Safety and Performance in an Open Packet Monitoring Architecture

K. G. Anagnostakis[⋆] S. Ioannidis[⋆] S. Miltchev[⋆] J. Ioannidis[†] M. Greenwald[⋆] J. M. Smith[⋆]

[⋆]CIS Department, University of Pennsylvania
{anagnost,sotiris,miltchev,mbgreen,jms}@dsl.cis.upenn.edu

[†]AT&T Labs – Research
ji@research.att.com

Abstract

Packet monitoring arguably needs the flexibility of open architectures and active networking. A significant challenge in the design of open packet monitoring systems is how to effectively strike a balance between flexibility, safety and performance. In this paper we investigate the performance of FLAME, a system that emphasizes flexibility by allowing applications to execute arbitrary code for each packet received. Our system attempts to achieve high performance without sacrificing safety by combining the use of a type-safe language, lightweight run-time checks, and fine-grained policy restrictions. Experiments with our prototype implementation demonstrate the ability of our system to support representative application workloads on Gbit/s links. Such performance indicates the overall efficiency of our approach; more narrowly targeted experiments demonstrate that the overhead required to provide safety is acceptable.

1 Introduction

The bulk of research on *Active Networks* [27] has been directed towards building general infrastructure [1, 29], with relatively little research driven by the needs of particular applications. Recently the focus has shifted slightly as researchers have begun to investigate issues such as safety, extensibility, performance, and resource control, from the perspective of specific applications [3, 22].

Network traffic monitoring is one such application. Originally used just to gather data for basic network research, today network traffic monitoring is important for three other reasons as well. First, ISPs need to analyze patterns of network traffic in order to adequately provision the network infrastructure. Second, the network occasionally finds itself in abnormal situations ranging from distributed denial of service attacks to network routing outages; real-time monitoring can potentially detect such conditions and react promptly. Third, analysis of traffic is needed for accounting and the verification of compliance with diverse policies.

Network traffic monitoring can benefit greatly from a measurement infrastructure with an open architecture. Static implementations of monitoring systems are unable to keep up with evolving demands. The first big problem is that, in many cases, measurement is required at multiple points in the network. No distributed monitoring infrastructure is currently deployed, so measurements must typically take place at the few nodes, such as routers, that already monitor traffic and export their results. Routers offer *built-in* monitoring functionality. Router vendors only implement measurement functions that are cost-effective: measurements that are interesting to the vast majority of possible customers. If one needs measurements that are not part of the common set, then there is may be no way to extract the needed data from the routers. Furthermore, as customer interests evolve, the router vendors can only add measurement functionality on the time-scale of product design and release; it can be months or years from the time customers first indicate interest until a feature makes it into a product. The second big problem is that most monitoring functionality is only accessible using mechanisms such as SNMP[6], RMON[28] or NetFlow[8]. Even if a particular router supports the needed measurements, the management interfaces offered by these mechanisms are fixed, and may fall short of satisfying user needs that were not anticipated at design time. Finally, the need for timely deployment cannot always be met at the current pace of standardization or software deployment, especially in cases such as detection and prevention of denial-of-service attacks.

In response to these problems, several prototype extensible monitoring systems [16, 3, 2] have been developed with the goal of providing the needed flexibility, building on open architecture and active networking concepts. The basic goal of such approaches is to allow the use of critical system components by users other than the network operator. However, providing users with the ability to run their own modules on nodes distributed throughout the network requires extensible monitoring systems to provide protection mechanisms.

Flexible protection mechanisms, and other methods of enforcing safety, are an essential part of the architecture of any extensible measuring system for two reasons. First, users, such as researchers who want to study network behavior, should not have access to all the data passing through a router. Rather, fine-grained protection is needed to allow the system to enforce policy restrictions, *e.g.,* ensuring privacy by limiting access to IP addresses, header fields, or packet content. Second, protection from interference is needed to guard against poorly implemented modules which could otherwise hurt functions that may be critical to the operation of the network infrastructure.

The thrust of our research is to determine whether programmable traffic monitoring systems that are flexible enough to be useful, and safe enough to be deployed, can perform well enough to be practical.

In LAME [3] we demonstrated that it is possible to build an extensible monitoring system using off-the-shelf components. Further investigation demonstrated performance problems with the use of off-the-shelf components in LAME. Our follow-on project, FLAME, presented a design that preserved the safety properties of LAME, but was designed for high performance. FLAME combines several well-known mechanisms for protection and policy control; in particular, the use of a type-safe language, custom object patches for run-time checks, anonymizing, and namespace protection based on trust management. In [2] we presented preliminary results that demonstrated that FLAME largely eliminated the performance problems of LAME.

The purpose of the study in this paper is to understand the range of applications and traffic rates for which a safe, open, traffic monitoring architecture is practical. We have implemented a number of test applications and have used them as our experimental workload. We use the data collected from these applications to quantify and analyze the performance costs, and to predict the workload at which our system will no longer be able to keep up with incoming traffic.

The rest of this paper is structured as follows. We present the system architecture, including protection mechanisms, in Section 2. In Section 3 we study the performance trade-offs of the resulting system, and we conclude in Section 4.

2 System architecture

The architecture of FLAME is shown in Figure 1. A more detailed description is available in [2]. Modules consist of kernel-level code K_x, user-level code U_x, and a set of credentials C_x. Module code is written in Cyclone [10] and is processed by a trusted compiler upon installation. The kernel-level code takes care of time-critical packet processing, while the user-level code provides additional functionality at a lower time scale and priority. This is needed so applications can communicate with the user or a management system (*e.g.,* using the standard library, sockets, *etc.*).

There has been a small architectural modification to FLAME since the publication of [2], after ex-

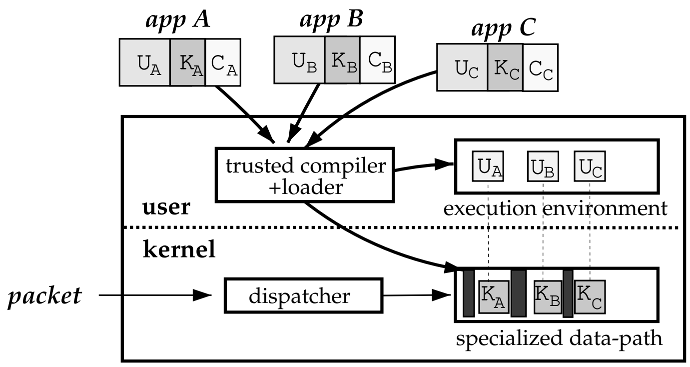

Figure 1: **FLAME Architecture**

perimentation under high load. The original FLAME architecture interacted with the network interface exclusively through interrupts. As others have noted [17], under high rates of incoming network traffic, interrupt handling can degrade performance. More recent versions of FLAME poll the network interface card (NIC) to read packets to avoid performance degradation. Note that the polling technique and the resulting performance improvement is well known and does not represent a contribution of this paper.

In terms of deployment, the system can be used as a passive monitor *e.g.* by tapping on a network link by means of an optical splitter, or using *port mirroring* features on modern switches. Ideally, a FLAME -like subsystem would be part of an enhanced router interface card. For the purposes of this paper, we consider FLAME in a passive monitor set-up.

Our current system does not attempt to explore extending resource management of user processes. A number of solutions have already been proposed for extending the Unix security model, *e.g.* [14]. For now, user-space modules run as normal Unix processes.

We detail how safe execution of in-kernel code is accomplished in Section 2.1. The basic approach is to use the set of credentials, C_x, at compile time to verify that the module is allowed by system policy to perform the functions it requests. The dark units in Figure 1 beside each K_x represent code that is inserted before each module code segment for enforcing policy-related restrictions. These units appropriately restrict access of modules to packets or packet fields, provide selective anonymization of fields, and so on.

2.1 In-kernel Safe Execution

There are a number of major design challenges for allowing user code to execute inside the operating system kernel: the system needs to guard against excessive execution time, privileged instructions, exceptions and random memory references. There has been extensive work in the operating system and language communities that addresses the above problems [10, 25, 21, 7, 30]. FLAME leverages these techniques to satisfy our security needs.

Bounding Execution Time. A simple method for bounding execution time is eliminating backward jumps [11, 19]. This has the advantage of providing us with an upper bound for the execution time: linear in the length of the program. However, such a programming model is rather limiting and hard to program in. Another approach executes each installed module as a kernel thread and context switches between threads when they exceed their allocated time slot. Unfortunately, this is too heavy weight for round-robin execution of monitoring functions on incoming packets. We take a different approach, similar to [12]: we augment the backward jumps with checks to a cycle counter; if the module exceeds its allocated execution time we jump to the next module. On the next invocation,the module can consult an appropriately set environment variable to check if it needs to clean-up data or exit with an error. This method adds an overhead of 5 assembly instructions for the check. If the check succeeds there is an additional overhead of 6 instructions to initiate the jump to the next module.

Exceptions. We modified the trap handler of the operating system to catch exceptions originating from the loaded code. Instead of causing a system panic we terminate the module and continue with the following one.

Privileged Instructions and Random Memory References. We use Cyclone [10] to guard against instructions that may arbitrarily access memory locations or may try to execute privileged machine instructions. Cyclone is a language for C programmers who want to write secure, robust programs. It is a dialect of C designed to be *safe*: free of crashes, buffer overflows, format string attacks, and so on. All Cyclone programs must pass a combination of compile-time, link-time and run-time checks to ensure safety.

2.2 Policy control

Before installing a module in our system we perform policy compliance checks[1] on the credentials this module carries. The checks determine the privileges and permissions of the module. In this way, the network operator is able to control what packets a module can access, what part of the packet a module is allowed to view and in what way, what amount of resources (processing, memory, *etc.*) the module is allowed to consume on the monitoring system, and what other functions (*e.g.,* socket access) the module is allowed to perform.

3 Experiments

This section describes a number of applications that we have implemented on FLAME and then presents three sets of experiments. The first involves the deployment of the system in a laboratory testbed, serving as a proof of concept. The second looks at issues of the underlying infrastructure, in order to specify the *capacity of our system* on Gbit/s links. The third set of experiments provides a picture of the processing cost of our example applications, and protection overheads.

3.1 Applications

We have implemented several applications to aid in the design as well as to demonstrate the flexibility and performance of our system.[2] The applications presented here have been chosen based on two criteria. First, we chose applications that are not currently available, but are expected to be useful, making them likely candidates for deployment using a system such as FLAME. Second, we focused on applications that go beyond network research to functions that are of interest to network operators.

Trajectory sampling. Trajectory sampling, developed by Duffield and Grossglauser[9], is a technique for coordinated sampling of traffic across multiple measurement points, effectively providing information on the spatial flow of traffic through a network. The key idea is to sample packets based on a hash function over the invariant packet content (*e.g.* excluding fields such as the TTL value that change from hop to hop) so that the same packet will be sampled on all measured links. Network operators can use this technique

[1] Our policy compliance checker uses the KeyNote [4] system.

[2] The set of applications presented in [2] were mostly different. The only application studied in both this paper and in [2] is the worm detection module.

to measure traffic load, traffic mix, one-way delay and delay variation between ingress and egress points, yielding important information for traffic engineering and other network management functions. Although the technique is simple to implement, we are not aware of any monitoring system or router implementing it at this time.

We have implemented trajectory sampling as a FLAME module that works as follows. First, we compute a hash function $h(x) = \phi(x) \mod A$ on the invariant part $\phi(x)$ of the packet. If $h(x) > B$, where $B < A$ controls the sampling rate, the packet is not processed further. If $h(x) < B$ we compute a second hash function $g(x)$ on the packet header that, with high probability, uniquely identifies a flow with a label (*e.g.* TCP sequence numbers are ignored at this stage). If this is a new flow, we create an entry into a hash table, storing flow information (such as, IP address, protocol, port numbers *etc.*). Additionally, we store a timestamp along with $h(x)$ into a separate data structure. If the flow already exists, we do not need to store all the information on the flow, so we just log the packet. For the purpose of this study we did not implement a mechanism to transfer logs from the kernel to a user-level module or management system; at the end of the experiment the logs are stored in a file for analysis.

Round-trip time analysis. We have implemented a simple application for measuring an approximation of round-trip delays observed by TCP connections passing through a network link. The round-trip delays experienced by users is an important metric for understanding end-to-end performance, mostly due to its central role in TCP congestion control[15]. Additionally, measuring the round-trip times observed by users over a specific ISP provides a reasonable indication of the quality of the service provider's infrastructure, as well as its connectivity to the rest of the Internet. Finally, observing the evolution of round-trip delays over time can be used to detect network anomalies on shorter time scales, or to observe the improvement (or deterioration) of service quality over longer periods of time. For example, an operator can use this tool to detect service degradation or routing failures in an upstream provider, and take appropriate measures (*e.g.,* redirecting traffic to a backup provider) or simply have answers for user questions.

The implementation of this application is fairly simple and efficient. We watch for TCP SYN packets indicating a new connection request, and watch for the matching TCP ACK packet in the same direction. The difference in time between the two packets provides a reasonable approximation of the round-trip time between the two ends of the connection.[3] For every SYN packet received, we store a timestamp into a hashtable. As the first ACK after a SYN usually has a sequence number which is the SYN packet's sequence number plus one, this number is used as the key for hashing. Thus, in addition to watching for SYN packets, the application only needs to look into the hash table for every ACK received. The hashtable can be appropriately sized depending on the number of flows and the required level of accuracy. A different algorithm that computes both RTTs and RTOs, but is significantly more complex and is not appropriate for real-time measurement, as well as an alternative, wavelet-based method are described in [13]. Note that this algorithm does not work for parallel paths where SYN and ACK may be forwarded on different links. Retransmission of the SYN packet does not affect measurement, as the timestamp in the hashtable will be updated. Retransmission of an ACK packet introduces error when the first ACK is not recorded. If this happens rarely, then this error does not affect the overall RTT statistics. If hapenning frequently, due to a highly congested link, this will be reflected in the overall statistics, and should be interpreted accordingly (there will be a cluster of samples around typical TCP Timeout values).

[3]Factors such as operating system load on the two end-points can introduce error. We do not expect these errors to distort the overall picture significantly, at least for the applications discussed here. These applications take statistics over a number of samples, so individual errors will not significantly alter the result. In fact, individually anomalous samples can be used to indicate server overload or other phenomena.

Worm detection. The concept of "worm" and techniques to implement them have existed since the early descriptions in[5, 26]. A worm compromises a system such as a Web server by exploiting system security vulnerabilities; once a system has been compromised the worm attempts to replicate by "infecting" other hosts. Recently, the Internet has observed a wave of "worm" attacks[18]. The "Code Red" worm and its variants infected over 300,000 servers in July-August 2001.

This attack can be locally detected and prevented if the packet monitor can obtain access to the TCP packet content. Unfortunately, most known packet monitors only record the IP and TCP header and not the packet payload. We have implemented a module to scan packets for the signature of one strain of "Code Red" (the random seed variant):

```
... GET /default.ida?NNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNN .....
```

If this signature is matched, the source and destination IP addresses are recorded and can be used to take further action (such as blocking traffic from attacking or attacked hosts *etc.*). Despite the ability to locally detect and protect against worms, widespread deployment of an extensible system such as FLAME would still have improved the fight against the virus.

It is worth noting that the "Code Red" worm attacked the Internet by exploiting a security bug less than 4 weeks after the bug was first discovered. The worm attacked over 300,000 hosts within a brief period after it was first launched. Only the most supple virus detection systems are likely to be able to respond promptly enough to have shut down this threat. While most intrusion detection systems do provide rule-based extensibility, it is unlikely, had code-red been more malicious, that the correct rules could have been applied on time.

On the other hand, we know of a mechanism that is able to deliver virus defenses at least as fast as the worm — another worm. A *safe* open architecture system can allow properly authenticated worms (from, say, CERT) to spread the defense against a malicious worm. In the future, detecting a worm may not be as simple as searching for a fixed signature, and more complicated detection and protection programs may require the flexibility of programmable modules.

Finally, providing a general-purpose packet monitoring system is likely to reduce cost due to the shared nature of the infrastructure, increase impact by coupling the function with network management (to allow, for example, traffic blocking) and result in more wide-spread deployment and use of such security mechanisms.

Real-time estimation of long-range dependence parameters. Roughan *et al.* [24] proposed an efficient algorithm for estimating long-range dependence parameters of network traffic in real-time. These parameters directly capture tha variability of network traffic and can be used, beyond research, for purposes such as measuring differences in variability between different traffic/service classes or characterizing service quality. We have ported the algorithm to Cyclone and implemented the appropriate changes to allow execution as a module on the FLAME system. Some modifications were needed for satisfying Cyclone's static type checker and providing appropriate input, *e.g.,* traffic rates over an interval. The primary difference between this module and the other applications is that it needed to have separate kernel and user space components. This requirement arises because the algorithm involves two loops: the inner loop performs lightweight processing over a number of samples, while the the outer loop performs a more computationally intensive task of taking the results and producing the estimate. As the system cannot interrupt the kernel module and provide scheduling, the outer loop had to be moved to user space.

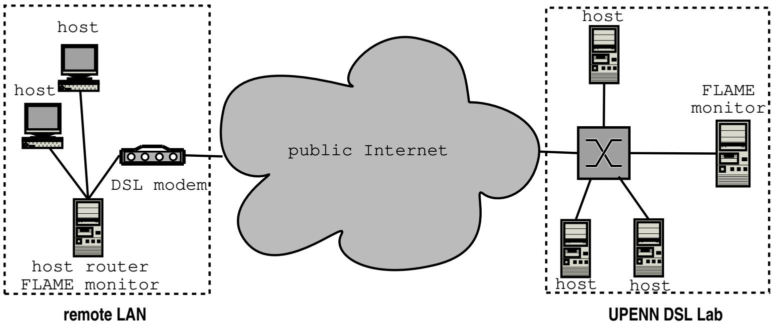

Figure 2: **Network configuration used for experiments**

3.2 Experiment setup

Given the distributed nature of the trajectory sampling and round-trip delay analysis applications, the testbed used for our experiments is shown in Figure 2 and involves two sites: a local test network at the University of Pennsylvania, and a remote LAN connecting to the Internet through a commercial ISP using a DSL link. The mean round-trip delay between the two sites is 24 ms. The test network at Penn consists of 4 PCs connected to an Extreme Networks Summit 1i switch. The switch provides port mirroring to allow any of its links to be monitored by the FLAMEsystem on one of the PCs. All PCs are 1 GHz Intel Pentium III with 512 MB memory, OpenBSD 2.9 operating system except for the monitoring capacity experiments where we used the Click [20] code under Linux 2.2.14 on the sending host. The FLAME system, as well as the host shown above the switch use the Intel PRO/1000SC Gigabit NIC. The hosts below the switch in Figure 2 have Alteon Acenic Gigabit NICs. The host router in the remote LAN is a 1 GHz Intel Pentium III and connects several hosts using Fast Ethernet.

3.3 Testbed demonstration

In this section we demonstrate the use of the round-trip delay analysis and trajectory sampling modules on our experimental setup. We have installed the round-trip delay analysis module on the two FLAME monitors, on the remote LAN and the PENN DSL test network. We initiated `wget` to recursively fetch pages, starting from the University of Pennsylvania main web server. In this way we created traffic to a large number of sites reachable through links on the starting Web page. The experiment was started concurrently on both networks to allow us to compare the results. One particular view of 5374 connections over a 1 hour period is presented in Figure 3, clearly showing the difference in performance which is in part due to the large number of local or otherwise well connected sites that are linked through the University's Web pages.

We also executed the trajectory sampling module and processed the data collected by the module to measure the one way delay for packet flowing between the two networks. The clocks at the two monitors were synchronized using NTP prior to the experiment. Note that appropriate algorithms for removing clock synchronization error, such as those described in [23], can improve the accuracy of such measurements in operational use. However, for the sake of illustrating the use of applications of our system this problem was not further addressed. The results are shown in Figure 4. Note that this is different from simply using ping to sample delays, as we measure the actual delay experienced by network traffic. The spikes show our attempts to overload the remote LAN using UDP traffic.

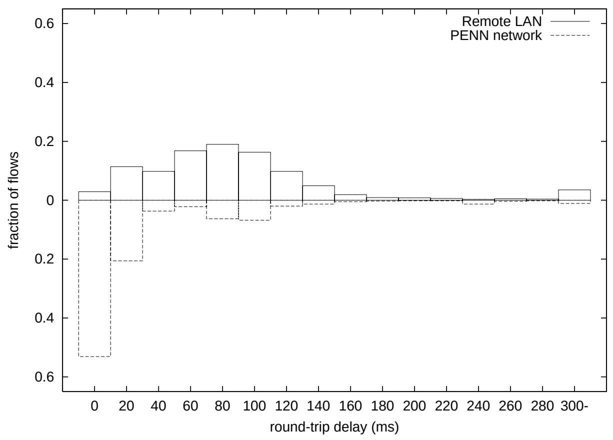

Figure 3: **Histogram for round-trip delays for the same targets, as seen from two different networks**

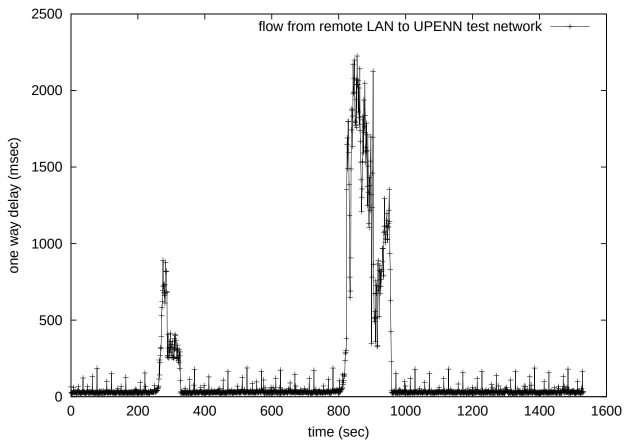

Figure 4: **Measuring one way delay between two networks**

3.4 System monitoring capacity

The purpose of this experiment is to measure how many processing cycles are available for FLAME to execute application code for different traffic rates. Two sender PCs generate traffic to one sink, and the switch was configured to mirror the sink port to the FLAME system. The device driver on the FLAME system was modified to disable interrupts and the FLAME system was instrumented to use polling for reading packets off the NIC. To generate traffic at different rates, we used the Click modular router system under Linux on the sending side. All experiments reported use 64 byte UDP packets. The numbers were discovered by inserting an idle loop into a null monitoring module consuming processing cycles and adapting the sending rate until no packets are dropped at the monitor. In Figure 5 we show the number of processing cycles available at different traffic rates, for FLAME without polling, and for FLAME with polling enabled. There are two main observations to make. First, the polling-based monitor performs significantly better as the packet rates increase. Second, the number of cycles available, even at high packet rates is reasonable enough, as we will discuss later in the analysis of the processing costs of different applications. Note that for a median packet size of 250 bytes, which is typical in the Internet, the packet rate would be approximately 500,000 packets per second. With the 1 GHz Pentium used for our measurements this would not provide a lot of headroom for monitoring applications. Note, that the 1 GHz configuration does not represent an upper bound on the technology curve: higher speed processors are already available and will likely continue to improve in speed. In the short term, we will thus be able to support a reasonable application workload even at fully loaded Gbit/s links. Of course, using FLAME on higher network speeds (*e.g.* 10 Gbit/s and more) is not currently practical and is outside the scope of our work.

3.5 Workload analysis and protection overhead

The application modules were instrumented using the Pentium performance counters, to obtain an accurate indication of the processing cost for each of our applications. We read the value of the Pentium cycle counter before and after execution of application code for each packet. In lack of representative traffic on our laboratory testbed, we fed the system with packets using a packet trace from the Auckland-II trace archive provided by NLANR and the WAND research group at the University of New Zealand. This is especially important as the processing cost for each application depends on data such as, for instance, IP addresses and flow arrival vs. overall traffic rate. The measurements were taken on a 1 GHz Intel Pentium III with 512 MB memory, OpenBSD 2.9 operating system, gcc version 2.95.3, and Cyclone version 0.1.2.

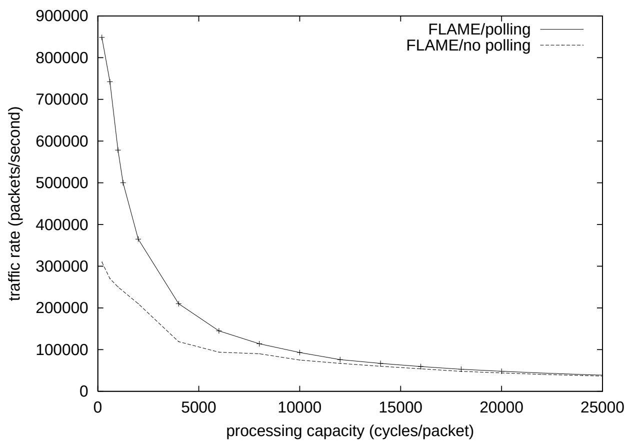

Figure 5: **Processing capacity vs. traffic rate, for FLAME with and without polling enabled.**

	gcc	Cyclone	Cyclone protection	Cyclone protection optimized
Traj. Sampling	381	420 (+10.2%)	458 (+20.2%)	430 (+12.8%)
RTT analysis	183	209 (+12.4%)	211 (+15.3%)	211 (+15.3%)
Worm detection	24	44 (+83.3%)	54 (+125%)	44 (+83.3%)
LRD estimation	143	154 (+7.6%)	158 (+10.4%)	156 (+9%)

Table 1: **Module processing costs (in cycles) and protection overheads.**

We compare the processing cost of a pure C version of each application to the Cyclone version, with and without protection, and using additional optimizations to remove or thin the frequency of backward jumps (these modifications were done by hand). We measure the median execution time of each module, averaged over 113 runs. The results from this experiment are summarized in Table 1. There are four main observations to make. First, considering also the results presented in the previous section, it appears that the cost per-application is well within the capabilities of a modern host processor, for a reasonable spectrum of traffic rates. Second, the cost of protection (after optimization), while not insignificant, does not exceed by far the cost of an unprotected system. Third, the costs presented are highly application dependent and may therefore vary. Finally, some effort was spent in increasing the efficiency of both the original C code as well as the Cyclone version; implementing monitoring modules requires performance-conscious design and coding. Thus, this experiment should only be used as a (positive) indication that providing protection mechanisms, and hereby opening the monitoring architecture to experimental applications, and untrusted users, is within realistic bounds.

4 Summary and Concluding Remarks

We have spent some time building, measuring, and refining an open architecture for network traffic monitoring. Several interesting observations are worth reporting:

The techniques developed to build general infrastructure are applicable and portable to specific applications. LAME was built using off-the-shelf components. FLAME, in contrast, required us to write custom code. However, it was constructed using "off-the-shelf technology". That is, the techniques we used for extensibility, safety, and efficiency were well-known, and had already been developed to solve the same problems in a general active-networking infrastructure. In particular, the techniques used for open architectures are now sufficiently mature that applications can be built by importing technology, rather than by solving daunting new problems.

Nevertheless, *careful design is still necessary.* Although the technology was readily available, our system has gone through three architectural revisions, as we discovering that each version had some particular performance problems. Care must be taken to port the *right* techniques and structure, otherwise the price in performance paid for extensibility and safety may render the application impractical.

Programmable applications are clearly more flexible than their static, closed, counterparts. However, to the limited extent that we have been able to find existing custom applications supporting similar functionality, we found that *careful engineering can make applications with open architectures perform competitively with custom-built, static implementations.*

More experience building applications is certainly needed to support our observations, but our experience so far supports the fact that high performance open architecture applications are practical.

References

[1] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare active network architecture. *IEEE Network*, 12(3):29–36, May/June 1998.

[2] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, M. B. Greenwald, and J. M. Smith. Efficient packet monitoring for network management. In *Proceedings of IEEE NOMS 2002 (to appear, earlier extended draft available as UPENN-TR-01-28)*, April 2002.

[3] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, and J. M. Smith. Practical network applications on a leightweight active management environment. In *Proceedings of the 3rd International Working Conference on Active Networks*, October 2001.

[4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust Management System Version 2. Internet RFC 2704, September 1999.

[5] J. Brunner. *The Shockwave Rider*. Del Rey Books, Canada, 1975.

[6] J. D. Case, M. Fedor, M. L. Schoffstall, and C. Davin. Simple Network Mangement Protocol (SNMP). RFC1157/STD0015, http://www.rfc-editor.org/, May 1990.

[7] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 80–85, 1993.

[8] Cisco Corporation. NetFlow services and applications. http://www.cisco.com/, 2000.

[9] N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proc. ACM SIGCOMM'00 Conference*. August 2000.

[10] G. M. et al. Cyclone: A next-generation systems language. In *http://www.cs.cornell.edu/projects/cyclone*, 2001.

[11] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proceedings of the Internation Conference on Function Programming (ICFP)*, September 1998.

[12] M. Hicks, J. T. Moore, and S. Nettles. Compiling plan to snap. In *Proceedings of the 3rd International Working Conference on Active Networks*, October 2001.

[13] P. Huang, A. Feldmann, and W. Willinger. A non-intrusive, wavelet-based approach to detecting network performance problems. In *ACM SIGCOMM Internet Measurement Workshop 2001*, November 2001.

[14] S. Ioannidis and S. M. Bellovin. Sub-Operating Systems: A New Approach to Application Security. Technical Report MS-CIS-01-06, University of Pennsylvania, February 2001.

[15] T. V. Lakshman and U. Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking*, 5(3), 1997.

[16] G. R. Malan and F. Jahanian. An Extensible Probe Architecture for Network Protocol Performance Measurement. In *ACM SIGCOMM'98*, 1998.

[17] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[18] D. Moore. The spread of the code-red worm (crv2). In *http://www.caida.org/analysis/security/code-red/*. August 2001.

[19] J. T. Moore, M. Hicks, and S. Nettles. Practical programmable packets. In *Proc. INFOCOM 2001*, April 2001.

[20] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Symposium on Operating System Principles*, pages 217–231, 1999.

[21] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, May 1999.

[22] C. Partridge, A. Snoeren, T. Strayer, B. Schwartz, M. Condell, and I. Castineyra. FIRE: Flexible Intra-AS Routing Environment. In *Proc. ACM SIGCOMM'00 Conference*, pages 191–203. August 2000.

[23] V. Paxson. On calibrating measurements of packet transit times. In *Proc. 1998 ACM SIGMETRICS Conference*, 1998.

[24] M. Roghan, D. Veitch, and P. Abry. Real-time estimation of the parameters of long-range dependence. *IEEE/ACM Transactions on Networking*, 8(4):467–478, August 2000.

[25] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. *Informatics: 10 Years Back, 10 Years Ahead*, pages 86–101, 2000.

[26] J. F. Shoch and J. A. Hupp. The 'worm' programs – early experiments with a distributed computation. *Communications of the ACM*, 25(3):172–180, March 1982.

[27] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE Communications Magazine*, pages 80 – 86, January 1997.

[28] S. Waldbusser. Remote Network Monitoring Management Information base. RFC2819/STD0059, http://www.rfc-editor.org/, May 2000.

[29] D. Wetherall. Active Network Vision and Reality: Lessons from a Capsule-based System. In *Proceedings of the 17th ACM Symposium on Operating System Principles, Kiawah Island, SC*, Dec. 1999.

[30] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous rpc: Low-latency protection in a 64-bit address space. In *Proceedings of 1993 Summer USENIX Conference*, June 1993.

Appendix A

```
KeyNote-Version: 2
Authorizer: NET_MANAGER
Licensees: TrafficAnalysis
Conditions:
  (app_domain == "flame" && module == "capture" &&
       (IPsrc  == 158.130.6.0/24 || IPdst  == 158.130.6.0/24))
    -> "HEADERS-ONLY";
 Signature: "rsa-md5-hex:f00f5673"
```

Figure 6: **Example credential that grants "TrafficAnalysis" the right to capture traffic to/from 158.130.6.0. The user is authorized to receive packet headers only.**