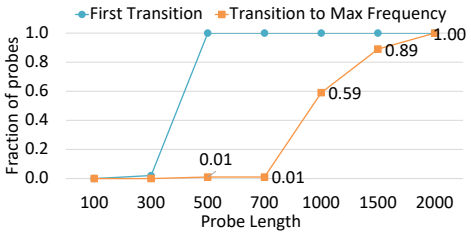


Figure 7: Poorly-selected train size may not trigger a transition to the maximum frequency; e.g., a train of 1000 packets has only 59% chances of observing the *maximum* frequency and 100% chances of observing the first transition to a higher frequency.

thousands. Hence, multi-threading and batching may also not play a role there.

Given this context, we distinguish two representative *NFTY* configurations based on the decision tree in Figure 6. The two configurations differ in generality and stealthiness.

NFTY-5K uses 5K packets to probe an *NF*, it will accurately estimate the capacity of *NF* with or without optimized deployments. Importantly, *NFTY-5K* is also effective with the one-sided threat model. As we show in Section 6, *NFTY-5K* can accurately estimate the capacity of diverse *NF* deployments within 10% error in the Internet in a two-sided and one-sided threat model. Moreover, *NFTY-5K* outperforms link-bandwidth estimation baselines by 30x.

NFTY-100 uses 100 packets to probe an *NF*. It will work for simple *NF* deployments, where DVFS is disabled. Due to its tiny network footprint *NFTY-100* is very stealthy. While *NFTY-100* is less accurate than *NFTY-5K*, it is still powerful enough to estimate the capacity of a commercial NF deployed in AWS. *NFTY-100* falls in the topmost branch of the decision tree.

Since *NFTY-5K* is strictly more accurate, we expect an attacker to use *NFTY-5K*, unless they have a strict budget or side-channel information about the NF deployment. We thoroughly evaluate both attacks in Section 6.

5 Sender and Receiver Setup

(maria: This might need to moved in 4? it is very weirdly placed) For our probing and estimation technique to work, we need a robust measurement infrastructure to get accurate, fine-grained, and noise-free dispersion measurements. To achieve this, we require:

Figure 8: Our testbed comprises three servers acting as the sender, the receiver and the *NF*. The DELAY_NF delays each packet by a predefined amount.

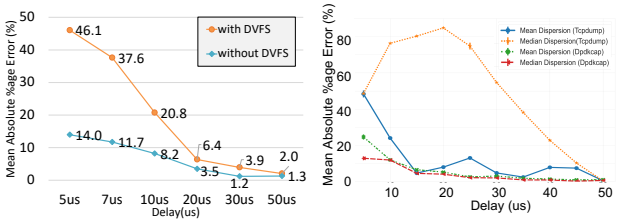


Figure 9: (a) Frequency scaling (DVFS) at the sender results in high error for lower processing times. Disabling frequency scaling at the sender significantly reduces this error. (b)TCPDump results in high error in the measured dispersion because of interrupt throttling.

Fast sending rate: The sending rate should be faster than the *NF* capacity $\mathbb{C}_p$ to ensure that packets are queued when they arrive at the *NF*.

Accurate timestamping: Packets should be timestamped at microsecond precision when they arrive at the receiver to capture the dispersion accurately.

We build a test setup (Figure 8) to systematically verify our sender and receiver setup. We create a dummy *NF*, namely DELAY_NF with a configurable processing time, implemented as a sleep operation for a configurable time window upon receiving a UDP packet. We implemented DELAY_NF using Click [32] and it serves as the ground truth for dispersion. For each experiment, we send packet trains of 10 packets and report the error in the mean dispersion. For packet generation, we evaluate tools such as Hping3, TCPReplay, and MoonGen. For capturing packets at the receiver, we test TCPDump and DPDKcap. We also explore certain sender (*e.g.,* DVFS [1]) and receiver side configurations (*e.g.,* RSS [4], interrupt throttling) that may affect the dispersion. Next, we present our findings which guide *NFTY* design.

***NFTY* turns off DVFS at the sender to send packets at a faster rate** We found that frequency scaling at the sender node can affect the send rate as also observed previously [40]. The Figure 9a shows the error in measured dispersion for different delay values of the DELAY_NF. At low CPU frequency, the sender is not able to send packets as fast. Hence, in the presence of frequency scaling at the sender, the error in measured dispersion can reach up to 46.1% for smaller delay values. Note that we observed a similar trend for kernel-based tools and MoonGen. Hence, we turn frequency scaling off

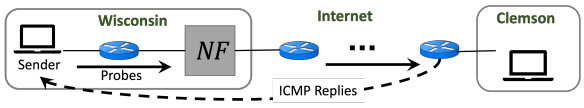


Figure 10: Our Internet setup forwards packets through the Internet after they have been processed by the *NF*. This is the worst-case scenario for the attacker (and *NFTY*) as the Internet noise can affect the dispersion signature of *NF*. In the one-sided experiments, we send probes to the Clemson node but they expire on path, causing ICMP replies to the sender(shown in dotted).

NF (μs)	Controlled	Internet	One-sided
SNORT-RL	67500	21000	21800
SUR-BL	142600	134640	143420
SUR-RL	200880	193750	225140
SUR-BL-mt	227720	224680	216000
SUR-RL-mt	336060	360460	309140

Table 1: The ground-truth processing capacity of our NFs measured in packets per second (pps) for our experimental setups. Processing capacity varies for different NFs and across deployments (*e.g.,* processors, NIC, configuration)

at the sender node. All bandwidth estimation tools do not consider this factor and hence may get affected as we show later in the evaluation section 6.

***NFTY* uses DPDKcap to avoid the negative effects of interrupt throttling on the timestamping of received packets.** On the receiver side, using DPDKcap reduces the noise in the measured dispersion and has up to 2X lower error compared to TCPDump as shown in Figure 9b. Essentially, TCPDump entails high errors because of interrupt throttling and delays in the kernel networking stack. While prior work [44, 57] tries to handle the effects of batching by estimating the batching interval and correcting for it, this is hard in the presence of adaptive interrupt coalescence [10]. DPDKcap, on the other hand, bypasses the kernel and the need for interrupts via userspace polling.

6 Evaluation

We evaluate the accuracy and the detectability of *NFTY* in two representative configurations, namely *NFTY-5K* and *NFTY-100*, which send 5K and 100 packets, respectively in controlled and Internet experiments. We find that *NFTY-5K* is up to 30x more accurately than a baseline sending the same number of packets per probe. The latter configuration, *i.e., NFTY-100* is accurate only on a subset of the NF deployments but is more stealthy. *NFTY-100* is up to 20x more accurate in controlled experiments compared to the equivalent baselines sending the same number of packets. Importantly, we find that *NFTY-100* can accurately estimate the capacity of a commercial *NF* in AWS within 7% of error. [We also evaluate the impact of each individual optimization in *NFTY* and find that step-detection and an optimized measurement infrastructure are crucial to *NFTY*'s accuracy.] (maria: Add something about the sensitivity analysis *e.g.,* say X and Y are the most important?)

6.1 Methodology

Probing baseline: (maria: This is not clear. please pass!) [Tried to change, please see if it is clearer now.] We evaluate our approach against a rate-based bandwidth-estimation technique, namely *SLoPS* in which the attacker uses binary search for *NFCR* [29] between 0-500Kpps. The attacker sends packets to the NF at an initial probing rate (250Kpps) and uses one-way delay of received packets to see if the probe rate (pps) caused congestion indicating that the rate was greater than the NF capacity. If so, it adjusts the probe rate accordingly using a binary search approach. *SLoPS* keep probing until the difference between minimum and maximum rate is 1000pps. To find an increasing trend in the relative one-way delay values, we use the technique implemented by Pathload [29]. Moreover, we use the same probe length for the baseline and for *NFTY*.

Estimation baselines: To evaluate the estimation technique used by *NFTY*, we compare with two baseline approaches adopted from the link bandwidth estimation literature *(i)* **mean-train** which takes the mean dispersion of the entire train [18]; and *(ii)* **median-train** which takes the median dispersion [35].

Network functions: We use five *NFs*. Both SNORT-RL and SUR-RL are realistic TCP SYN rate-limiting *NFs* which we implemented in Suricata [6] and Snort [5] respectively. Both *NFs* track the number of SYN packets per flow and drop the SYNs of flow that have exceeded a threshold. For SNORT-RL and SUR-RL, we send SYN packets. SUR-BL is a deny list containing rules provided by the emerging threats database [3]. We only send UDP traffic to SUR-BL to trigger the rules concerning UDP traffic. We implemented SUR-BL in Suricata [6]. SUR-BL-mt and SUR-RL-mt are multi-threaded versions of SUR-BL and SUR-RL respectively.

Ground-truth measurements: To measure the ground-truth processing capacity for our *NFs*, we send packets to a receiver node via each *NF* using Scapy *sendpfast* [48]. We do a binary search on the sending rate until we find the maximum rate that allows the input rate of the *NF* node, to equal its output rate (*i.e.,* the maximum sending rate without packet drops at the *NF*). We repeat this exercise five times and choose the median as the ground-truth. We show our results in Table 1. We measure the ground-truth capacity for each setup and *NF* separately, as it can be affected by hardware and software factors (*e.g.,* processor, NIC). We repeat each experiment 100 times and report the Median Absolute Percentage Error (MdAPE) in the estimated processing capacity compared to the ground truth (Table 1).

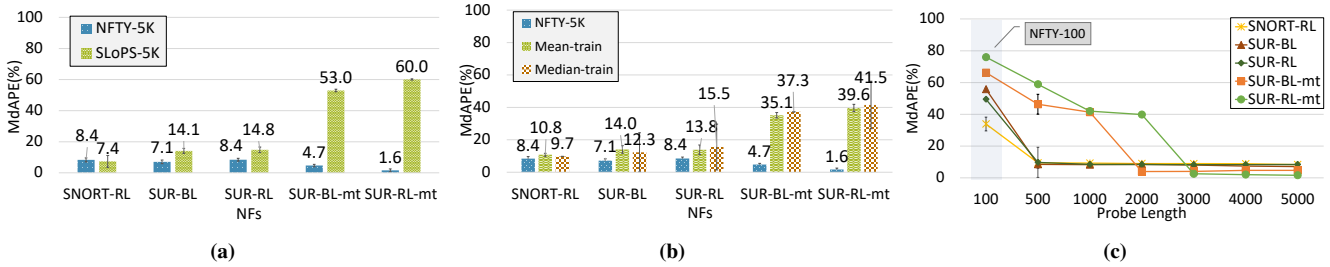


Figure 11: (a) *NFTY-5K* is up to 30x more accurate than the baseline in optimized *NF* deployments for the same probes length. (b) By applying step detection to the dispersion values, the MdAPE of *NFTY-5K* is reduced by up to 25x. (maria: add to label step-detection?) (c) As we increase the probe length, the accuracy of *NFTY* improves.

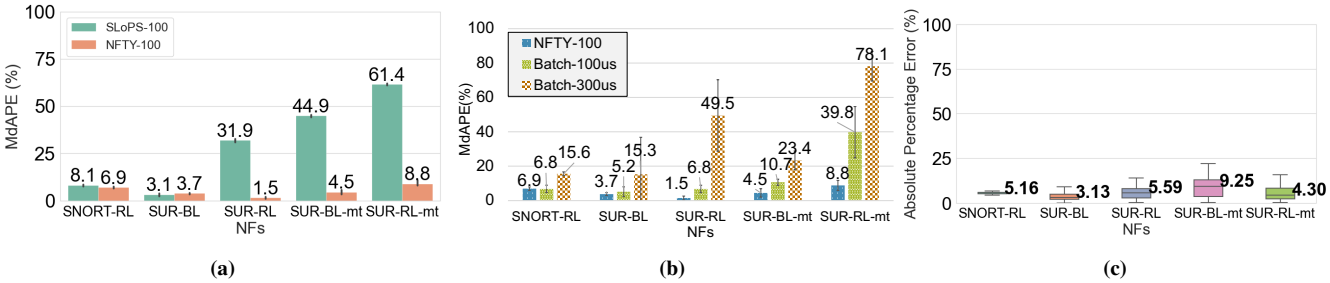


Figure 12: (a) *NFTY-100* results in 3% to 9% error in the estimated processing capacity with the optimized measurement infrastructure. *SLoPS*-100 results in high error (up to 61%) for faster *NFs* as the sender cannot send packets faster than the *NF* because of DVFS at sender. (b) When we use *NFTY-100* in an optimized measurement infrastructure, the MdAPE is reduced by up to 33x. The improvement is significant for the larger batch interval (batch-300us) as compared to the smaller interval (batch-100us). (c) *NFTY-100*'s error remains below 10% in the internet. The spread is a bit higher than the controlled experiments for very fast *NFs* due to the noise added by the Internet.

6.2 Controlled experiments

We give an overview of how *NFTY-5K* and *NFTY-100* perform against baselines. We focus on the two-sided model because our baseline does not work for the one-sided threat model. We use a controlled lab experiment to investigate how *NFTY* design decisions contribute to its precision.

Controlled setup: We use three nodes in CloudLab [20], a sender, a receiver, and an *NF* connected such that the sender communicates to the receiver via the *NF* node. All Intel Haswell EP nodes with two 10-core CPUs with 2.60 GHz frequency.

***NFTY-5K* outperforms probing baselines by 2x-30x while sending less packets in total.** Figure 11a shows the MdAPE in the estimated capacity by *NFTY-5K* and *SLoPS*. *NFTY-5K* results in 2%-9% error for the five *NFs* and outperforms the baseline. The improvement is more prominent in the faster *NFs*, compared to SNORT-RL which is slower (see Table 1). The baseline also ends up sending 9x more packets (median in NFs) than *NFTY-5K*, due to its iterative probing. Still, the baseline's poor accuracy is due to its obliviousness to the fluctuation of one-way delays caused by the NF's optimized deployment (*i.e.,* frequency transitions). Moreover, for multi-threaded NFs (last two bars), the baseline ends up triggering only a single thread because both the multi-threaded *NFs* assign packets to threads on the basis of flow. Importantly, both techniques use the optimized measurement infrastructure we describe in §5. The baseline precision further degrades with an unoptimized measurement infrastructure, as we show later (Figure 12a).

Step-detection improves *NFTY-5K* accuracy by up to 25x To investigate the benefit of step detection, we compare *NFTY-5K* with the estimation baselines, mean-train and median-train. We summarize our results in Figure 11b. We observe that the reduction in MdAPE is significant. For SNORT-RL the improvement over the baseline is less compared to the other NFs because SNORT-RL changes to maximum frequency sooner than the other *NFs*. In effect, mean-train is closer to the one corresponding to the maximum frequency. median-train precision degrades further for multi-threaded *NFs*, because of the fluctuations in queueing delays for subsequent packets it causes.

Longer probe lengths are critical for NFs with DVFS. Figure 11c shows the effect of different probe lengths on *NFTY*'s error in the presence of DVFS. Longer trains for *NFTY* result in low errors (MdAPE) because they always trigger the transition to the maximum frequency. In contrast, small trains never trigger the transition to maximum frequency. Thus, if the attacker were to perform *NFCR* with *NFTY-100* in the presence of DVFS, she would have a high error. The frequency transi-

	TCP SYN				UDP			
NFTY-5K	✗	✗	✓	✗	✗	✓	✓	✗
SLoPS-5K	✓	✓	✓	✗	✓	✓	✓	✗
NFTY-100	✗	✗	✗	✗	✗	✗	✗	✗
SLoPS-100	✗	✗	✗	✗	✗	✓	✗	✗

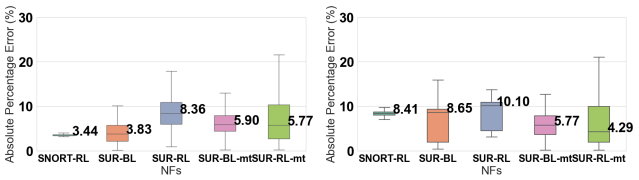
Table 2: *NFTY* is less detectable than *SLoPS* while being more accurate (maria: What are the columns?)

tions happen sooner for slow *NFs* as compared to the faster ones. Note that all *NFs* show a transition at length of 500 packets.

***NFTY-100* accurately estimates capacity within 9% of error.** Since we have shown that *NFTY-100* cannot estimate the capacity of NFs whose deployments are optimized with DVFS (Figure11c), we evaluate *NFTY-100* for simpler *NF* deployments. We find that *NFTY-100* estimates the capacity of these with 2% to 9% error as shown in Figure 12a. We compare *NFTY-100* with *SLoPS*-100. For *NFTY-100*, we turn DVFS off at sender, as with small number of packets, its effect becomes significant. For the baseline, DVFS is not off at the sender. We observe that the error and the spread for *SLoPS*-100 increase up to 61%. Specifically, for *SLoPS*-100 the error is huge for *NFs* (SUR-RL to SUR-RL-mt) where the attacker cannot send at rates higher than the *NF* processing speed. This highlights the importance of our sender-side optimizations. Hence, if an attacker were to use *SLoPS*-100 without our sender-side optimization, she would incur a huge error.

***NFTY*'s measurement optimizations improve the accuracy of *NFTY-100* by up to 33x** In the previous experiment, we already hinted that the most impactful factor in *NFTY-100* accuracy is the measurement infrastructure, as we discuss in §5. To verify that we plot the overall improvements caused by our measurement optimizations in Figure 12b . For the unoptimized measurement infrastructure, DVFS is enabled at the sender, and we show two receiver configurations, one with a batch interval of 100us and the other with a batch interval of 300us. For the optimized measurement infrastructure, DVFS is disabled at the sender, and there is no packet batching. As shown in Figure 12b, our measurement-infrastructure optimizations in *NFTY-100*, reduce its MdAPE by upto 33x.

***NFTY* is less detectable than *SLoPS* while being more accurate** To evaluate the detectability of *NFTY*, we use the ruleset of known commercial firewalls Palo Alto [], Juniper [], Fortinet [], and Snort community rules. We implement these rulesets in Suricata, and record if these rulesets detect *NFTY* and *SLoPS*. Table 2 summarizes our results for TCP SYN traffic processed by SUR-RL, SUR-RL-mt, and SNORT-RL and UDP traffic processed by SUR-BL, and SUR-BL-mt. [A tick indicates that the firewall was able to detect the attack. A cross indicates that the attack is not detected. Overall, *NFTY* is less detectable than the baseline.] (maria: Explain ticks)



(a) **(b)**

Figure 13: (a) *NFTY-5K* results in 4% to 9% MdAPE in the Internet with optimized *NF* deployment (with DVFS). The spread in error increases with faster *NFs*. (b) *NFTY* results in 4%-10% MdAPE with one-sided control. Our spread is a bit high for fast *NFs* compared to SNORT-RL, *i.e.,* the slowest *NF*.

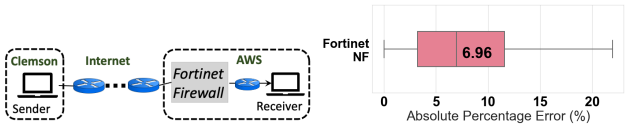
6.3 Internet-2 and One-sided Experiments

So far we only evaluated *NFTY* in pristine network conditions. Having shown that *NFTY* outperforms baselines by orders of magnitude, in this subsection we aim to verify whether *NFTY* is usable in more realistic settings *e.g.,* on the Internet or in cases where the attacker only controls the sender (single-sided) and thus relies on a router.

Internet setup: Our sender and the *NF* node run in CloudLab Wisconsin, and our receiver node runs in Clemson. As we illustrate in Figure 10, the packets from the *NF* to the receiver go through the Internet. We stress that this is the worst-case scenario for the attacker. Indeed, if the packets in a probe see congestion in the Internet before reaching the *NF*, the random delay/noise from the Internet would not affect the *NF*-induced dispersion. We run this experiment at different times of the day to capture the effect of varying congestion in the Internet. The sender and the receiver use the same nodes as in the controlled environment. The receiver uses Intel Haswell E5-2683 v3 nodes with 14-core CPUs and 2.00 GHz frequency.

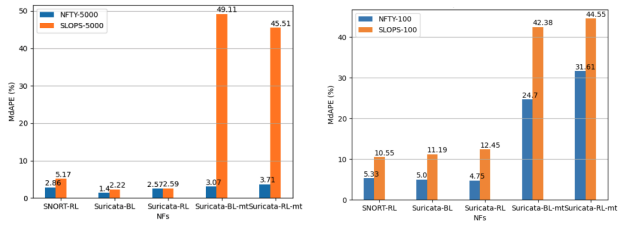
Both *NFTY-100* and *NFTY-5K* remain within 10% error in the Internet. We evaluate *NFTY-100* (without DVFS at the *NF*) in the Internet to show that *NFTY* can accurately estimate the capacity even in the Internet using only a small number of packets. We summarize our results in Figure 12c. *NFTY-100* can accurately estimate *NF*'s capacity within 10% error. The spread in error values for faster *NFs* is higher than in controlled experiments. We also evaluate *NFTY-5K* (with DVFS at the *NF*) to show that even with added noise from the Internet, shown in Figure 13a the *NFTY-5K*'s estimation using step detection works well. *NFTY-5K* has a higher spread in MdAPE in the Internet compared to the controlled environment. However, the MdAPE in the Internet for *NFTY-5K* also remains within 9% error.

One-sided setup: We illustrate the setup of this experiment in Figure 10. The sender and the *NF* are in the Wisconsin cluster and they send packets to a public server in the Clemson cluster. The TTL is set such that it expires in 4 hops after being



(a) (b)

Figure 14: (a) In the AWS experiments, the sender probes the *NF* from the Internet. (b) *NFTY-100* results in 7% MdAPE while probing a commercial *NF* Fortinet next generation firewall [21]. The spread is high because here, the receiver and the NF are VMs and may induce extra noise.



(a) (b)

Figure 15: (a) Comparing SLoPS-5K and NFTY-5K in internet experiments with an optimized sender and unoptimized *NF*. (b) Same setting as (a), but now comparing NFTY-100 and SLoPS-100.

processed by the NF. We choose g_{TTL} (*i.e.,* the gap in packets for which we set the correct TTL) to be 100 packets.

***NFTY-5K* is usable under the one-sided threat model.** We summarize our results in Figure 13b. For the experiments with one-sided threat model, *NFTY-5K*'s MdAPE remains around 10%. Faster *NFs* such as SUR-RL-mt show a higher spread as in the case of two-sided Internet experiments. This shows that indeed, *NFTY* with one-sided control is feasible and can result in reasonable estimation of the processing capacity of an *NF*.

6.4 Internet Experiments

Now we evaluate NFTY outside of the Internet2 setting which had packets sent only between Cloudlab machines. In these experiments, we provide an even more realistic setting for an attacker operating in a two sided model where they control both the sender and receiver.

Internet setup: Our sender and the NF run in Cloudlab Wisconsin, and our receiver node runs in Vultr Los-Angeles. Similar to figure 6, as the packets travel from the Cloudlab machines to the Vultr machine, they traverse the Internet. Unlike in the Internet2 experiments, the packets are no longer travelling through the isolated environment that the Cloudlab network services provide. We run these experiments with an unoptimized NF (DVFS off) and an optimized sender (DVFS off) and at different times of day to capture any variance caused by time of day. We perform these experiments with an unoptimized NF to see the viability of NFTY in the internet setting. The sender and NF use the same nodes as the previous experiments, and the receiver uses an Intel E-2288G processor with 8-core CPUs and 4.70 GHz frequency. Our experiments utilize the 2-sided method.

NFTY-5K remains within 8% error in the Internet. Figure 15a shows that the MdAPE for NFTY-5k in this setting never exceeded 5% for all *NFs*. For all 100 experiments for each NF, the calculated error never exceeded 8%. Thus, NFTY-5K achieves very low error rates even in the Internet environment such that we have DFVS disabled at the *NF*. We can also infer that, due to the closeness of the MdAPE and maximum error, that there is little spread and variance of NFTY-5K in the Internet. Thus, NFTY-5K proves itself as a reliable method where noise and delays may be prevalent. Like our previous experiments, we also see a general trend where the faster *NFs* have larger MdAPE results due to more noise in the dispersion values when the packets are sent across the Internet.

NFTY-100 results in less than 10% MdAPE for single-threaded NFs. Figure 15b indicates that NFTY-100 achieves low MdAPE values for the single-threaded *NFs*: SNORT-RL, Suricata-RL, and Suricata-BL. None of these MdAPE values exceeded 6%, demonstrating the high accuracy that NFTY-100 can achieve in the Internet setting with such a small amount of packets. Compared to the spread of the Internet2 experiments, NFTY-100 has a greater spread (not shown in figure 15b) due to the added noise of the Internet. For the multi-threaded *NFs*, there are two queues that packets are processed in, and so each queue processes only 50 packets each. With so few dispersion values for each queue and the noise of the Internet prevalent at the beginning of a probe, NFTY produces an inaccurate estimate. Figure 15a indicates that using 5000 packets instead for NFTY produces far more accurate results that outweighs the noise of the Internet that is prevalent in the first 100 packets of the probe.

Both NFTY-100 and NFTY-5K consistently outperform SLoPS. When comparing performance of NFTY to SLoPS, it has a smaller MdAPE for all NFs for both techniques. As seen in the results for the controlled environment and Internet2, SLoPS performs poorly for the multi-threaded *NFs*. This is because the threads are assigned to based on flow, and SLoPS sends all packets to only one thread, thus predicts the processing capacity of only one queue. Although NFTY-100 has reduced accuracy for the multi-threaded *NFs*, it still outperforms SLoPS. When comparing performance for the single-threaded *NFs*, NFTY-100 cuts the MdAPE by approximately 50%. For NFTY-5K, it's performance is similar to SLoPS' for the single-threaded *NFs* but reduces MdAPE by at least 90% for the multi-threaded *NFs*.

6.5 Case Study: *NFTY* against Commercial NF in the Cloud

Having tested *NFTY* in the lab and in the Internet we aim to investigate whether our results generalize beyond open-access *NFs* and in-house deployments. To this end, we use

NFTY to calculate the capacity of the Fortinet next-generation firewall[21] which is a closed-source and commercial NF.

Cloud setup: We deploy the Fortinet firewall in the AWS US East cluster on an AWS c4.large instance type in front of a private subnet which was the recommended instance type by the vendor. Thus, all traffic entering the subnet passes through the firewall as we show in Figure14a. We have configured the firewall to use its default attack detection profiles for all UDP traffic. We find the ground truth processing capacity to be 62738 pps. [We measure the ground truth using a local node within AWS by flooding the NF and measuring the received rate at the receiver. We found that when we flood the NF, the receiver constantly receives packets at a rate of 62738 pps, and the remaining packets are dropped at the NF.] We use our Clemson Cloudlab node to probe the firewall using UDP traffic to implement *NFTY-100*. Observe that traffic from Clemson to the AWS nodes is forwarded through public Internet, thus is subject to cross-traffic.

We find that *NFTY-100* can accurately estimate the processing capacity of the Fortinet firewall within 7% error. The spread in the error is high, which might be due to the the noise from use of a VM of either *NF* or the receiver node 14b. This shows that given an unknown NF with unknown deployment, *NFTY* can accurately estimate its processing capacity.

(maria: How do we measure ground truth here? Did we 'get caught'?)

7 Countermeasures

In this section, we present several countermeasures against an *NFCR* attack, which essentially try to either conceal the dispersion values or the resources available at the *NF*. We evaluate them for their effectiveness and the added overhead they impose.

Adding a random delay: A network operator could deliberately add a random delay to some packets, to affect the measured dispersion and thus obfuscate the real *NF* processing delay. While intuitive and simple, its not clear whether this countermeasure will work in practice despite adding overhead to legitimate traffic since adding random delays might still preserve the average dispersion, making an *NFCR* attack feasible.

Additional packet batching:: Rather than releasing packets as they arrive, an *NF* could buffer and release them in batches to change the dispersion signature, and affect capacity estimation. This countermeasure's effectiveness is affected by the buffer size, while also adding latency for legitimate flows.

Reorder using the multiple-queues: Another countermeasure is to leverage multiple queues on NICs to trigger packet reordering. Concretely, by forcing packets of the same flow to be sprayed across different queues the attacker's packets would arrive reordered, effectively disrupting the dispersion signature. While this countermeasure would not increase the one-way delay, it will affect the TCP performance which is known to suffer from reordering.

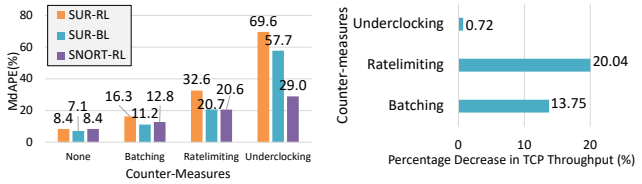


Figure 16: (a) Under-clocking is the most effective countermeasure. It would increase *NFTY*'s MAPE by 3x to 7x. (b)The countermeasures impose overhead on legitimate traffic. For example, under-clocking would decrease the throughput if a TCP flow by 0.72%.

Rate-limiting: Another alternative is to make the *NF* appear to have lower capacity, by rate-limiting packets from the same flow or IP source for example. While an attacker mitigates this countermeasure by using multiple IPs, this would again require her to deal with packet re-orderings and also require keeping per-flow state which can be impractical. This countermeasure will have low overhead, as a single flow should not use *even instantly* the full capacity of the *NF*.

Under-clocking the *NF*: An *NF* can set the parameters of DVFS governors to keep at a lower frequency for longer and switch to higher frequencies after more sustained loads. Recall that *NFTY* sends a few thousand packets to trigger DVFS, and with this countermeasure, an attacker would need to send many more packets making them detectable. However, delaying frequency scaling would also process legitimate traffic slower.

Countermeasures evaluation: We investigate the trade-off between effectiveness and overhead of these countermeasures. To do so, we measure the accuracy of *NFTY* in estimating the capacity of three realistic *NFs*, with various countermeasures deployed. To characterize overhead, we measure the loss in TCP throughput for a single Iperf [54] flow. We perform 10 runs with each countermeasure and report the MdAPE and throughput reduction.

We simulate three countermeasures, *excessive batching* with a batch interval of 300μs, *rate-limiting* by 20%, and *under-clocking*. Batching aims at blurring the measured delay between packets (dispersion); rate-limiting permanently obfuscates the *NF*'s true capacity, while under-clocking hides it for some time. We configure a batching interval of 500us, rate-limiting per flow to 20% of the maximum bandwidth, and under-clocking by 100ms.

Figures 16a, and 16b summarize our results. We observe that under-clocking is the most effective countermeasure as it increases *NFTY* MdAPE error by 3 to 5 times, while also incurring the least overhead since the low-frequency setting is only applicable for a short period.Note that, this result is subject to our chosen overhead metric. Intuitively, the overhead of under-clocking would have been higher if we used another metric *e.g.,* tail flow completion time.

Accurately evaluating the overhead of each trade-off is beyond the scope of our paper. Instead, this section provides various practical countermeasures that an operator could deploy based on their goals and the importance of their *NF*.

8 Related Work

We have already discussed the most closely related link-bandwidth estimation literature. In this section, we discuss other work related to *NFCR*.

Reconnaissance and prevention: Prior work in network reconnaissance tries to infer the topology of the network to construct attacks [30]. Similarly, Samak et al., [51] infer rules of a remote firewall by sending carefully crafted probes. Ramamurthy et al. [45] uses mini flash crowds to estimate various bottlenecks such as access bandwidth, CPU utilization, and memory usage of a server. Salehin *et al.* [49] aims at estimating the delay in the networking stack of a server by sending packet probes. These works do not perform reconnaissance for *NF* capacity and are thus orthogonal to our work. In addition to reconnaissance techniques, many prior works aim at protecting against reconnaissance attacks [11, 28, 39]. For example, NetHide [39] uses virtual topologies to deceive reconnaissance attackers.

***NF* capacity modeling:** Multiple works aim at modeling *NFs* for various reasons such as predicting memory impact on contention [37], generating adversarial workloads to trigger slow execution paths [43] or predicting the performance [25]. Such works are complementary to *NFCR* and can be used to further optimize inference.

9 Discussion

NFTY* measures the bottleneck *NF Note that in the case where there are multiple *NFs* chained in a given network, *NFTY* will only measure the packet processing capacity of the bottleneck *NFTY*. To target a particular *NF*, *NFTY* would need one hop access to the *NF* in the case that the *NF* of interest is not the bottleneck *NF*. ***NFTY* does not require the location of *NF*** We propose *NFTY* to estimate the packet processing capacity of an *NF* deployed in a target network. Note that we do not know the location of the *NF*. When *NFTY* is run against a target network, it will automatically measure the packet processing capacity of the bottleneck *NF* in the target network for a given packet type. Hence, *NFTY* does not require explicit information about the location of the *NF*.

NFTY* does not need the exact packet types of *NF Since, the target of *NFTY* is to measure the processing capacity of bottlecneck *NFs* for a given packet type, deployed in a target network, *NFTY* can use popular coarse grained packet types *e.g.,* TCP SYN, UDP, DNS etc., to probe for packet processing capacity. Finding packet types that would lead to slow execution paths is an interesting future work direction.

***NFTY* assumes links are not bottleneck** Between the attacker and the *NF*, *NFTY* assumes that links are not the bottleneck, which would typically be true as *NFs* do more computation than regular routers doing the forwarding. Note that for link bandwidth estimation techniques, they assume the alternate thing is that *NFs* are not the bottleneck.

***NFTY* does not work for *NFs* that elastically scale.** The current implementation of *NFTY* does not work for *NFs* that scale elastically with traffic load. Extending *NFTY* to handle this scenario is possible, however, it will make *NFTY* more noticeable, as it may need to trigger the elastic scaling. This in theory is very similar to DVFS, however, the number of packets or the scale is very different. Hence, extending *NFTY* to such *NFs* while remaining stealthy is another interesting future work direction.

***NFTY* does not work for *NFs* whose processing capacity change with packet history** We scope *NFTY* as a first step towards measuring the processing capacity of bottleneck *NFs* remotely. In the future, *NFTY* should be extended to include such stateful *NFs*. To measure such an *NF*, *NFTY* would need to send multiple probes to figure out some part of the black box state machine of such an *NF*.

***NFTY* experiences trade-off between accuracy and detectability** We evaluate *NFTY* with two configurations. *NFTY-100* is less detectable but does not measure accurately in all cases. *NFTY-5K* is more visible but measures accurately in the cases we evaluated. We do not claim that *NFTY-5K* will be enough to detect accurately in all cases. In the end, it depends on the attacker's budget. Based on the attacker budget, an attacker may measure accurately but get detected or vice versa.

10 Conclusions

This paper presents the first formulation and feasibility analysis of the Network Function Capacity Reconnaissance (*NFCR*) problem. While anecdotal data suggests that attackers have used some form of *NFCR* to scout the resources of their victims prior to actual DDoS attacks, such attempts are not documented, leaving network operators unable to detect or mitigate them. To bridge this gap, we put ourselves in the shoes of an attacker and investigate various probing strategies, measurement infrastructures, and threat models. In doing so, we constructed a practical tool catered to the *NFCR* problem, namely *NFTY*. We identify and evaluate two representative *NFTY* configurations, namely *NFTY-100* and *NFTY-5K*. *NFTY-100* can accurately estimate the capacity of simple NF deployments while being extremely stealthy. *NFTY-5K* is more accurate under diverse *NF* deployments but has a larger network footprint. While less accurate than *NFTY-5K*, *NFTY-100* is still powerful enough to estimate the capacity of a commercial NF deployed in AWS within 7% error. Finally, we present and evaluate countermeasures against *NFTY*.

Availability

We will make all our code public upon publication.

References

[1] CPU frequency scaling - ArchWiki. https://wiki.archlinux.org/title/CPU_frequency_scaling.

[2] ICMP rate-limiting. https://techhub.hpe.com/eginfolib/networking/docs/switches/RA/15-18/5998-8161_ra_2620_mcg/content/ch05s03.html.

[3] Proofpoint Emerging Threats Rules. https://rules.emergingthreats.net/open/suricata/rules/.

[4] Receive Side Scaling on Intel® Network Adapters. https://www.intel.com/content/www/us/en/support/articles/000006703/network-and-i-o/ethernet-products.html.

[5] Snort - Network Intrusion Detection & Prevention System. https://www.snort.org/.

[6] Suricata. https://suricata.io/.

[7] Traceroute(8) - Linux man page. https://linux.die.net/man/8/traceroute.

[8] Adaptive ddos attacks warrant next gen defense. https://informationsecuritybuzz.com/infosec-news/adaptive-ddos-attacks-warrant-next-gen-defense/, 2014.

[9] Azure marketplace. https://azuremarketplace.microsoft.com, 2021.

[10] Interrupt Coalescing . https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/aix/7.2?topic=options-interrupt-coalescing, Oct. 2021.

[11] ACHLEITNER, S., LA PORTA, T. F., MCDANIEL, P., SUGRIM, S., KRISHNAMURTHY, S. V., AND CHADHA, R. Deceiving network reconnaissance using sdn-based virtual topologies. *IEEE Transactions on Network and Service Management 14*, 4 (2017), 1098–1112.

[12] ALLMAN, M. Measuring end-to-end bulk transfer capacity. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement* (2001), pp. 139–143.

[13] BAI, J. Estimating multiple breaks one at a time. *Econometric theory 13*, 3 (1997), 315–352.

[14] BOLOT, J.-C. Characterizing end-to-end packet delay and loss in the internet. *Journal of High Speed Networks 2*, 3 (1993), 305–323.

[15] CARTER, R. L., AND CROVELLA, M. E. Measuring bottleneck link speed in packet-switched networks. *Performance evaluation 27* (1996), 297–318.

[16] CHEN, J., AND GUPTA, A. K. Parametric statistical change point analysis: with applications to genetics, medicine, and finance.

[17] DETAL, G., HESMANS, B., BONAVENTURE, O., VANAUBEL, Y., AND DONNET, B. Revealing middlebox interference with tracebox. In *Proceedings of the 2013 conference on Internet measurement conference* (2013), pp. 1–8.

[18] DOVROLIS, C., RAMANATHAN, P., AND MOORE, D. What do packet dispersion techniques measure? In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)* (2001), vol. 2, IEEE, pp. 905–914.

[19] DOVROLIS, C., RAMANATHAN, P., AND MOORE, D. Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Transactions On Networking 12*, 6 (2004), 963–977.

[20] DUPLYAKIN, D., RICCI, R., MARICQ, A., WONG, G., DUERIG, J., EIDE, E., STOLLER, L., HIBLER, M., JOHNSON, D., WEBB, K., ET AL. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)* (2019), pp. 1–14.

[21] FORTINET. Fortinet cloud security amazon web services (aws).

[22] FRYZLEWICZ, P. Wild binary segmentation for multiple change-point detection. *The Annals of Statistics 42*, 6 (2014), 2243–2281.

[23] GARREAU, D., AND ARLOT, S. Consistent change-point detection with kernels. *Electronic Journal of Statistics 12*, 2 (2018), 4440–4486.

[24] GUNES, M. H., AND SARAC, K. Resolving anonymous routers in internet topology measurement studies. In *IEEE INFOCOM 2008-The 27th Conference on Computer Communications* (2008), IEEE, pp. 1076–1084.

[25] IYER, R., PEDROSA, L., ZAOSTROVNYKH, A., PIRELLI, S., ARGYRAKI, K., AND CANDEA, G. Performance contracts for software network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), pp. 517–530.

[26] JACOBSON, V. Congestion avoidance and control/van jacobson. SIGCOMM.

[27] JACOBSON, V. Pathchar: A tool to infer characteristics of internet paths, 1997.

[28] JAFARIAN, J. H., AL-SHAER, E., AND DUAN, Q. An effective address mutation approach for disrupting reconnaissance attacks. *IEEE Transactions on Information Forensics and Security 10*, 12 (2015), 2562–2577.

[29] JAIN, M., AND DOVROLIS, C. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with tcp throughput. *ACM SIGCOMM Computer Communication Review 32*, 4 (2002), 295–308.

[30] KANG, M. S., LEE, S. B., AND GLIGOR, V. D. The crossfire attack. In *2013 IEEE symposium on security and privacy* (2013), IEEE, pp. 127–141.

[31] KAPOOR, R., CHEN, L.-J., LAO, L., GERLA, M., AND SANADIDI, M. Y. Capprobe: A simple and accurate capacity estimation technique. *ACM SIGCOMM Computer Communication Review 34*, 4 (2004), 67–78.

[32] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. *ACM Transactions on Computer Systems (TOCS) 18*, 3 (2000), 263–297.

[33] LAI, K., AND BAKER, M. Measuring bandwidth. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)* (1999), vol. 1, IEEE, pp. 235–245.

[34] LAI, K., AND BAKER, M. Nettimer: A tool for measuring bottleneck link bandwidth. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS 01)* (2001).

[35] LI, M., CLAYPOOL, M., AND KINICKI, R. Wbest: A bandwidth estimation tool for ieee 802.11 wireless networks. In *2008 33rd IEEE Conference on Local Computer Networks (LCN)* (2008), IEEE, pp. 374–381.

[36] LIANG, K. Improve network performance by setting per-queue interrupt moderation in linux*, May 2017.

[37] MANOUSIS, A., SHARMA, R. A., SEKAR, V., AND SHERRY, J. Contention-aware performance prediction for virtualized network functions. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 270–282.

[38] MATHIS, M. Diagnosing internet congestion with a transport layer performance tool. *Proc. INET'96, June* (1996).

[39] MEIER, R., TSANKOV, P., LENDERS, V., VANBEVER, L., AND VECHEV, M. {NetHide}: Secure and practical network topology obfuscation. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 693–709.

[40] NILSSON, J. Cost-effective packet generation for performance evaluation of network equipment, 2016.

[41] PANJWANI, S., TAN, S., JARRIN, K. M., AND CUKIER, M. An experimental evaluation to determine if port scans are precursors to an attack. In *2005 International Conference on Dependable Systems and Networks (DSN'05)* (2005), IEEE, pp. 602–611.

[42] PAXSON, V. End-to-end internet packet dynamics. In *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication* (1997), pp. 139–152.

[43] PEDROSA, L., IYER, R., ZAOSTROVNYKH, A., FIETZ, J., AND ARGYRAKI, K. Automated synthesis of adversarial workloads for network functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 372–385.

[44] PRASAD, R., JAIN, M., AND DOVROLIS, C. Effects of interrupt coalescence on network measurements. In *International Workshop on Passive and Active Network Measurement* (2004), Springer, pp. 247–256.

[45] RAMAMURTHY, P., SEKAR, V., AKELLA, A., KRISHNAMURTHY, B., AND SHAIKH, A. Remote profiling of resource constraints of web servers using mini-flash crowds.

[46] RAVAIOLI, R., URVOY-KELLER, G., AND BARAKAT, C. Characterizing icmp rate limitation on routers. In *2015 IEEE International Conference on Communications (ICC)* (2015), IEEE, pp. 6043–6049.

[47] RIBEIRO, V. J., RIEDI, R. H., BARANIUK, R. G., NAVRATIL, J., AND COTTRELL, L. pathchirp: Efficient available bandwidth estimation for network paths. In *Passive and active measurement workshop* (2003).

[48] ROHITH, R., MOHARIR, M., SHOBHA, G., ET AL. Scapy-a powerful interactive packet manipulation program. In *2018 international conference on networking, embedded and wireless systems (ICNEWS)* (2018), IEEE, pp. 1–5.

[49] SALEHIN, K. M., ROJAS-CESSA, R., LIN, C.-B., DONG, Z., AND KIJKANJANARAT, T. Scheme to measure packet processing time of a remote host through estimation of end-link capacity. *IEEE Transactions on Computers 64*, 1 (2013), 205–218.

[50] SALIM, J. H., OLSSON, R., AND KUZNETSOV, A. Beyond softnet. In *5th Annual Linux Showcase & Conference (ALS 01)* (2001).

[51] SAMAK, T., EL-ATAWY, A., AND AL-SHAER, E. Firecracker: A framework for inferring firewall policies using smart probing. In *2007 IEEE International Conference on Network Protocols* (2007), IEEE, pp. 294–303.

[52] SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. D. Sprobe: A fast technique for measuring bottleneck bandwidth in uncooperative environments. In *IEEE INFOCOM* (2002), p. 1.

[53] SCHNEIER, B. Someone is learning how to take down the internet, Sep 2016.

[54] TIRUMALA, A. Iperf: The tcp/udp bandwidth measurement tool. *http://dast. nlanr. net/Projects/Iperf/* (1999).

[55] TRUONG, C., OUDRE, L., AND VAYATIS, N. Selective review of offline change point detection methods. *Signal Processing 167* (2020), 107299.

[56] Verisign q2 2016 ddos trends: Layer 7 ddos attacks a growing trend. https://blog.verisign.com/security/verisign-q2-2016-ddos-trends-layer-7-ddos-attacks-a-growing-trend/, Aug 2016.

[57] YIN, Q., AND KAUR, J. Can machine learning benefit bandwidth estimation at ultra-high speeds? In *International Conference on Passive and Active Network Measurement* (2016), Springer, pp. 397–411.