Open Packet Monitoring on FLAME: Safety, Performance, and Applications*

Kostas G. Anagnostakis, Michael Greenwald, Sotiris Ioannidis, and Stefan Miltchev

CIS Department, University of Pennsylvania
200 S. 33rd Street, Philadelphia PA 19104, USA
{anagnost,mbgreen,si,miltchev}@dsl.cis.upenn.edu

Abstract. Packet monitoring arguably needs the flexibility of open architectures and active networking. In earlier work we have implemented FLAME, an open monitoring system, that balanced flexibility and safety while attempting to achieve high performance by combining the use of a type-safe language, lightweight run-time checks, and fine-grained policy restrictions.
We seek to understand the range of applications, workloads, and traffic, for which a safe, open, traffic monitoring architecture is practical. To that end, we investigated a number of applications built on top of FLAME. We use measurement data and analysis to predict the workload at which our system cannot keep up with incoming traffic. We report on our experience with these applications, and make several observations on the current state of open architecture applications.

1 Introduction

The bulk of research on *Active Networks* [23] has been directed towards building general infrastructure [1, 24], 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 [4, 18].

Network traffic monitoring is one such application. [3, 4] makes the case that network traffic monitoring can benefit greatly from a monitoring infrastructure with an open architecture, as static implementations of monitoring systems are unable to keep up with evolving demands. The first big problem is that, in many cases, monitoring is required at multiple points in the network. No distributed monitoring infrastructure is currently deployed, so monitoring must typically take place at the few nodes, such as routers, that already monitor traffic and export their results. While routers do offer *built-in* monitoring functionality, router vendors only implement monitoring functions that are cost-effective: those that

* This work was supported in part by the DoD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795, and by NSF under grant ANI-00-82386.

J. Sterbenz et al. (Eds.): IWAN 2002, LNCS 2546, pp. 120–131, 2002.

are interesting to the vast majority of possible customers. If one needs functions that are not part of the common set, then there may be no way to extract the needed data from the routers. Furthermore, as customer interests evolve, the router vendors can only add monitoring 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. Therefore, 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 [14, 4, 3, 11] have been developed. One 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 essential for extensible monitoring systems 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 (or malicious) 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 [4] 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.

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. In [3] we presented preliminary results that demonstrated that FLAME largely eliminated the performance problems of LAME. We have implemented a number of additional 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 general tenor of the results reported here (although not the specific numbers) should be more widely applicable than just to FLAME. For example, the

Open Kernel Environment (OKE) of Bos and Samwel [6] adopts a similar approach to FLAME. The OKE designers also carefully considered the interaction between safety features and performance implications. OKE, among other features, provides additional flexibility through the use of trust-controlled elastic language extensions. These extensions provide increased control over the trade-offs between safety and performance, as, for example, certain checks which are hard-wired in our design can be eliminated, if appropriate trust credentials are provided. The work reported in this paper should give some indications about the workload supportable by systems such as OKE, also.

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

2 Overview of the FLAME Architecture

The architecture of FLAME is shown in Figure 1. A more detailed description is available in [3]. 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 [12] 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 [3], after experimentation under high load. The original FLAME architecture interacted with the network interface exclusively through interrupts. As others have noted [15, 22], 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. A preliminary study shows how such a subsystem can be built using a network processor board [2]. For the purposes of this paper, we consider FLAME in a passive monitor set-up.

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.

For allowing user code to safely execute inside the operating system kernel, the system needs to guard against excessive execution time, privileged instruc-

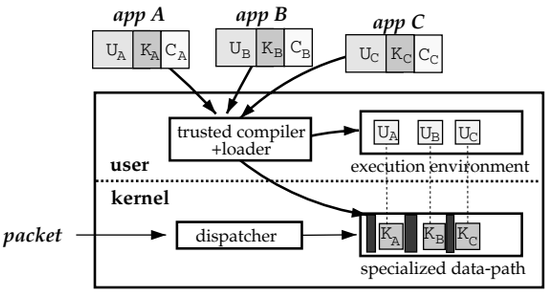

Fig. 1. FLAME Architecture

tions, exceptions and random memory references. There has been extensive work in the operating system and language communities that addresses the above problems (c.f. [20, 8, 25]). FLAME leverages these techniques to satisfy our security needs.

Bounding Execution Time. For bounding execution time we take an approach similar to [10]: 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 [12] 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.

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.

[1] Our policy compliance checker uses the KeyNote [5] system.

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 present examples of applications that a) are widely regarded as useful but appear to be stalled in the standardization process (trajectory sampling), b) would be difficult to deploy in time to be useful (worm detection) and c) may be valuable in certain situations but may not be globally useful to make it worth implementing in routers (RTT analysis, LRD analysis).

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 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.

Round-trip time analysis. We have implemented a simple application for obtaining an approximation of round-trip delays for TCP connections passing through a link. The round-trip delay is an important metric for understanding end-to-end performance due to its role in TCP congestion control [13]. Additionally, measuring the round-trip times observed 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 variation in service quality over longer periods of time.

The implementation is both 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 events provides a reasonable approximation of the round-trip time between the two ends of the connection. For every SYN packet received, we store a timestamp into a hash-table. 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 desired level of accuracy.

Worm detection. The concept of "worms" and techniques to implement them have existed since the early descriptions in [7, 21]. 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 [16]. 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). 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.

Real-time estimation of long-range dependence parameters. Roughan *et al.* [19] proposed an efficient algorithm for estimating long-range dependence parameters of network traffic in real-time. These parameters directly capture the variability of network traffic and can be used, beyond research, for purposes such as measuring differences in variability between different traffic classes and 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 separate kernel and user space components were needed. This requirement arises because the algorithm involves two loops: the inner loop performs lightweight processing over a number of samples, while the outer loop performs the more computationally intensive task of taking the results and producing the estimate. As the system

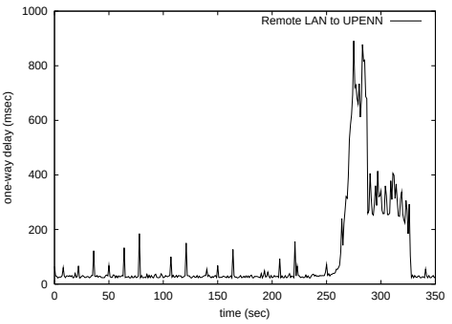

Fig. 2. Measuring one way delay between two networks.

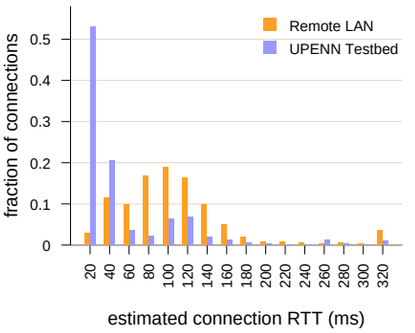

Fig. 3. Histogram for RTT estimates for the same targets seen from two networks.

cannot interrupt the kernel module and provide scheduling, the outer loop had to be moved to user space.

3.2 Experiment Setup

The testbed used for our experiments involves two sites: a local test network at Penn, and a remote LAN connecting to the Internet through a DSL link. The minimum 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 FLAME system 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 [17] code under Linux 2.2.14 on the sending host. The FLAME system uses the Intel PRO/1000SC Gigabit NIC.

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 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 partly due to the large number of local or otherwise well connected sites that are linked through the University's Web pages.

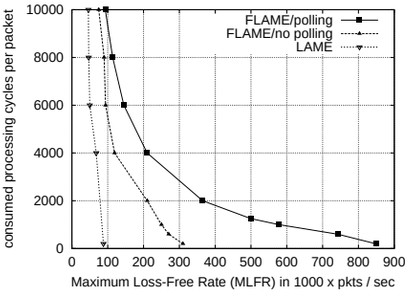

Fig. 4. Available processing cycles per packet as a function of maximum loss-free traffic rate.

Module	gcc	Cyclone	Cyclone protection	Cyclone protection optimized
Traj.smpl.	381	10.2%	20.2%	12.8%
RTT est.	183	12.4%	15.3%	15.3%
Worm det.	24	83.3%	125%	83.3%
LRD est.	143	7.6%	10.4%	9%

Fig. 5. Module processing costs (in cycles), Cyclone overhead, FLAME protection overhead, and optimization effect.

We also executed the trajectory sampling module and processed the data collected by the module to measure the one way delay for packets flowing between the two networks. The clocks at the two monitors were synchronized using NTP prior to the experiment. The results are shown in Figure 2. Note that this is different from simply using `ping` to sample delays, as we measure the *actual* delay experienced by network traffic. The spike shows our attempt to overload the remote LAN using UDP traffic.

3.4 System Performance, Workload Analysis, and Safety Overheads

We determine how many processing cycles are available for executing monitoring applications at different traffic rates. We report on the performance of FLAME with and without the interface polling enhancement as well as LAME.

The experiment is designed as follows. Two sender PCs generate traffic to one sink, with the switch configured to mirror the sink port to the FLAME monitor. The device driver on the FLAME system is modified to disable interrupts and the FLAME system is instrumented to use polling for reading packets off the NIC. To generate traffic at different rates, we use the Click modular router system under Linux on the sending side. All experiments involve 64 byte UDP packets. The numbers are determined by inserting a busy loop into a null monitoring module consuming processing cycles. The sending rate is adapted downward until no packets are dropped at the monitor. This may seem overly conservative, because packet losses occur when even one packet is delivered to FLAME too early. However, the device driver allocates 256 RxDescriptors for the card to store 2K packets. Therefore the card can buffer short-term bursts that exceed the average rate without incurring packet loss, but cannot tolerate sustained rates above the limit. In Figure 4 we show the number of processing cycles available at different traffic rates, for LAME, FLAME without polling, and FLAME with polling enabled.

There are two main observations to make on these results. First, as expected, the polling system performs significantly better, roughly 2.5 times better than

the non-polling system. Second, the number of cycles available for applications to consume, even at high packet rates, appears reasonable. In the next sections we will discuss these figures in light of the processing needs of our experimental applications.

To obtain an rough estimate of the processing cost for each application, we instrumented the application modules using the Pentium performance counters. We read the value of the Pentium cycle counter before and after execution of application code for each packet. Due to lack of representative traffic on our laboratory testbed, we fed the system with packets using the Auckland-II packet trace provided by NLANR and the WAND research group. 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.

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 over 113 runs. The results from this experiment are summarized in Table 5.

There are four main observations to make. First, the cost per-application appears to be well within the capabilities of a modern host processor, for a reasonable spectrum of traffic rates. Second, the cost of protection (after optimization), 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 Thus, care must be taken not to overstate these results. This experiment *does* indicate that it is feasible to provide protection mechanisms in an open monitoring architecture, enabling support for experimental applications and untrusted users. However, the numbers should not be considered representative of off-the-shelf compilers and/or carelessly designed applications.

3.5 Modeling Supportable Workloads and Traffic Rates

We can roughly model the expected performance (maximum supportable packet rate) of FLAME as a function of workload (number of active modules). We derive the model from our measured system performance from Section 3.4, and the costs of our experimental applications and the measured safety overheads from Section 3.4.

We can approximately fit the number of available cycles to $a_0 r^{b_0}$, where r is transmission rate in packets per second and a_0 and b_0 are constants. Computing a_0 and b_0 using least squares, and dropping the data point at 848k packets per second[2], we get the number of available cycles for processing is $3 \times 10^9 r^{-1.1216}$.

[2] The fit is remarkably good for packet rates under 500,000 packets per second. The fit is good for packet rates up to about 800,000 packets per second, but our measurements when the gigabit network was running full bore sending 64 byte packets (small), yielded fewer available cycles than predicted by our model.

Packets per second, r, can itself be computed as $B/8s$ where B is the transmission rate in bits per second, and s is the mean packet size in bytes. Assuming a mean module computation cost of 210 cycles per module (based on the assumption that our applications are representative), and using our measured overhead of 60 cycles per module, we can support a workload of $\lfloor \frac{1}{9}10^8 r^{-1.1216} \rfloor$ modules for an incoming traffic rate of r packets per second, without losing a single packet. Conversely, we can compute the maximum traffic rate as a function of the number of available cycles, c, by $r = 2.816 \times 10^8 c^{-0.8916}$ (or $r = 1.914 \times 10^6 n^{-0.8916}$, where n is the number of modules).

To apply this model on an example, consider a fully-utilized 1 Gbit/s link, with a median packet size of 250 bytes, which is currently typical for the Internet. In this scenario, r, the input packet rate, is approximately 500,000 packets per second. The model predicts enough capacity to run 5 modules. For comparison, note that we measured the maximum loss-free transmission rate for 1310 cycles on a 1 Ghz Pentium to be 500,004 packets per second; 1310 cycles comfortably exceeds the total processing budget needed by the 4 applications in this study (841 cycles with safety checks, and 731 cycles without any safety checks). Alternatively, with 20 active modules loaded, and an average packet size of 1K bytes (full-size ethernet data packets, with an ack every 2 packets), the system can support a traffic rate over 1 Gbps.

The demonstrated processing budget may appear somewhat constrained, assuming that users may require a much richer set of monitoring applications to be executed on the system. However, in evaluating the above processing budget, three important facts need to be considered. First, faster processors than the 1 GHz Pentium used for our measurements already exist, and processors are likely to continue improving in speed. Second, a flexible system like FLAME may not be required to cover *all* monitoring needs: one can assume that some portion of applications will be satisfied by static hardware implementation in routers, with an open architecture supporting only those functions that are not covered by the static design. Third, the figures given above represent the rate and workload at which no packets are lost. As the number of active applications increases, it will be worthwhile to allow the system to degrade gracefully. The cost of graceful degradation is an increase in the constant per-module overhead due to the added complexity of the scheduler — thus packet loss will occur under slightly lighter load than in the current configuration, but an overloaded system will shed load gracefully.

Based on our results, we can assert that FLAME is able to support a reasonable application workload on fully loaded Gbit/s links. Using FLAME on higher network speeds (*e.g.* 10 Gbit/s and more) does not currently seem practical and is outside the scope of our work.

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, after 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 and H. Bos. Towards flexible real-time network monitoring using a network processor. In *Proceedings of the 3rd USENIX/NLUUG SANE Conference (short paper)*, May 2002.

[3] 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 the 8th IFIP/IEEE Network Operations and Management Symposium (NOMS)*, pages 423–436, April 2002.

[4] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, and J. M. Smith. Practical network applications on a lightweight active management environment. In *Proceedings of the 3rd Int'l Working Conference on Active Networks (IWAN)*, pages 101–115, October 2001.

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

[6] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *Proceedings of IEEE OPENARCH 2002*, June 2002.

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

[8] 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.

[9] N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions on Networking*, 9(3):280–292, June 2001.

[10] M. Hicks, J. T. Moore, and S. Nettles. Compiling PLAN to SNAP. In *Proceedings of the 3rd Int'l Working Conference on Active Networks (IWAN)*, pages 134–151, October 2001.

[11] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis. xPF: packet filtering for low-cost network monitoring. In *Proceedings of the IEEE Workshop on High-Performance Switching and Routing (HPSR)*, pages 121–126, May 2002.

[12] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of USENIX 2002 Annual Technical Conference*, June 2002.

[13] 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):336 – 350, June 1997.

[14] G. R. Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proceedings of ACM SIGCOMM*, pages 215–227, August 1998.

[15] 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.

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

[17] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP)*, pages 217–231, December 1999.

[18] C. Partridge, A. Snoeren, T. Strayer, B. Schwartz, M. Condell, and I. Castineyra. FIRE: Flexible intra-AS routing environment. In *Proceedings of ACM SIGCOMM*, pages 191–203. August 2000.

[19] M. Roughan, 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.

[20] 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.

[21] 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.

[22] J. M. Smith and C. B. S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[23] 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.

[24] D. Wetherall. Active network vision and reality: Lessons from a capsule-based system. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP)*, pages 64 – 79, December 1999.

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