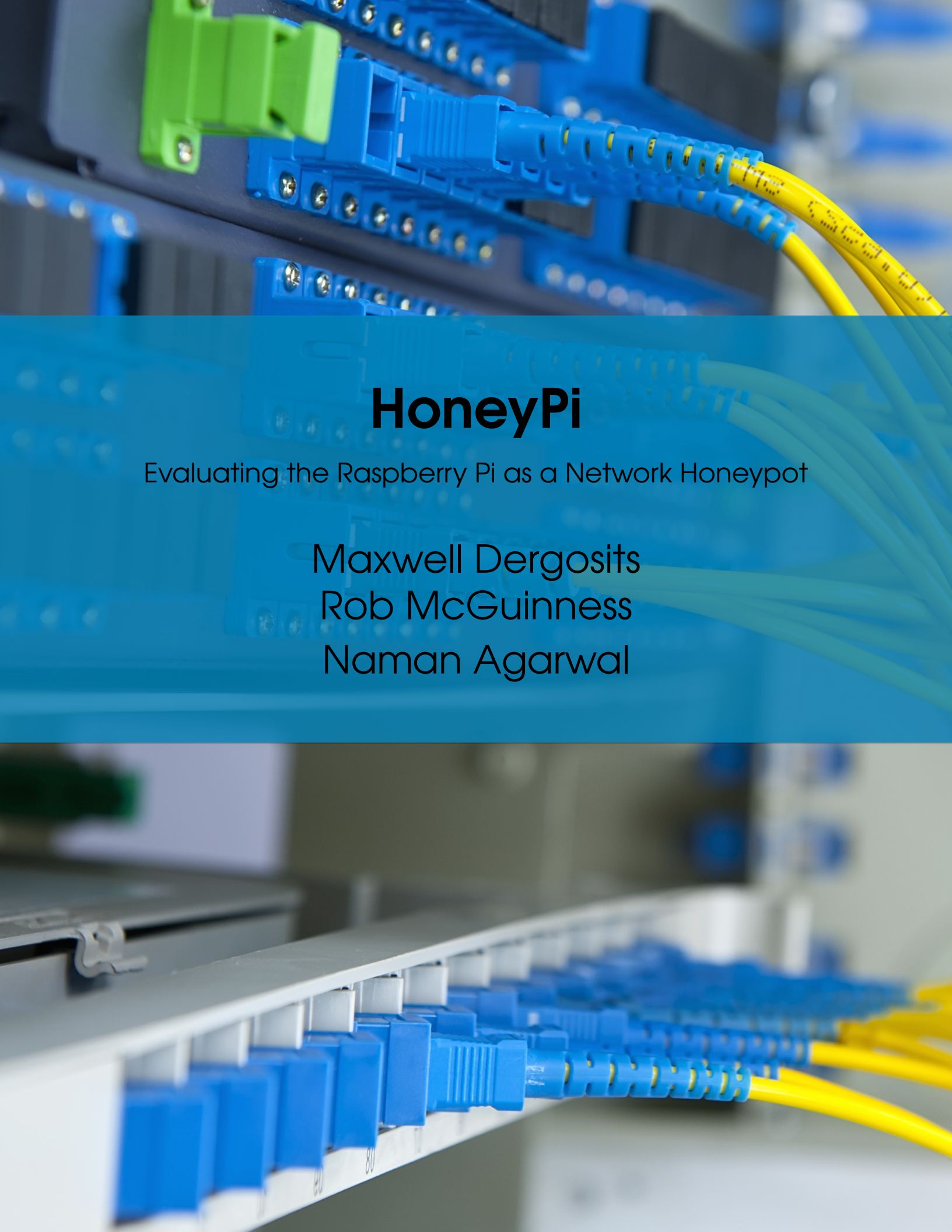

HoneyPi

Evaluating the Raspberry Pi as a Network Honeypot

Maxwell Dergosits
Rob McGuinness
Naman Agarwal

Abstract

We describe the operation of HoneyPi, an architecture utilizing Raspberry Pis to create a scalable distributed honeypot to capture and analyze network traffic. We evaluate the feasibility of using Raspberry Pis in such an architecture, and the performance of the system itself. Finally, we aim to create a general project that may be utilized by CS 3410 students.

FINAL PROJECT, CS 5413: HIGH PERFORMANCE SYSTEMS AND NETWORKING, CORNELL UNIVERSITY

GITHUB.COM/NAMANANAMAN/HONEYPI

September - December 2014

1. Introduction

Network honeypots are systems designed to capture and analyze large amounts of network traffic efficiently, while providing users with the ability to filter traffic by specific criteria and rules to gather significant statistics over a period of time. Honeypots are a great introduction to several systems required to perform basic computer networking tasks, including network drivers, the OSI stack, multithreading, and more.

Network honeypots traditionally are run on a single host that runs the network interface card in promiscuous mode to capture all packets that may cross the network. As such, the hosts running the honeypot are dedicated solely to this task, utilizing all available CPU power to process packets and aggregate user-defined information. Honeypots may utilize specific drivers or other kernel-level code to optimize processing throughput.

A honeypot is utilized in the computer architecture course at Cornell University (CS 3410), and is the final project that students are required to implement for the semester. Students are given a virtualized environment with multiple CPU cores and are told to implement a network driver and statistical aggregator that will accept commands defined by an automatic packet generation application. This ‘packet generator‘ supplies ‘command packets’ to the network driver that the aggregator must handle to start capturing statistics about some network information. The exact information to capture is given in the content of the command packet. There are four pieces of interesting information that can be gathered:

- The address the packet was sent from
- The procotol type of each packet (TCP, UDP, etc.)
- The destination port the packet was sent to
- The hashed value/signature of the packet

Address information enables tracking of ‘spammer’ addresses that are known to flood the network with packets, the protocol is useful for determining what types of traffic are crossing the network, the destination port allows tracking network probes or user application traffic, and the hashed value/signature permits a user to detect ‘bad’ or ‘evil’ packets that are known to have the same hashed value.

The network honeypot utilized by 3410 students and elsewhere is a shell of what normally is

required to capture network traffic. Students are fed packets via a specific interface given to them, but in practice, capturing network traffic efficiently and accurately is a nonuniform solution that must be tailored to the host environment and requires in-depth knowledge of the operating system kernel and the networking stack. However, a cheap, robust, and uniform environment can be found in the Raspberry Pi.

A Raspberry Pi is a single-board computer containing the bare minimum of hardware components required to run an operating system, and is designed to be run on Linux environments. Each Pi is outfitted with a basic USB controller, HDMI output, CPU, GPU, and Ethernet port, which is sufficient enough to run and test a network honeypot. Raspberry Pis by themselves are not as powerful as modern systems, but are very cheap in terms of cost; this makes them a desirable target for creating a distributed architecture with multiple Pis where each Pi does a subset of the work required for a traditional honeypot (it becomes a 'node' within the distributed honeypot architecture).

1.1 Motivation

This project evaluates the power and scalability Raspberry Pi as a node in a distributed honeypot. Our goal is to achieve the maximum performance possible on each Raspberry Pi, while scaling well with multiple Pis. The distributed honeypot we implement is based on the CS 3410 honeypot project, and could replace the current honeypot if we create a suitable skeleton for students to implement. We are also interested in utilizing a networking switch to perform IP routing, allowing us to evenly split network traffic across the Pis and make the packet generator send to an entire network, rather than a set of pre-defined locations.

2. Architecture

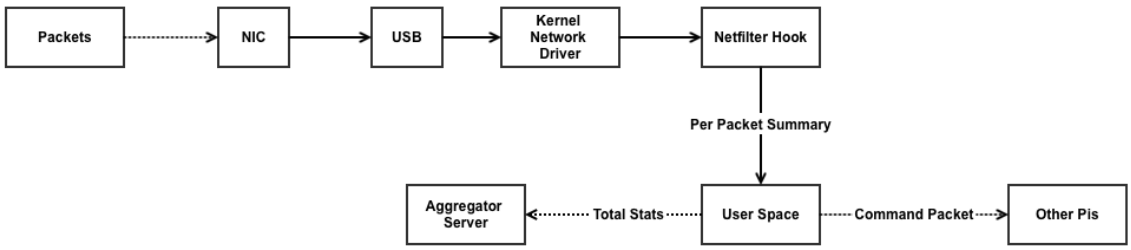

Figure 2.1: Packet Analysis Architecture

We iterated through many different approaches for implementing the honeypot. Our first major design idea was to use a framework for zero copy packet analysis. This would allow us to capture packets in userspace with little to no extra CPU processing required, giving us the ability to hash the packet in the Raspberry Pi's GPU. The Pi's GPU can hash data up to 14 times faster than the CPU [2], which would certainly be a significant performance benefit (the honeypot is required to hash every packet). We tried a few different libraries to achieve this functionality. We first attempted to use netmap [1], but we abandoned it as because it would not compile on any available Linux distributions compatible with the Raspberry Pi. In order to take advantage of the features of netmap we need, the kernel must be recompiled with netmap support. However, this was impossible due to the Raspberry Pi's ethernet driver, which is closed source. We then tried using the netmap implementation built into FreeBSD, but we ran into the same issue. Next, we tried using libnetfilter_queue, but this was not compatible with the Linux kernels available to us.

After running some tests (Figure 2.2), we decided that we could hash the packets on the CPU, as with roughly 1024 byte blocks (just below the maximum transmission unit size for the network), a Pi has roughly 125 Mbps of SHA256 hashing capability. However, a Pi's network card can only deliver 100 Mbps of traffic to the kernel module. Since this implied we could simply hash on the CPU without dropping packets, we decided to not copy the packets at all and do the hashing within

the kernel module itself. We ultimately decided to use DJB2 hashes since there were no available implementations of SHA256 that were linkable into a kernel module. In addition, DJB2 hashing rates were four times that of SHA256 (Figure 2.3).

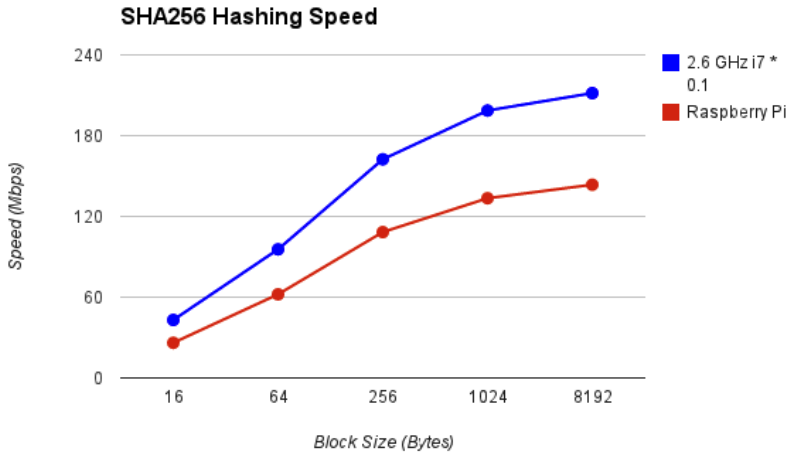

Figure 2.2: SHA256 Hash Rates

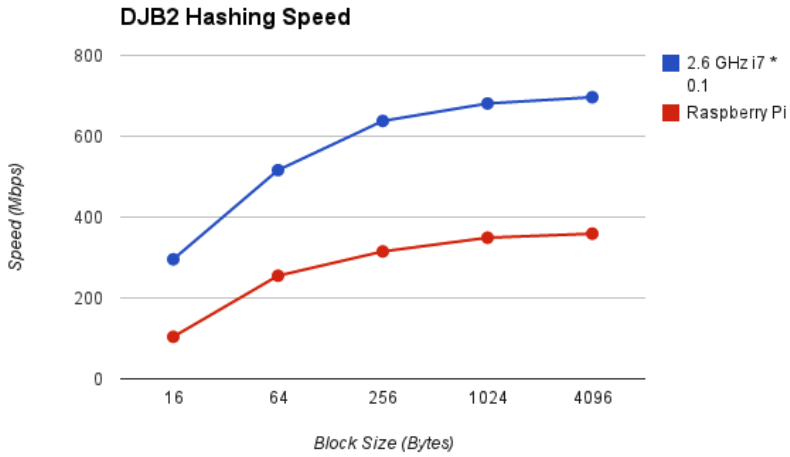

Figure 2.3: DJB2 Hash Rates

2.1 Packet Generator

The packet generator sends packets at a target bit-rate to a range of IP addresses. Every interval, it chooses a random IP address by randomizing the last byte of the IP address passed in as an argument to the program, and then chooses a packet type to send. The packet generator sends both honeypot command packets and random data packets. The command packets are identified by a magic number (the secret), and contain a command ID and command data. The commands are:

- HONEYPOT_PRINT
- HONEYPOT_ADD_VULNERABLE
- HONEYPOT_ADD_SPAMMER
- HONEYPOT_ADD_EVIL
- HONEYPOT_DEL_VULNERABLE
- HONEYPOT_DEL_SPAMMER
- HONEYPOT_DEL_EVIL

The add and delete commands instruct the honeypot to start or stop tracking the specified value for the destination port (VULNERABLE), source IP address (SPAMMER), or packet hash (EVIL). The print command instructs the honeypot to dump its statistics. Data packets may contain any one of the tracked values, or none at all. Packet types are chosen by a random probability distribution, with a much greater weight given to command packets if the number of tracked values is not near the target for each statistic. The generator stores the expected statistics to allow for verification of honeypot performance during testing. The generator does not need to highly optimized, and it does not add or delete many entries, so having these operations be relatively slow will not bottleneck the system.

Once the type of packet is chosen and the buffer is filled, the system sends the packet over a raw UDP socket to the randomized IP address. The system then waits for the calculated wait time using nanosleep. This wait time is calculated by dividing the average packet size (in bits) by the data rate. The data rate is assumed to be in Mbps, and therefore the calculated wait time is assumed to be in microseconds. The generator subtracts the elapsed time spent generating and sending the packet in order to stay as close as possible to the target data-rate. However, we discovered that nanosleep had too much overhead to be useful at high data-rates, so we use CPU spinning instead for shorter wait durations. The packet generator has multiple verbosity levels to print the command packets as they are sent and also print all of the expected honeypot statistics every 10 seconds.

2.2 Kernel Module

The (computational) bulk of the analysis happens in a loadable Linux kernel module. We are able to use a vanilla Linux kernel, with the only caveat being that the compiler must have the kernel headers to build against. Consumer distributions such as Ubuntu have these included by default, but the headers seemingly are not available for distributions such as Raspbian, the Linux distribution installed by default on Raspberry Pis. We opted to use Arch Linux as the necessary headers are included by default.

The kernel module registers a Netfilter hook, such that the kernel's network stack will call our hook for every packet received on Pi's network interface card. This Netfilter hook is inserted into the Netfilter hook chain before any IP routing is performed. We found that if we inserted the hook later in the kernel's IP stack, the kernel dropped any packets not destined for an IP associated with the Pi's network interface card, opposing our goal of capturing all network traffic. Within the Netfilter hook, we analyze each packet and extract the four pieces of information defined in the introduction- the source IP, protocol type, destination port, and DJB2 hash. This information is not extracted from command packets sent from the packet generator, but are instead passed directly to userspace.

The Netfilter hook is computationally expensive, as hashing the packets requires multiple CPU instructions per byte contained in the packet. We pack all of the extracted values into a small packet summary struct to be read and used by the userspace program. Once all of the fields of the struct have been populated, we place the packet summary struct struct into a ring buffer. If the ring buffer is full, we drop packets in the Netfilter hook until there is space available. In order to analyze the

performance of our honeypot, we also record the number of bytes in each packet and how many packets are dropped in the Netfilter hook.

The second part of the kernel module is the function that the permits the userspace program to read from the aformentioned ring buffer via a device file. The device file for our project is "/dev/honeypi". Each call to `read("/dev/honeypi",...)` writes one or more packet summary structs from the ring buffer into userspace or blocks until one is available.

2.3 Userspace Program

Many operations are not available in the kernel module, such as file I/O and transport-level networking, so a user space program is required to perform these tasks. We place all of the non-essential operations in the user-space program. This means that we record and report the statistics of all captured packets in the user-space program.

The userspace program reads each packet from the kernel module's device file. For each packet, the program checks if any of the four recorded values matches a value specified by a previously received command packet; if the packet is a command packet, the program updates what statistics it is actively recording. If one of the four values matches a value the program has been instructed to record, the program increments the corresponding counter for that specific value in a hashtable by 1 (the counter starts at 0). The exception to this is the packet protocol, which is always recorded and not controlled by command packets.

Since each Pi is only receiving a fraction of the packets sent by the packet generator, we need to relay command packets to the other Pis so that each of them are keeping track of the same statistics. To do this with low overhead, we decided to propagate the command packet via UDP. Each Pi binds to UDP port 4000 and broadcasts to all Pis when it receives a command packet from the kernel module. Since each of these packets is very small, it introduces a very small amount of network overhead. We do not include the command packet secret when it is propagated to prevent repropagation of already propagated packets.

Lastly, the program transmits the information it has recorded to the aggregation server. We use TCP connections to aggregate statistics due to the amount of information that needs to be sent with each update. We send the information to the aggregator every time a print command packet is received and every UDP 50,000 packets. While this introduced some overhead into the network, we found the accuracy provided by slightly more frequent updates to be necessary during testing.

2.4 Aggregator

The aggregator is a Python program, generally run on the same host as the packet generator (though the only requirement is that it's on the same network as the Pis and the generator), designed to aggregate all the statistics emitted by the userspace programs running on each Pi in the network into a central location. Combined with the packet generator, which keeps track of all the statistics it emits, we use the aggregator to evaluate the overall performance of the distributed honeypot.

The aggregator has 32 threads listening on a TCP socket that will accept connections from the userspace program on the Pis whenever they want to dump their recorded statistics. The userspace program establishes a 'session' with the aggregator that persists across multiple connections, and is represented by a message sent by the userspace program when it first starts executing. The aggregator records any statistics already emitted by a Pi during its session, such that the total statistics emitted

by the aggregator accurately represents the sum across all Pis capturing packets.

2.5 Switch

We use a HP Procurve 2900 switch to connect the Pis, packet generator, and aggregator. The switch is used to route packets from the packet generator to a single Pi, using IP routing tables. Each Pi is given a static IP address. We partition the IP space amongst the Raspberry Pis by programming 4 IP routes:

1. 192.168.1.0/26 -> 192.168.2.4
2. 192.168.1.64/26 -> 192.168.2.3
3. 192.168.1.128/26 -> 192.168.2.4
4. 192.168.1.192/26 -> 192.168.2.3

Note that more than 4 Pis could easily be utilized in the same network by increasing the subnet mask- for example, a mask of /27 would allow $2^{(27-24)} = 2^3 = 8$ different Pis. The switch is assigned to be a default gateway for the network, such that it can forward traffic appropriately.

3. Evaluation & Conclusions

Overall we have achieved most of our goals. We were able to create a distributed Honeypot that scales with the number of Raspberry Pis connected to the network. We were able to handle "random" traffic such that we don't need to hard code the IPs of each Pis into the packet generator. We can set up rules that could handle any number of Pis in the network. Ideally we would have been able to dynamically assign the routing rules based on the amount of network traffic to each Pi, but without an OpenFlow switch this would be extremely difficult and potentially require additional hardware.

3.1 Evaluation

Setup

We evaluated our system with two Raspberry Pis (the other two we were assigned were not functioning at the time of evaluation), a Macbook Pro running Ubuntu 14.04 with a Quad-Core i7 2.6 GHz processor, and an HP ProCurve 2900 switch. The aggregator and packet generator were both running on the Macbook Pro. Each of the Pis was running the kernel module and the user-space program.

Performance

The single Pi performance is poor due to of the limitations of the system-on-chip (SoC) network interface card (NIC) that the Raspberry Pi is equipped with. While the Raspberry Pi does have a Ethernet NIC built into the board, it is actually a 100 Mbps Ethernet to USB adapter. While we initially thought that we were going to be bottle-necked by the NIC of the Raspberry Pi, we found that the limiting factor was actually the Raspberry Pi's single core 700 MHz ARMV6 CPU. We found that with none of our software running on a Pi (no kernel module or user space program), the kernel's network stack consumes up to 95% of the CPU with only 60 Mbps of traffic being sent to the Pi (Figure 3.1). This prevented us from running the Pi's NIC at 100Mbps with the kernel module from the start. Even with as little as 25% of the CPU being utilized to hash packets, compile statistics, and send them to the aggregator, we are only able to achieve roughly one third of the line

rate of the network card without dropping packets (Figure 3.2).

Despite the sub-line rate performance we are able to easily scale with multiple Raspberry Pis. We are able to achieve almost double the line right of a single Pi with two Pis (Figure 3.3). This was a key result of our evaluation. We could have easily scaled with more Pis because each Pi only introduced a small amount of overhead on the switch and packet generator, and it's likely until there are dozens of Pis that the overhead would not have a noticeable affect.

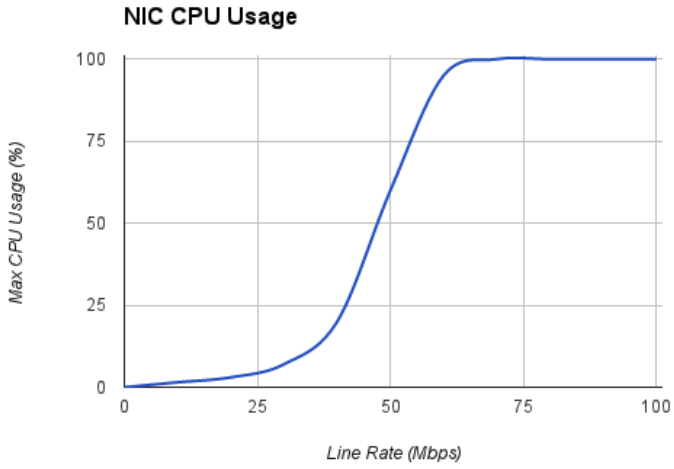

Figure 3.1: CPU usage vs Rx Mbps without software

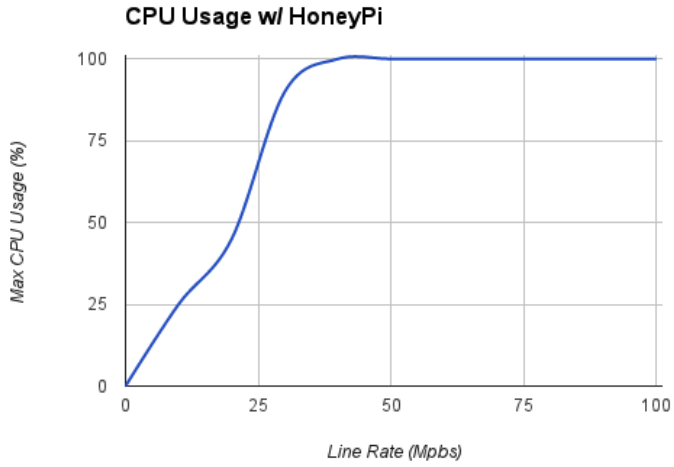

Figure 3.2: CPU usage vs Rx Mbps with software

Discussion

One motivation for this project was to evaluate the use of Raspberry Pis in high performance networking applications, and while it cannot perform nearly as well as dedicated servers, it provides suitable performance for how inexpensive it is. The Raspberry Pi was intended for hobbyist applications, making things that weren't "smart" able to connect to the Internet or interact with TVs, but we have proven that it is suitable for light networking loads and to cheaply scale distributed

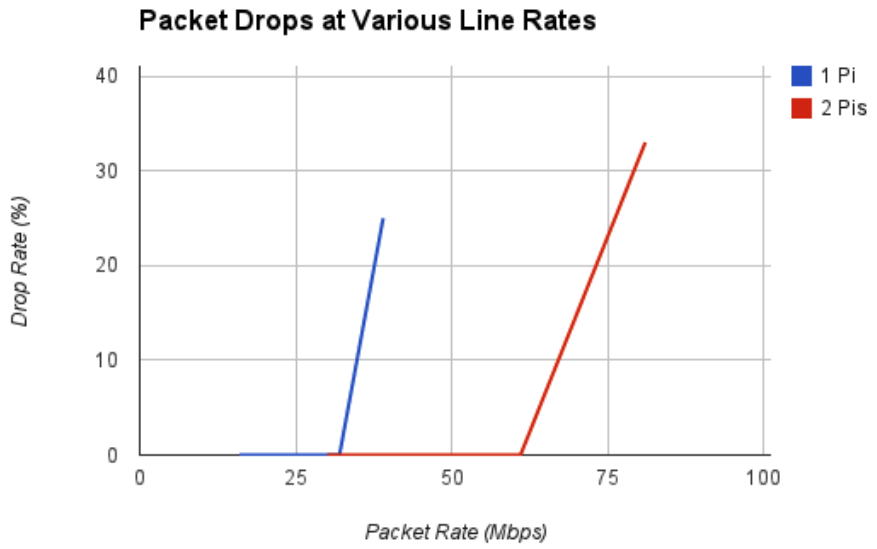

Figure 3.3: Honeypot drop rates

applications.

The limiting factor in our honeypot was due to the USB network card consuming a significant amount of the Pi's CPU. If the NIC had DMA and didn't require the CPU to handle all of the packet coyping, we would have probably been able to saturate the NIC and achieve 100Mbps per Pi.

3.2 Future Work

3.2.1 Optimizations

There are many possible optimizations that we could have implemented that would have improved honeypot performance. These include batching reads from the kernel module, batching command packet broadcasts, or finding a way to dynamically shape the traffic such that one Pi doesn't get more saturated than others due to randomness. Unfortunately, the fact of the matter is that the NIC's USB driver would have superseded any small possible improvements we could have made.

3.2.2 Other Boards

If we had more time and funding we would have ported the system to different system-on-chip platforms. One possible board is the ODroid[3]. Sadly it still has the same drawback that the Raspberry Pi has, a 100 mbps Ethernet to USB NIC. While we have not evaluated its performance, it would likely fall victim to the same network processing overhead issues as the Pi. However, the ODroid does have a much better processor (Quad core 1.7 Ghz), so the issue would be far less severe.

Another baord worth investigating is the Adapteva Parallella[4]. It has a 1 Gbps network card, an FPGA, 2 ARM cores, and 16-64 RISC cores. Additionally, it has two 1.4 GB/s interconnects that would allow for each board to communicate without having to go through the network at all.

3.2.3 3410 Lab/ Project

While the project as a whole is certainly too complex for an average CS 3410 student, there are parts of the architecture that can be removed and refactored as a suitable assignment. In particular, the userspace program and aggregator certainly fall within the realms of CS 3410 knowledge, and the aggregator could be repurposed to be written in C. This would be a great project for students to learn about distributed architectures and how to interact with them. The packet generator would have to be repurposed and take a static set of IPs, as it's impossible to provide every student with a switch with preestablished static IP routing tables. It's possible that the netfilter hook and read function from the kernel module could be factored to be implemented by a CS 3410 student, but this is likely beyond the scope of the knowledge required by the course.

Bibliography

[1] Luigi Rizzo, *netmap: a novel framework for fast packet I/O*. 2012.

[2] Unknown, *Hacking The GPU For Fun And Profit*. Link. 2014.

[3] Hardkernel Link.

[4] Adapteva Link.