A Passive Network Appliance for Real-Time Network Monitoring

Michael J. Schultz, Ben Wun, and Patrick Crowley
mjschultz@wustl.edu, bw6@cse.wustl.edu, pcrowley@wustl.edu
Washington University in Saint Louis
Department of Computer Science and Engineering
Saint Louis, MO 63130-4899

ABSTRACT

Network administrators lack the tools they need to understand and react to their changing networks. This makes it difficult for them to make informed, timely decisions regarding network management, capacity planning, and security. These challenges will only increase as networks continue to gain in throughput, become more complex, and encrypt more and more of their traffic.

This paper describes the Passive Network Appliance, or PNA, which is our proposed solution to this problem. The PNA provides snapshots of network behavior through time, in a cost-effective manner. The PNA is implemented on commodity hardware and can enforce network policy in real-time at the granularity of network frame arrival. This paper describes the system, and its evaluation in laboratory and real-world deployments.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network Monitoring*

General Terms

Design, Measurement, Performance

1. INTRODUCTION

This paper presents our Passive Network (Monitoring) Appliance (PNA). Unlike other monitoring software, our goal is to provide network administrators with a comprehensive view of their network at any moment in time, not just detailed analysis of specific protocols. Our PNA is built on low-cost commodity hardware and allows network administrators to grasp their network in greater detail without much setup and configuration. In this paper we:

- Describe and evaluate a *kernel-space* network monitor in both a laboratory and deployed environment
- Provide a *real-time* monitoring API that allows network administrators to enforce policy at the rate of network frame arrival
- Perform a *quantitative comparison* of kernel- and user-space packet throughput rates to motivate monitoring in the kernel

We developed our PNA system because we saw a lack of good accountability in current systems. Network administrators often use packet sampling, which captures a fraction of network traffic (Cisco's documentation suggests sampling 1 out of every 100 packets, which should account for 80% of flows [2]). Unfortunately this sampling technique can, and often does, provide a biased view of the network towards large "elephant" flows [14, 18].

The only way to have an unbiased view of the network is to capture and process every packet. This is difficult to achieve given plateauing processor clock rates and increasing network bandwidth. Additionally, end-users are beginning to use secure communications (HTTPS, VPNs, etc.) that make most forms of packet inspection obsolete. Section 2 further discusses our motivation for this problem.

While hardware solutions to some of these problems exist, they are typically expensive and can be difficult to maintain; software solutions are easier to maintain but are slow. Our PNA system finds a balance between the two extremes by using an efficient kernel module to provide a software system that is both fast and easy to maintain. The high level details of our system are discussed in Section 3. Section 4 then digs down into the specific design details of our module and gives an explanation of how our software can be extended to support the specific application policy and monitoring for network administrators.

Section 5 presents our evaluation of the kernel module in a laboratory setting and discusses how our deployed system performs in an open enterprise level network. Our use of a kernel module is motivated in Section 6 and shows why we chose to operate in kernel-space instead of developing a user-space application.

2. MOTIVATION AND BACKGROUND

Network administrators currently use a hodgepodge of networking equipment and software tools in their networks. These systems can be routers, network/port address translators (NATs), or firewalls, among others. Each of these systems can introduce strange behavior to the network, fail unexpectedly, or be purposefully taken down through attack vectors. While these systems may provide some level of network accountability their focus is either too narrow (ignoring packets) or too general (skipping packets to maintain service guarantees) for full transparency. Simply put: understanding how a network is being used at any moment in time remains a difficult problem.

As a motivating example of a failure, imagine a collection of end-hosts connected to the Internet through a network/port address translator (NAT). Suddenly, one of the end hosts opens a large number of HTTP sessions. This causes the NAT to become overloaded, dropping new connections and denying service to all the end-hosts behind the NAT. The NAT no longer provides the service it should and is not able to log all the sessions that were opened, making it impossible for the network administrators to track down the misbehaving end-host. This is a specific example of a general problem our system is designed to detect in real-time: allowing network administrators to quickly detect and correct potential issues.

Another goal of our system is to provide accountability for different network types. Enterprise networks can range from having an "open" policy to a "closed" policy. An open network is one where network administrators allow all Internet traffic and avoid traffic shaping and protocol restrictions. A closed network takes the heavy handed approach and only allows connections that match their defined network policy. In both situations, network administrators will want to account for who and what consumes their network resources. Additionally, a closed network should continuously audit their traffic to ensure existing systems are properly implemented and enforcing the correct policy.

In both a denial of service and accounting for network traffic, network administrators want a monitor that provides the information quickly and accurately. To do this, a network monitor must capture every packet sent through the network. Unfortunately, most hardware options are expensive or difficult to develop or modify so they are not in wide use [1, 3]. Software options are lower cost and offer more flexibility but are typically slower than their functionally equivalent hardware counterparts [6].

In general, network administrators want a full view of what their network at any given time. This way they know what machines consume most of the network bandwidth, what ports are most commonly used, what ports transfer the most data, and what systems open the most connections or sessions. The most accurate method is to capture every packet in the network and log them to disk, but this takes a massive amount of storage. An alternative is to store only the header data, but this can still take a significant amount of storage and writing to disk may not be able to keep up with the data stream.

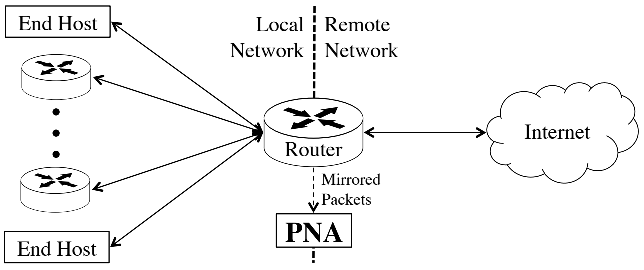

Figure 1: A PNA installation with the local network connected to the Internet through a gateway router that mirrors packets to our PNA.

We wanted a network monitor that would satisfy the following conditions:

- Run on a low-cost platform
- Able to keep pace with the 1 gigabit per second bandwidth of typical enterprise level networks
- Offer network administrators the ability to easily customize the monitor to their needs

We could not find a monitor that met all these conditions, so we built our Passive Network Appliance (PNA). Our PNA system puts a different spin on data collection by making a compromise between the visibility we have into the network, the details we store, and the monitored time granularity for that network. This allows us to use a low-cost commodity system that sees almost all the packets, maintains logs for offline analysis, and allows real-time monitors to enforce policy as network frames arrive without significant development effort.

3. PNA DESIGN

This section discusses the design of each of the major steps in our PNA system; a more detailed discussion of the implementation is presented in Section 4. To achieve our goal of a fast, low-cost, easy to maintain system we built a Linux kernel module that hooks on every packet received by the system. The core PNA software is broken into two components: a two part logging step that maintains both active and inactive flow summary statistics and a real-time monitoring step that can look for specific patterns.

Consider a PNA system that is installed at the gateway router between two networks, as seen in Figure 1. The left portion of the figure shows the "local" network, while the right is the "remote" network. Packets that traverse the gateway router are mirrored to our PNA software for processing.

3.1 Active Logging and Log Generation

Our PNA is designed to record statistics for every packet in the network, so the active logging step occurs first and ensures every packet is processed as quickly as possible. This step of processing records the number of packets and bytes seen in each network flow. When a packet arrives at the system, it is passed to our PNA software. The software determines the directionality of the packet and whether it originates from a "local IP"—an IP that resides on the monitored network. If the packet belongs to an existing bi-directional flow, it updates byte and packet counts for that flow and passes the information to the next step. If the flow is new to the system it creates an entry in the flow data table, sets the initial byte and packet counts, records the first packet arrival time for this flow, and then moves to the next step.

Flow data is maintained in a table stored in main memory for a fixed length of time. Periodically this table is flushed to disk to provide a snapshot of what the network looked like during that time window. Main memory allows us to have a large hash table while still maintaining flexibility. This translates into monitoring many active flows and helps reach our goal of tracking all packets in the network.

During the active logging step, the table that maintains flow entries is periodically flushed by a user-space log monitor. This ensures entries in the flow table represent only the most recent data and allows the snapshots to remain small. A snapshot is used to represent the state of the network during the preceding time period and allows network administrators to look through traffic data and perform *post hoc* analysis.

These snapshots are stored locally and then can be aggregated to a central data store for offline processing. Though certain details (such as packet arrival time and packet payloads) have been removed, we believe that—for offline analysis—the crucial data remains (active IPs, active ports, traffic volumes, packet counts, and relevant timestamps). This data can then be processed to rebuild the approximate network conditions during a specific time period. Moreover, long term trends in network activity can be visualized and analyzed using these logs to understand how the network is used.

However, offline log analysis will not be able to detect or enforce network policy in real-time so we have also developed a real-time monitoring system for analyzing network frames as they arrive at our PNA.

3.2 Real-Time Monitoring

With the packet committed to memory, the PNA focus on the real-time monitoring step that enables network administrators to enforce network policies *as network frames arrive*. From the active logging step, the packet and flow counters are passed to the real-time monitoring subsystem. This system allows an arbitrary number of monitors to be chained together for deeper and more specific analyses. Conceptually, these monitors are able to maintain private state between packets or perform packet inspection as the network administrators see fit. Our current implementation uses two stateful, real-time monitors:

- A connection monitor that tracks IP-to-IP connections between local and remote hosts
- A local IP monitor that tracks statistics about local hosts that have been active

While these are the only monitors we have implemented, our API allows a developer to add new monitors that look at specific packet types or track other data as they see fit.

Our real-time monitoring system hooks into an alert system that enables the network administrators to log, email, or isolate an offending IP the moment our system processes and detects the violating packet.

4. PNA IMPLEMENTATION

Our PNA software is implemented as a Linux kernel module split into an active logging step and a real-time monitoring step that allows analysis of network frames as they arrive through our API. This sections discuss the implementation details and explains the decisions we made while developing the PNA.

A PNA system runs on commodity hardware, so we use a standard server-grade system with a multi-port gigabit Ethernet card. Our system is co-located with a core network router for an enterprise level organization using a 1 gigabit per second link, an example which can be seen in Figure 1 of the previous section. The router mirrors both packets destined to the Internet and packets from the Internet to our device for capture and analysis. Once the packet is mirrored to our PNA, it enters the system and follows the path outlined in Figure 2. While our system could be installed in-line with the packet flow, that has not been our goal nor have we explored how the system operates in that scenario. Once our system has received a packet it begins processing by decoding the packet headers.

Our system uses a kernel module built against a modern Linux kernel. A kernel module allows us to hook into the Linux networking stack at a very early point in processing, enabling us to achieve higher throughput than standard user-space approaches (Section 6 covers this in more detail). This approach comes with the risk of using the Linux kernel API that may need to be updated as the kernel continues to evolve. However we

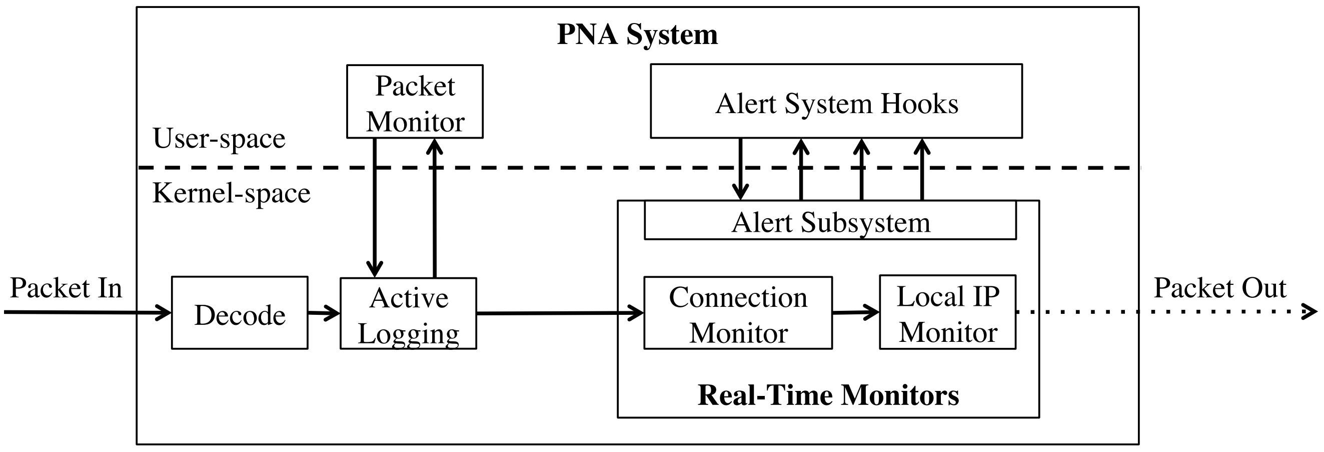

Figure 2: Block diagram of the PNA software architecture.

have tried our best to minimize interactions with the kernel so changes to the kernel API should be simple to integrate into our module through patches or compile time options.

4.1 Active Logging

Our software runs in kernel-space to enable quick, lightweight packet processing by finding the flow key for every packet and storing summary statistics in a hash table. We take advantage of running in kernel-space by hooking into the network stack during the `netif_receive_skb()` function prior to the traditional network layer handling. We chose this point because it is the first opportunity during packet processing that is not tied to the underlying device driver. It is also early enough that the kernel has not spent excessive time working through unnecessary packet processing steps.

Since we hook early in the networking stack, our software must first decode the Ethernet, IP, and TCP/UDP headers to extract the flow 5-tuple (source and destination IP addresses, source and destination ports, and transport protocol). Next, we determine if the packet is inbound or outbound based on the network prefix and network mask the system has been configured to use. By doing this we introduce the notion of "local" and "remote" IP addresses and ports to our system. Local addresses and ports belong to machines in the network we are monitoring while remote addresses and ports belong to machines that are outside of our network and control.

Figure 3 breaks down how a flow key will be generated for a connection between a local machine using the IP 192.168.53.7 and port 53271 to talk to a remote machine on port 80 at 128.252.165.4 over TCP. The top of the figure shows the bi-directional TCP session with the local machine on the left and the remote machine on the right. The middle of the figure shows an

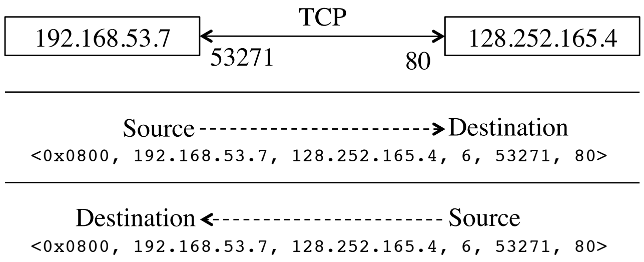
192.168.53.7
TCP
128.252.165.4
53271
80
Source
Destination
<0x0800, 192.168.53.7, 128.252.165.4, 6, 53271, 80>
Destination
Source
<0x0800, 192.168.53.7, 128.252.165.4, 6, 53271, 80>

Figure 3: Top: A TCP session between a local and remote host. Middle: The local hosts sends an "outbound" packet resulting in the same flow key. Bottom: The remote host sends an "inbound" packet giving the flow key shown.

outbound packet whose source is in the local network and is destined for an IP on a remote machine. In this example, since the PNA is configured to treat IPs in the 192.168.0.0/16 range it recognizes that the source is a local machine and constructs the appropriate flow key, as seen below the connection. The flow key is `<network-protocol, local-ip, remote-ip, transport-protocol, local-port, remote-port>`. The inbound packet seen at the bottom of the figure, will generate the same flow key—despite being in the opposite direction. Again, this is because our PNA is aware of the network to consider local. Since we assume the PNA exists at a gateway, internally routed traffic should not be seen (if we do see internal traffic, the source IP will always act as the local IP and flows will be monitored as uni-directional flows).

Using the flow key, a hash value is computed and we find or create the corresponding entry in a large hash table containing all the flows for the current time period (10 seconds). If a collision occurs we use quadratic probing until we find a clean entry or we have tested 32 entries[1].

The hash table represents a 10 second snapshot of the network conditions by accumulating data for individual flows. Every flow entry accounts for the current inbound and outbound bytes and packets, the timestamp of the most recent packet seen in that flow, and the timestamp and direction of the first packet seen in the flow to determine when and which IP initiated the conversation. Additional summary data (such as the state the flow claims to be in at the beginning and end of the snapshot) could be added to the entry as well, however we have yet to find a need for this.

Currently, the PNA can hold over 8 million flow entries in memory for each 10 second snapshot period. For most enterprise level networks this should be sufficient for recording all the active flows. Once the active logging step has updated the flow entry, the packet and entry are passed on to the real-time monitoring step.

4.2 Real-Time Monitoring

The real-time monitoring system provides a simple API for developers to implement a monitor that looks at network frames as they arrive and detect specific misbehavior. This step follows directly after the active logging step completes. Multiple monitors can be chained together sequentially with each monitor performing a separate analysis on the packet or flow. Our current implementation consists of two real-time monitors (a connection monitor and local IP monitor) and allows the additional stages conforming to our API to be added by network administrators. If any of the monitors detect malicious or abnormal behavior they are able to hook into our alert subsystem to notify the network administrators.

4.2.1 Connection Monitor

Our first monitor tracks unique IP-to-IP connections. Tracking IP connections informs a network administrator about how many conversations two machines are having (many simultaneous sessions may indicate bad behavior) or the volume of data transferred between two specific hosts (copying a database off site). In these cases, the network administrator may wish to know about and act on this activity as soon as it begins happening; this is what our connection monitor is designed to achieve.

When a packet is received by the connection monitor it uses only the local and remote IPs from the flow key to generate a hash value. This value is used as an index into a connection table and finds or creates the corresponding entry. An entry tracks the number of unique sessions (transport layer port pairs), bytes, and packets divided into transport protocol types and packet direction. Currently our connection table can hold over 2 million unique IP pairs for each snapshot of time. Periodically, a function executes to reset the hash table for the next time period.

After every entry modification of the connection table, the system checks the configured thresholds and an alert is issued if the threshold is surpassed. As an example, if a connection has opened an excessive number of sessions between two machines (an average connection has about 3 sessions[2]), then our system detects that violation immediately and issues an alert that can be emailed to a network administrator or isolates the offending IP. Once the connection monitor has completed, the packet and exit status of the connection monitor are passed to the next real-time monitor.

4.2.2 Local IP Monitor

The local IP monitor keeps track of the activity for each unique local IP on the monitored network. The data this monitor maintains allows network administrators to monitor the number of ports, remote IPs, bytes, and packets machines on their network connect or send. This helps identify machines that are scanning for open ports on a remote host or connecting to a large number of external hosts.

An entry is keyed using the local IP address and, like the connection monitor, computes a hash value and creates or selects an entry from the local IP table. Once an entry is found it tracks the number of port-to-port connections, bytes, and packets that a local IP has been involved in (again, divided into transport protocol types and packet direction). Currently our local IP table can hold over 130 thousand entries for each snapshot of time. A periodic function executes to reset the hash table for the next time period.

Like the connection monitor, after each entry modification, configured thresholds are checked and an alert is sent if a threshold is violated. If a local IP suddenly tries to connect to many remote hosts, our system will detect that the moment it happens and issue the alert.

4.2.3 Additional Monitors

Our real-time monitoring API provides the basic hooks that allow network administrators to develop additional monitors in a straightforward fashion. Both the connection and local IP monitors were built around this API. So far, our experience developing these monitors has been simple and we believe other monitors can be added without much trouble. Adding a monitor to the system requires two steps: implementing the necessary functions and then declaring those functions to the system.

The functions a monitor is able to implement are:

[1]An empirical analysis of our deployed system showed that over 90% of flows are fulfilled within 32 attempts.

[2]Again, this value comes from empirical data gathered from our deployed system.

Function	Description
`init()`	Initializes data structures for a monitor.
`release()`	Releases resources held by a monitor.
`hook()`	Hooks on every packet processed by the PNA. Takes a flow key, direction, socket buffer, and data pointer as arguments.
`clean()`	Periodic function for maintenance of data structures used by a monitor.
`pna_alert()`	Sends an alert to a user-space process. Takes an alert reason, violator, and timestamp as arguments.

Table 1: Summary of our real-time monitoring API functions.

`init()`, `release()`, `hook()`, and `clean()`. While this may appear similar to the `netfilter` API, we want to be able to hook on any packet—not just IP packets. What each of these functions does is summarized in Table 1. The `init()` function executes once when the kernel module is initially loaded, our monitors use this to allocate memory for the hash tables and set the memory to known values. Conversely, the `release()` function executes when the kernel module is unloaded; our monitors use this to free the memory allocated during `init()`. The `hook()` function acts as the main workhorse and takes several arguments:

- The flow `key` identifies the entry in the initial table (during the active logging stage). This is convenience data as it is simply data extracted from the packet, but earlier steps have already determined and set the local and remote IPs, local and remote ports, and the transport protocol that the packet uses.
- The `direction` of this packet. This informs the hook as to whether this packet is entering the network or exiting the network since the flow key has abstracted the source and destination values.
- The kernel socket `buffer` contains the complete packet data. This may be useful if a monitor only wants to look at DNS (port 53) packets. It can quickly check the port value based of the flow key, and then perform further packet processing as needed.
- The `data` argument can contain arbitrary data. This allows meta-information to be passed from one monitor to the next if needed. In our monitors, this argument is used to pass the return value of the previous monitor.

Finally, the `clean()` function executes periodically and can be used to run any asynchronous maintenance routines; our monitors use the `clean()` function to reset the hash tables every 10 seconds so thresholds operate on 10 second intervals.

A simple monitor that requires no state only needs to implement the `hook()` function, which executes for every packet received.

When the necessary functions are implemented they must be declared and told when to execute in our system. The PNA will then begin normal monitoring with an additional real-time step executing for each packet received.

4.2.4 Alert Subsystem

Our alert system allows any monitor to check for arbitrary violations and issue an alert causing the violation to be logged, emailed, or automatically handled by existing systems. Both the connection and local IP monitors have configurable thresholds and will issue alerts when there is a potential violator on the network. Alerts are issued using a function call to `pna_alert()` that takes a defined reason, the violating IP address, and the timestamp of the violating packet. This alert is then asynchronously forwarded to a user-space process that emits a log message with the reason, violator and timestamp and then can execute a program to perform some action. We place no restriction on the program that executes. The receiving program will be executed with the three arguments above in user-mode so it does not interfere with the normal operations of our system.

4.3 Log Generation and Analysis

Flows recorded during the active logging step are periodically flushed to disk through a user-space program to allow records to be aggregated over time and from multiple locations, allowing deeper analysis of the data. To create the time snapshots of the table, prior to the reset, a user-space program accesses the table and logs each of the flow entries to a file on disk. Given our current flow table size of 8 million entries, if during a 10 second window our table is completely filled, we calculate that the log file for that snapshot will be under 300 MiB and take about 1.6 seconds to emit in the worst case. Once the user-space logging process begins executing our kernel module switches to a second in-memory table so no packet is missed while the table is being dumped to disk. Since the user-space program runs separately from our kernel module, it does not interfere with normal operations.

Once the log file is complete, it can be stored locally or pushed to a central data store such as a file server or storage cloud. As this data represents an identical view to the data stored during the active logging phase, it can be analyzed offline over larger time scales.

Our current offline analysis parses each log file and provides daily reports for the "heavy hitters" in terms of data volume and packet counts and also identifies IP addresses which have connected to the most external IP addresses. Even with these two simple analyses the information produced by our deployed PNA has been used to identify interesting results in the network that may have gone undetected in normal operation and has allowed network administrators to better understand how their network is used throughout the day.

5. EVALUATION

The evaluation of our system is twofold:

- We developed and tested the system described in this paper in a laboratory setting
- We deployed an early version of our software and have been running it in an operational setting for the past 9 months

Both the laboratory and deployed systems were purchased with the same hardware configurations. The platform is a Dell PowerEdge R410 with two quad-core Intel Xeon L5520 ("Nehalem," no Hyper-Threading) processors operating at 2.27 GHz with 12 GiB DDR3 (1066 MHz) system RAM, and a 160 GB 7200 RPM hard drive for storing log files.

5.1 Laboratory Results

Our laboratory experiments looked at both correctness (did the PNA capture all the flows we sent it) and throughput performance. For our laboratory experiments, we connected our system to the Open Network Laboratory (ONL) infrastructure [20]. ONL allows us to connect real machines together to replay trace files or generate packet streams at gigabit speeds. During laboratory testing we use the CentOS 5.5 distribution with a custom configuration of Linux kernel version is 2.6.37 using the `igb` driver (version 2.1.0-k2) for the Intel 82576 network adapter. The configuration is based on the suggestions by Gasparakis and Waskiewicz to disable unneeded systems like sound, USB, and IPv6 [11].

To confirm that our system correctly captures all packets when it is not fully loaded, we replayed a packet trace using `tcpreplay` [19] and confirmed that the summary information that was logged matched what was sent to the system.

5.1.1 Performance Evaluation

Our performance evaluation focuses on the worst case traffic throughput, finding how the packet size affects our throughput, and how many flows we are able to record and how many flows we miss in the worst case. Testing the system behavior under heavy load is done using the Linux kernel packet generator (LKPG) [16]. The LKPG allows us to define exactly how the packets are generated so we can have a consistent load throughout testing. It is designed to generate packets as fast as the end-host allows, but since this is less than our target of 1 Gbps we have constructed an ONL network topology that aggregates five packet generators together to create the packet stream we want.

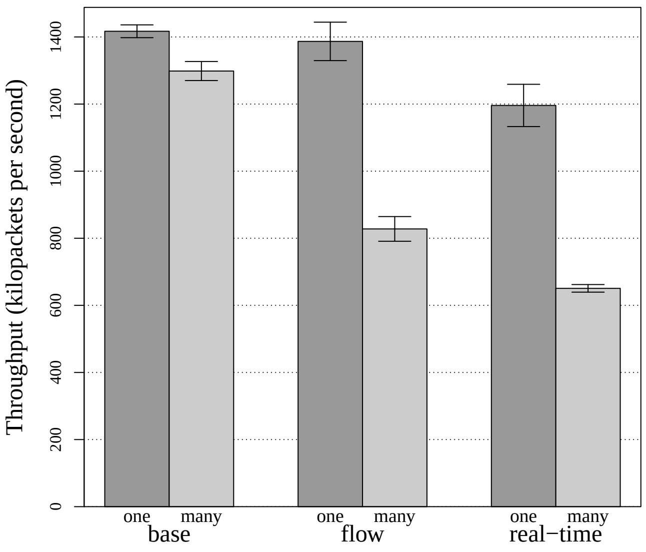

Figure 4: 64 byte packet throughput in kilopackets per second for the three monitor configurations and two packet flow patterns with 95% confidence intervals shown.

We ran four separate experiments to measure the throughput for minimum sized Ethernet packets (64 bytes) and maximum sized Ethernet packets (1514 bytes) using either a single flow (a single source/destination IP and port) or multiple flows (many source IPs to many destination IPs over many source-destination port pairs). Each experiment was run 10 times for 60 seconds each excluding a short warm-up period with performance data collected every 10 seconds. For reference, we abbreviate 64-/1514-byte packet sizes as min/max and single/multiple flows as one/many in our figures.

Our evaluations run the PNA software in several configurations:

- The "base" monitor hooks into Linux, decodes the packet headers, and discards the packet. No entries are inserted or updated in the table.
- The "flow" monitor does the same as the base monitor plus it performs the active logging step described in previous sections.
- The "real-time" monitor does the same as the flow monitor plus it executes the two real-time monitors we have developed to track connections and local IPs.

Figure 4 shows the throughput, in kilopackets per second, for 64 byte packets using the three monitor configurations with both one flow and many flows. The peak throughput using 64 byte packets is 1,488.10 kilopackets per second. This represents the worst case performance bound for each of the three configurations. Both the "one" and "many" flow patterns are worst case scenarios for the network card and low level Linux networking because they create large numbers of distinct packets in the system. The packet throughput decreases as each monitor configuration adds more code, especially when the real-time monitors are added. The throughput also decreases when the workload involves many flows, due to the increased number of hash table insertions and the additional time spent decoding headers and finding open table entries. Since each packet belongs to a unique flow, no hash table entry will be used twice. This causes the monitor to check each entry until it reaches the 32 entry probe limit.

Under these conditions, the base configuration shows the worst case performance of our software with no load beyond hooking into Linux and processing the packet headers. This demonstrates the peak possible performance of our PNA hardware. The flow configuration shows the worst case performance when packets are inserted only into the flow table. The difference between the "one" and "many" distribution patterns becomes much more clear because each packet in the many pattern must search for a new table entry instead of continually using the same entry. Finally, the real-time configuration shows the performance when the system performs both active logging and runs the connection and local IP real-time monitors.

Figure 5 shows the throughput of our PNA under several different packet sizes under worst case conditions (i.e. each packet is part of a unique flow and packet arrivals are at line rate). This demonstrates that the PNA is able to process almost every packet once the packet size is greater than 256 bytes. Note that throughput is represented as a percentage of the maximum possible throughput for a given packet size. For example, with 128 byte packets a gigabit link will be able to achieve about 844 kilopackets per second and our PNA sees about 644 kilopackets per second resulting in about 76% of peak throughput.

To determine the percentage of packets dropped by the PNA under worst case conditions we count the number of insertions that were successful and unsuccessful, then calculate the maximum number of packets that could be dropped at the network card. Figure 6 shows these values with successful entries labelled as "inserts," unsuccessful entries labelled as "drops," and network card drops labelled as "max-nic." To evaluate under worst case conditions, 64 byte packets are generated at line rate with each packet belonging to a unique flow. A successful entry is defined as an entry that was inserted into the hash table and an unsuccessful entry is one that looked for an entry in the hash table but could

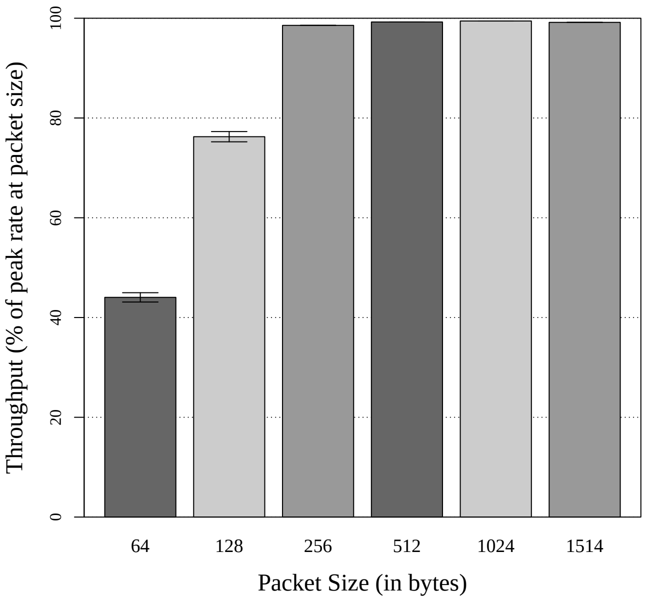

Figure 5: Throughput expressed as percentage of peak throughput for packets sizes 64, 128, 256, 512, 1024, and 1514 under the "many" flow pattern using the real-time monitor (with 95% confidence intervals).

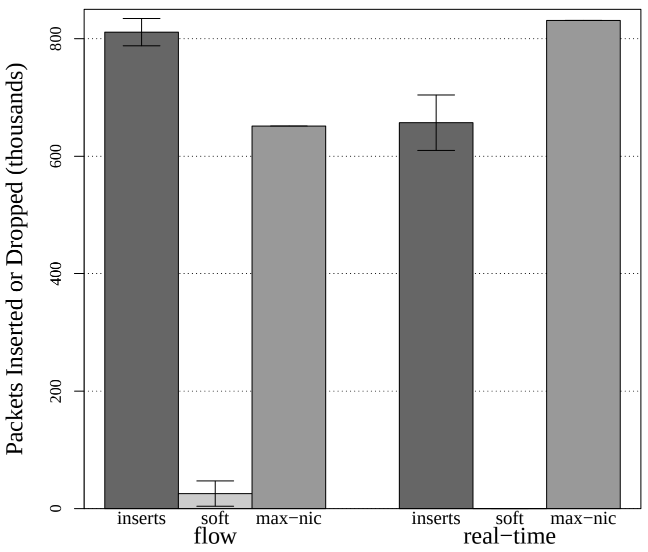

Figure 6: Comparison of successful table insertions against packets that could not be inserted by the PNA software and packets that were dropped at the network card under worst case traffic conditions.

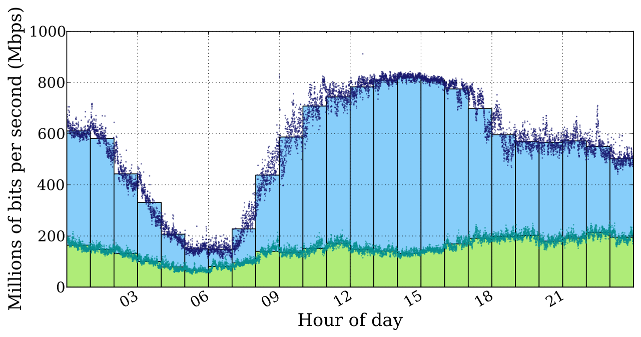

Figure 7: The throughput measured by our deployed PNA node during a typical day.

not find a free entry before the probe limit was reached. The theoretical number of packets drops at the network card is calculated as the maximum number of 64 byte packets per second less the number inserted in the table less the number dropped in software. While the number of drops at the network card may look high, it is in line with the throughput under worst case conditions. Additionally, capturing 40% of packets under this workload is (by definition) 40 times better than sampling at a ratio of 1:100 as recommended by Cisco documentation [2].

5.2 Deployed System Results

Our deployed system has been used in an open enterprise network for the past 9 months and has reinforced its technical abilities as well as demonstrated how it can be useful to network administrators. The prototype system uses the CentOS 5.5 distribution and runs the default configuration of Linux kernel 2.6.34 and the same Ethernet driver as the 2.6.37 kernel. Table entries in the prototype version are much larger than the updated version but are constrained to the same memory footprint. Thus, the prototype is only able to hold about 65k flow entries, 32k connection entries, and 16k local IP entries during a 10 second window. The insertion/updating scheme is also much more cumbersome, but it is designed to track the same data in real-time as the system discussed in this paper.

Log files are emitted at 10 second intervals and are compressed and sent off-site for analysis at 10 minute intervals. Our system has proved sufficiently useful to the network administrators that we have not had an opportunity to upgrade to the current release.

A typical day's traffic can be seen in Figure 7 with the solid bars representing per hour average throughput for inbound (blue) and outbound (green) traffic and each dot representing the 10 second averages. While the deployed system has produced a wealth of interesting data, there are some caveats that we have discovered. During heavily loaded periods of the day (full link utilization), our deployed monitor drops about 1% of the traffic at the network card. This still allows us to know what the vast majority of network traffic is and we expect our new release to perform better.

Even with the older deployed PNA software, we have been able to detect network anomalies that were unknown or previously corrected and had since regressed. An example of an anomaly our system has detected is the periodic scanning of the network for printers with open HTTP ports. Most new printers are not only network-aware, but also contain a small web server with status information, print queue access, and other potentially confidential information. While a network administrator will be aware of this, the person installing the printer will not know the network policy and may not think to disable or protect the web service allowing remote hosts to have unauthorized printer access. Increasingly many embedded devices that are network-aware will send data or have open ports by default that may be unknown to network administrators or installers, leaving the network open to both new and old attack vectors.

An operational problem that we have yet to handle is that the network operates a bi-directional 1 Gbps link (2 Gbps in aggregate) and packets are mirrored from the router to our system over a 1 Gbps link. If the network realizes the full 1 Gbps bi-directional utilization, half the packets that our system should see will be dropped at the router. One solution to this problem is to use a 10 Gbps link between the router and our system. This will add additional cost to a deployment and may not be supported by all routers. Another solution is to split the traffic over two 1 Gbps links and merge the streams in our system. Early experiments have shown this to be a feasible solution though we have not focused on the performance aspects yet. In both cases, there are operational consequences that must be considered by network administrators before either option is deployed.

6. KERNEL-SPACE VS. USER-SPACE

Operating in kernel-space allows us to avoid system call overhead to realize higher throughput than user-space monitors on commodity hardware. The relatively high cost of system call overhead for packet processing was discussed by Luca Deri [7] and re-emphasized by Braun et al. [5]. While most solutions try to minimize the overhead, we decided to move the software into the kernel and completely avoid the overhead.

For a typical developer, the difficulty and cost of kernel development are too high and advantages too slim. Our premise is that the PNA system has done the difficult part of kernel development and provides a straightforward method for developing real-time monitors that can be used without much more hassle than a user-space

Kernel Module	PF_PACKET	PF_RING
951.75 ∓ 1.23	495.89 ∓ 1.01	747.72 ∓ 7.38

Table 2: Throughput of packet capture methods measured in megabits per second with 98% confidence intervals.

program.

To quantify the throughput of the different approaches, we compared our "base" kernel module against the default Linux kernel packet capture stack and Luca Deri's `PF_RING` capture stack [7]. The base kernel module used for this measurement study is the same as in our laboratory evaluation above. `PF_PACKET` is the default packet capture method for Linux kernel 2.6.37 and we use the `pcount` program developed by Luca Deri to measure throughput. `PF_RING` is a packet capture method that associates a ring buffer with a dedicated network card to bypass the standard Linux networking stack allowing less system call overhead resulting in higher throughput. Performance of `PF_RING` is monitored through the `pfcount` program which mimics the behavior of `pcount`, both programs are designed to simply count the number of packets received and discard them without further processing (like our kernel module).

It should be noted that when combined with Fusco and Deri's Threaded NAPI the `PF_RING` implementation can achieve full Gbps line rate in a multi-core configuration [10]; however we are only interested in single-core performance for this study. Our interest in single-core performance is motivated by the fact that we believe we can exploit multiple cores by taking a pipelining approach to increase the amount of work each monitoring step can perform. Because of this we are focusing on the single-core performance for this work and will focus on multi-core performance in future work.

Table 2 breaks down the three packet reception routines we evaluated and their associated throughput measured in megabits per second with 98% confidence vales. Both the PF_PACKET and PF_RING performance numbers closely match those found by Fusco and Deri in their most recent evaluations [10, 8], differences can be attributed to our custom kernel configuration and different hardware specifications. The kernel module achieves its high throughput by avoiding all unnecessary system calls and demonstrates the potential gains we can get by running in kernel space.

7. RELATED WORK

The PNA is not the first work to focus on network monitoring. However, most existing monitors that we are aware of focus on quickly filtering network traffic to achieve high-speeds and policy enforcement. Our PNA system is designed to look at every packet to allow high-speed auditing of network traffic.

The common trend in network monitoring has been to focus on filtering the traffic so the monitor can have sufficient time to deal with known protocols. Recently, Sekar et al. have challenged that trend and suggest that a minimalist approach may be as good, if not better [18]. Their approach is to have routers implement a small number of primitives to collect packet traces and evaluate them offline instead of having several distinct monitors competing for resources. Our approach pushes collection off the router entirely, allowing the router to focus on its core function, and put the emphasis on collecting packet summaries followed by monitor specific functionality. We agree with their concerns that focusing effort on developing distinct monitors, while fruitful, may not be the best approach when looking for a holistic view of the network.

7.1 Network Monitoring

The earliest major network monitoring system we are aware of is Bro, which achieves high-speed monitoring by "judiciously leveraging packet-filtering techniques" [17, 13]. Bro is based on `libpcap`, allowing it to be portable across several platforms. `libpcap` is the first step of processing and uses BSD Packet Filters [15] to begin filtering the packet stream; the filtered packets are then passed to an event engine that does some stateful processing. Events can be generated for any number of reasons, usually by an exceptional occurrence, and are passed to a policy interpreter. The policy interpreter uses the `Bro` language to express network policy decisions and checks events against the written policy. Our PNA explicitly avoids filtering packets in the primary stage because we want to provide network administrators with the most comprehensive view of the network possible. Unlike Bro, we sacrifice cross-platform portability by tying the PNA to Linux to gain the high throughput we need.

Another popular network monitoring suite is CoralReef from CAIDA [12]. This system focuses heavily on usability by aggregating distinct traffic source (`libpcap`, traces, device drivers) into a library so application developers can focus on processing not collection. CoralReef provides a consistent API for C, C++, and Perl. In their paper, the authors state that CoralReef provides a superset of the functionality of Bro. Unlike our current system, CoralReef focuses on high level details for processing traffic and avoids concentrating on high speed throughput. We believe our PNA summary output could be converted into an input format, allowing the CoralReef monitoring suite to be used for offline analysis.

Fraleigh et al. present the IPMON system for the Sprint IP backbone [9]. This is a very large scale system deployed at 40 Sprint Points-of-Presence across the continental United States with the goal of collecting and storing header information for every packet in a stream with 5 μs timestamp synchronization. Monitors are provisioned with 330 GB disk arrays which they state will allow capturing of at least several hours worth of trace data at full link utilization. Once the traces are collected (by the disk array filling up or fixed time period elapsing), they are stored to a 10 terabyte storage area network and then are available for offline data analysis. While the IPMON system has very high goals, we believe they are also very costly goals that are not necessary for a typical enterprise level network. Our system also provides real-time monitoring which IPMON does not offer.

Birch provides a “NIC-to-Disk” capture system that captures packets directly to disk and exists entirely in kernel-space [4]. By operating solely in kernel-space, the NIC-to-Disk system is able to realize a drop rate 8.9% less than the user-space equivalent. Like IPMON, the NIC-to-Disk system captures packets directly to disk with no real-time monitoring option.

7.2 High Speed Packet Processing

Beyond the monitoring aspects of our PNA, we also try to process packets as quickly as possible which means receiving the packet, performing our processing steps, and discarding or forwarding the packets. Luca Deri presents the best user-space capture method we know of in his paper describing PF_RING [7]. PF_RING is designed to minimize system call overhead by using a memory mapped ring buffer shared between kernel- and user-space. This allows the user-space application to access packets through a shared buffer—avoiding some system call overhead, memory copying operations, and the standard Linux networking stack. Deri has also written an extension to `libpcap` that allows existing applications to be easily ported to the PF_RING networking option.

Building on top of the PF_RING work previously done, Fusco and Deri [10] create a “Threaded NAPI” (TNAPI), extending the existing Linux networking NAPI to take advantage of multiple hardware receive queues on multi-core systems. Using the TNAPI, a programmer can take advantage of every available core on a machine for packet processing. This distribution across many cores leads to higher packet throughput allowing a PF_RING and TNAPI system to operate at full gigabit speeds. Our PNA currently only uses a single core of a multi-core processor to achieve the throughput discussed in this paper. In the future we plan on pursuing methods to separate processing over multiple cores, allowing increased throughput and monitor complexity. As our approach breaks the typical flow level distribution of allocating flows to different queues and therefore processing core, our focus is on increasing per-core throughput to allow higher total throughput.

Though not focused on developing new methods for high speed packet processing, Braun et al. break down the various user-space packet capturing methods on different hardware and software platforms [5]. Their explorations include comparing FreeBSD to the Linux kernel, comparing PF_PACKET to PF_RING, and comparing Intel to AMD processors.

8. CONCLUSIONS AND FUTURE WORK

This paper has presented our Passive Network Appliance that we developed and deployed on commodity hardware in an enterprise level network. We have:

- Explained our architectural decisions to build a kernel module that automatically provides historical network snapshots that can be analyzed offline to provide insights into a live network.
- Provided an API that allows monitoring traffic at the rate of network frame arrival to enforce or audit policy decisions. We also describe two real-time monitors that track IP-to-IP connections and local IP data.
- Evaluated our PNA system under several worst case scenarios and determined, through a deployed PNA, that we are able to provide tangible benefits to network administrators.
- Gave a comparison of monitoring solutions in kernel-space versus user-space that showed, for a single-core, a kernel-space monitor can outperform the best user-space monitor. This is what allows our PNA to see more packets than existing user-space applications.

With the system infrastructure in place, we have shown that the complete system with real-time monitors is capable of handling 438 Mbps in the worst case (64 byte packets arriving at 1 Gbps each belonging to a unique flow) and over 98% of packets when packets are sized over 256 bytes—all using only a single processing core.

8.1 Future Work

We believe this work still has several interesting avenues for exploration. Our current push is to develop a pipelined version that will be backward compatible with the real-time monitors already developed. With each real-time monitor running on its own processor, more resources will be available to each stage allowing more complexity without reducing the throughput of the system. We expect the throughput of this system to be close to the performance of the “base” kernel module. Once each step works in a pipeline, it may be interesting to explore the cost of moving steps onto network processors that are optimized for fast lookups. Our hope is that some combination of the above ideas and taking better advantage of commodity resources will allow our system get closer to 10 Gbps throughput.

In the longer term, we are working to provide a better support infrastructure for developing real-time monitors. While we find the current programming environment for developing monitor to be a step in the right direction, we believe making a monitor should be easy enough that a network administrator that does not have a background in systems programming should be able to make new monitors tailored to their network.

Acknowledgements

This paper is based upon work supported by the National Science Foundation under Grant No. CNS 0964201.

9. REFERENCES

[1] ArcSight ESM - Enterprise Security Manager. URL `http://www.arcsight.com/products/products-esm/`.

[2] Netflow performance analysis. Technical report, Cisco Systems, Inc., 2007.

[3] G. Antichi, D. J. Miller, and S. Giordano. An open-source hardware module for high-speed network monitoring on netfpga. In *European NetFPGA Developers Workshop*, 2010.

[4] S. W. Birch. Performance characteristics of a kernel-space packet capture module. Master's thesis, Air Force Institute of Technology, 2010.

[5] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle. Comparing and improving current packet capturing solutions based on commodity hardware. In *Internet Measurement Conference*, pages 206–217, 2010.

[6] L. Deri. ntop: Netflow, netflow-lite and sflow based open source network traffic monitoring. URL `http://www.ntop.org/`.

[7] L. Deri. Improving passive packet capture: Beyond device polling. In *International System Administration and Network Engineering Conference*, 2004.

[8] L. Deri. PF_RING, May 2011. URL `http://www.ntop.org/PF_RING.html`.

[9] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-level traffic measurements from the sprint ip backbone. *IEEE Network*, 17(6):6–16, 2003.

[10] F. Fusco and L. Deri. High speed network traffic analysis with commodity multi-core systems. In *Internet Measurement Conference*, pages 218–224, 2010.

[11] J. Gasparakis and J. Peter P Waskiewicz. Design considerations for efficient network applications with intel multi-core processor-based systems on linux. Technical report, Intel Embedded Design Center, 2010.

[12] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, and kc claffy. The architecture of the coralreef internet traffic monitoring software suite. In *Passive and Active Network Measurement Workshop*, 2001.

[13] L. B. N. Laboratory. Bro intrusion detection system. URL `http://www.bro-ids.org/`.

[14] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang. Is sampled data sufficient for anomaly detection? In *Internet Measurement Conference*, 2006.

[15] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX Annual Technical Conference*, 1993.

[16] R. Olsson. pktgen the linux packet generator. In *Ottawa Linux Symposium*, 2005.

[17] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1999.

[18] V. Sekar, M. K. Reiter, and H. Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Internet Measurement Conference*, 2010.

[19] A. Turner. Tcpreplay. URL `http://tcpreplay.synfin.net/`.

[20] C. Wiseman, J. Turner, M. Becchi, P. Crowley, J. DeHart, M. Haitjema, S. James, F. Kuhns, J. Lu, J. Parwatikar, R. Patney, M. Wilson, K. Wong, and D. Zar. A remotely accessible network processor-based router for network experimentation. In *Architectures for Networking and Communications Systems*, pages 20–29, New York, NY, USA, 2008. ACM.