October 2006

Flexible Network Monitoring with FLAME

Kostas G. Anagnostakis
Institute for Infocomm Research (I2R)

Michael B. Greenwald
Bell Laboratories

Sotiris Ioannidis
Stevens Institute of Technology

Dekai Li
Thomson

Jonathan M. Smith
University of Pennsylvania, jms@cis.upenn.edu

Postprint version. Published in *Computer Networks*, Volume 50, Issue 14, 2006, pages 2548-2563.
Publisher URL: http://dx.doi.org/10.1016/j.comnet.2006.04.018

Flexible Network Monitoring with FLAME

Abstract

Increases in scale, complexity, dependency and security for networks have motivated increased automation of activities such as network monitoring. We have employed technology derived from active networking research to develop a series of network monitoring systems, but unlike most previous work, made application needs the priority over infrastructure properties.

This choice has produced the following results: (1) the techniques for general infrastructure are both applicable and portable to specific applications such as network monitoring; (2) tradeoffs can benefit our applications while preserving considerable flexibility; and (3) careful engineering allows applications with open architectures to perform competitively with custom-built static implementations.

These results are demonstrated via measurements of the lightweight active measurement environment (LAME), its successor, flexible LAME (FLAME), and their application to monitoring for performance and security.

Keywords

active networks, network monitoring, network measurement, worm detection

Comments

Postprint version. Published in *Computer Networks*, Volume 50, Issue 14, 2006, pages 2548-2563.
Publisher URL: http://dx.doi.org/10.1016/j.comnet.2006.04.018

Flexible Network Monitoring with FLAME

K. G. Anagnostakis [a,*], M. B. Greenwald [c], S. Ioannidis [d], D. Li [b], J. M. Smith [b]

[a] *Institute for Infocomm Research, 21 Heng Mui Keng Terrace, Singapore 119613*

[b] *Distributed Systems Laboratory, CIS Department, University of Pennsylvania 200 S. 33rd Street, Philadelphia, PA 19104, USA*

[c] *Bell Labs, 600 Mountain Ave, Murray Hill, NJ 07974, USA*

[d] *CS Department, Stevens Institute of Technology, Hoboken, NJ 07030, USA*

Abstract

Increases in scale, complexity, dependency and security for networks have motivated increased automation of activities such as network monitoring. We have employed technology derived from active networking research to develop a series of network monitoring systems, but unlike most previous work, made application needs the priority over infrastructure properties.

This choice has produced the following results: (1) the techniques for general infrastructure are both applicable and portable to specific applications such as network monitoring; (2) tradeoffs can benefit our applications while preserving considerable flexibility; and (3) careful engineering allows applications with open architectures to perform competitively with custom-built static implementations.

These results are demonstrated via measurements of the Lightweight Active Measurement Environment (LAME), its successor, Flexible LAME (FLAME), and their application to monitoring for performance and security.

1 Introduction

The bulk of research on *Active Networks* [1,2] has been directed towards building general infrastructure [3,4], with relatively little research driven by partic-

* Corresponding author.

Email addresses: kostas@i2r.a-star.edu.sg (K. G. Anagnostakis), greenwald@research.bell-labs.com (M. B. Greenwald), si@cs.stevens.edu (S. Ioannidis), dekai@dsl.cis.upenn.edu (D. Li), jms@dsl.cis.upenn.edu (J. M. Smith).

ular network functions. Recently, the focus has shifted slightly as researchers seek network functions with a *clear need* for some form of extensibility, and design appropriate *domain-specific* active-network substrates. The main experimental question to be answered in such cases is whether the desired level of flexibility can be achieved while meeting the safety and performance requirements of the particular function in question. Several experiments have been reported in this direction, including the use of smart packets for network management [5] and the design of a system for flexible intra-domain routing[6].

A third such function, that we discuss in this paper, is *network monitoring*, which is becoming increasingly critical for supporting the reactive mechanisms needed to make the Internet more efficient, robust and secure. Network providers need to analyze properties of network traffic in order to adequately provision and fine-tune the network infrastructure. Furthermore, the network occasionally finds itself in abnormal situations caused by distributed denial of service attacks, network routing outages, *etc.*. Real-time monitoring can potentially detect such conditions and react promptly. Finally, analysis of traffic is needed for network management, accounting and the verification of compliance with diverse policies.

When examining current network monitoring architectures, the weaknesses of the basic abstractions used and the need for a more flexible interface becomes evident. Static implementations of monitoring systems are unable to keep up with evolving demands. The main issue is that routers offer a set of *built-in* monitoring functions but router vendors only implement functions that are cost-effective: those that are interesting to the vast majority of possible customers. If one needs functionality that is not implemented on the router, then it becomes difficult - if not impossible - to extract the needed data from the routers. Furthermore, as customer interests evolve, router vendors can only add 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. Another critical issue is that 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.

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 this paper, we report on the design and implementation of FLAME, a system for flexible network monitoring. FLAME decouples packet monitoring from packet forwarding, offering a flexible monitoring substrate operating in an environment of common IP forwarders. Our design has three main features. First, it allows efficient implementation of functions that cannot be easily integrated in current router designs. Second, it offers robustness through the use of lightweight protection

mechanisms to prevent crashes or information leaks due to malicious or misbehaving functions. Third, it provides a flexible policy model allowing users with different degrees of trust to use the system.

Aspects of our work that were novel at the time have now been validated by their acceptance in the larger community. In network measurement, the advantages of directly using scripts or code operating on packets rather than publishing sanitized traces for offline processing are now widely recognized[7–9]. Additionally, the Cyclone-based safety model first used in FLAME has been adopted in other systems for the purpose of allowing safe general-purpose kernel extensions[10] and for upgrading TCP algorithms on end-hosts through safe mobile code[11].

Our OpenBSD-based prototype, tested on a 1 GHz Pentium PC, provides approximately 1,300 processing cycles per packet for monitoring modules on a fully-loaded 1 Gbit/s link. A typical workload on our system was measured to consume approximately 800 cycles per packet. The safety overhead for the same workload was measured to be approximately 25%.

2 Architecture

There are three main design goals in FLAME:

Flexibility. For a monitoring system to offer the needed flexibility it is important to provide a programming abstraction at the lowest possible level. Our system structure needs to allow users to inject their own code for processing packets and extracting the needed information.

Performance. Performing per-packet computations in real time (or near-real time) requires adequate processing capabilities as well as efficient handling of communication, computation and memory resources. Furthermore, executing multiple applications on the same infrastructure introduces complications. It increases the system complexity and the overall execution overhead. For the architecture to be scalable such overheads must be avoided.

Security. A monitoring system that allows several users to inject code in the data-path needs to be robust against malicious or faulty modules. Fine-grained security models are needed to increase usability because more precise policies can be specified and enforced. For example, for some users the system should allow access to the payload (under specific conditions) and for others, the IP addresses of the packet must be anonymized. The challenges in this dimension lie both in the specification, as well as the enforcement of policy for the loaded applications. This is a critical issue as security must be carefully

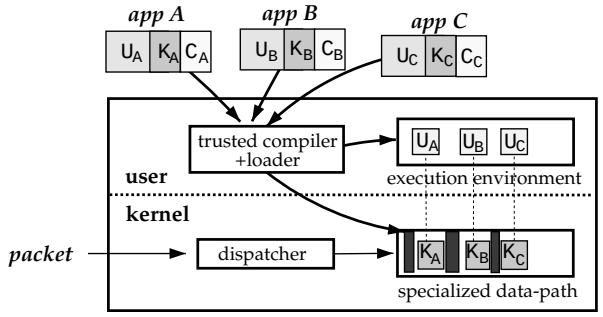


Fig. 1. **FLAME Architecture**

```
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"
-----
KeyNote-Version: 2
Authorizer: NET_MANAGER
Licensees: TrafficAnalysis
Conditions:
 (app_domain == "flame" && module == "capture" &&
 @num_cycles == 40000 && @stackspace_limit == 4096 &&
 @memory_limit == 128000)  -> "ALLOW";
Signature: "rsa-md5-hex:9a203ee8"
```

Fig. 2. **Example module credentials.**

balanced with performance and flexibility.

2.1 Architecture overview

The high-level view of the FLAME architecture is shown in Figure 1. Users can submit code to the monitoring station using a simple Web-based interface. The code submitted consists of a kernel-level module K_x, a user-level module U_x, and a set of credentials C_x. The submitted code is then processed by a trusted compiler and loaded on the monitoring station, in order to perform its intended functions. We will discuss these components in the following paragraphs.

Kernel Modules When the user submits a module to FLAME, a segment of the module is intended to run in the kernel. This piece of code, K_x, is placed on a specialized data-path, along with other such kernel modules, in order to operate on every packet received on the link. Every kernel module is allowed to execute up to a predetermined amount of time, before it is preempted by the next module in the data-path. The maximum execution time, as well as limits on memory consumption, are determined by the credentials, C_x, that the user provided to the system.

User Modules While the kernel modules, K_x, perform the bulk of the processing for the user, the user-level module U_x is responsible for higher level functions such as collecting the statistics from the kernel, and possibly passing them back to the user. Communication between the user- and the kernel-level modules is possible by using an `ioctl(2)` [1] call provided by the FLAME system. User-level modules are executed as standard operating system processes.

[1] `ioctl` is a standard mechanism in Unix operating systems for passing information between user and kernel space, particularly for setting or querying device drivers.

Credentials The final piece of data that users provide to the monitoring station is a set of credentials C_x. These credentials contain information about the resource constraints of the user code. Specifically they tell the system how many cycles the kernel module is allowed to consume, and how much memory it is permitted to allocate. The credentials are checked once at load time, and therefore don't burden the performance of the system.

We have chosen a Trust Management [12] approach to mobile code security. Entities in a trust-management system (called "principals") are identified by public keys, and may generate signed policy statements (which are similar in form to public-key certificates) that further delegate and refine the authorization they hold. This results in an inherently decentralized policy system: the system enforcing the policy only needs to consider the relevant policies and delegation credentials, which the user must provide. The KeyNote [13] implementation is used as our trust management system. An example KeyNote credential is shown in Figure 2. In the current FLAME design, we perform policy compliance checks while loading the incoming object code.

Although the FLAME prototype specifies policies using KeyNote, we are investigating the specification and representation of higher-level policies in a new project, the results of which are intended for use in FLAME-like systems.

Trusted compiler and loader The code for the kernel and user modules is written in Cyclone [14] and is processed by a trusted compiler upon submission. The compiler guarantees that memory references are contained, in order to avoid stray pointers and buffer overflows. After compilation the object code is handled by our loader which loads the kernel modules inside the operating system kernel and starts the user process.

Exception Handling To handle exceptions caused by the module code executing in the kernel we modified the trap handler of the operating system to catch possible exceptions originating from the loaded code. The trap handler checks to see whether the source of the exception is from within a FLAME module. For example, if a trap is caused by dereferencing a null pointer, the handler checks whether the pointer resides inside the memory allocated to a FLAME module. If the trap originated within a FLAME module, then the handler does not cause a system panic. Instead, the module is terminated and control is returned to the FLAME scheduler.

Bounding Execution Time The simplest method for bounding execution time would be to prohibit backward jumps. This has the advantage of providing an upper bound on execution time that is linear to the length of the

program. The drawback of this method is that it makes programming cumbersome. One alternative is to execute each installed module as a kernel thread and context switch between threads when they exceed their allocated time slot. However, preemptive scheduling and context switching are costly when we only need sequential execution of monitoring functions on incoming packets. We take a different approach similar to [15]: we augment the backward jumps with checks of a cycle counter; if the module exceeds its allocated execution time we jump to the next module. This adds an overhead of 5 assembly instructions for the check, and another 6, if the check succeeds, to initiate the jump to the next module.

Bounding Memory Use The scheduler specifies a limit on stack space allowed for each module. The function-calling checkpoint function (added by the cyclone compiler) checks the current stack pointer against the limit and jumps back to the scheduler if the limit is exceeded. This allows stack limits to be separated from time limits.

Every module also has its own separate memory allocator that only manages a small and reserved chunk of memory. The allocation algorithm is a simplified version of a generic purpose memory allocator used in C and C++ programming. The scheduler allocates the reserved space before instantiating the module and no dynamic memory allocation is needed afterwards.

Finally, to ensure proper packet buffer access, we re-encapsulate the packet in a Cyclone-compatible structure when the kernel passes the packet to a Cyclone module. This structure records the length of the packet so modules cannot access beyond the buffer.

SMP support We dispatch each packet, on arrival, to a single processor and execute all modules there. Section 3.3 discusses the trade-offs between either pinning modules or pinning packets to processors. If packets are processed on multiple processors, a single module may run concurrently on two or more processors and may experience race conditions that could corrupt the module's persistent memory region. FLAME does not export an IPC or synchronization interface, so FLAME itself must enforce mutual exclusion between multiple instances of the same module. The scheduler locks each module upon entry, and equal priority modules are selected for execution in random order. If the scheduler finds that a lock is held, the scheduler simply chooses another module.

3 Evolution

In this section we briefly present the historical progression of our system and the issues raised while designing and implementing the architecture.

3.1 The LAME system

LAME [16] attempted to provide the flexibility and safety properties needed for programmable network monitoring (traffic access policies, crash and corruption safety and support for multiple concurrent applications) using only off-the-shelf components. The meta-goal of this work was to show that experimental active networking systems can be easily prototyped on top of existing operating systems and that the focus of researchers' efforts should be on applications rather than infrastructure.

LAME was implemented as a daemon that manages the lifecycle of monitoring modules on the system. Modules can be written in any language (including C) and are presented to the LAME daemon as shared libraries with known entry points for initialization and per-packet processing. The modules are setup as separate UNIX processes to provide protection against crashes and isolation between different modules. Modules use the familiar `libpcap(8)` API for reading packets from the network [17,18], and the LAME daemon intercepts these references when loading the module and invokes custom wrapper functions to perform all the necessary policy-related functions.

3.2 The FLAME system: rethinking LAME

The limitations of LAME were mostly performance-related: the high cost of context switching grows as the number of modules increases (given that each module has to execute as a separate process for reasons of protection), and the networking subsystem in the commodity OS introduces further overheads that limit the performance of the system.

The FLAME system [19] addresses the performance limitations of LAME in two ways. First, we avoid the high price of safety of the process-based isolation model by executing all modules in the same address space. This is achieved using the language-based safety mechanisms of Cyclone along with a minimal set of modifications for limiting the processing time of each application. Although this requires the use of a new language for writing applications, in most cases porting C applications to Cyclone has been fairly straightforward. Second, we inject the compiled Cyclone modules in the kernel of the operating

system to avoid the cost of copying packets and switching context from kernel to user-space. Because the kernel code is limited to fast-path packet processing only, we also allow users to supply additional user-level functions to enable post-processing of results in a similar fashion to LAME.

3.3 SHAME: fine-tuning a second generation FLAME

SHAME [20], the latest version of FLAME, provides three engineering enhancements to the original prototype: support for polling-driven packet processing, kernel-independent packaging and support for multi-processors.

Polling. Interrupt-driven I/O in commodity operating systems introduces a performance bottleneck as the load (and therefore, the frequency of interrupts) increases[21,22]. To avoid this problem SHAME uses polling as implemented in some of the more advanced NICs. This has resulted in a significant performance improvement compared to the original FLAME prototype. (See measurements in Section 5.1).

Kernel-independent packaging. FLAME was originally packaged as a modified OpenBSD kernel that included the kernel-level core of FLAME and the necessary hooks in the network stack and device drivers (*i.e.,* for polling). We encountered three practical problems with this approach. First, as the underlying OS evolves this requires porting the FLAME package for each new release of the OS. Second, it requires additional effort for maintaining backward compatibility as the FLAME system itself evolves. Third, it discourages the use of the system given the need to rebuild the kernel, especially when the kernel modifications conflict with FLAME. We therefore re-engineered FLAME as an external, loadable kernel module that requires no kernel patching at all. One problem we faced is that there was no standard device-independent API for polling devices in the original OpenBSD prototype. However, more recent releases of the Linux OS provide such an API (called NAPI) and similar support is being developed for FreeBSD. The current generation of SHAME on Linux uses NAPI. Additionally FLAME requires the use of a custom trap handler. For this, FLAME modifies the entry point to the interrupt handler for page faults, using assembly language. (This small piece must be reimplemented for each architecture.)

SMP support. FLAME modules are expected to run in isolation, thus the main challenge we faced when FLAME began to use multiple processors was in the design of the scheduling algorithm. Our main concern is to maximize

throughput. FLAME has multiple network monitoring modules processing multiple incoming packets. Two strategies of allocating tasks to processors are natural: either bind modules to processors and have each packet flow through each processor in sequence, or dispatch each incoming packet to a processor and execute all applicable modules on that processor.

Context switching of FLAME modules is very lightweight; all modules run in the kernel address space and only require some preliminary memory allocation and setup of Cyclone structures. Switching to a FLAME module is more similar to a function call (with static, private data) than a context switch. Modules are only preempted because of violating a resource limit and therefore are never resumed. They can be cleaned up immediately and no intermediate state need be stored.

The primary method of maximizing throughput is to keep processor utilization high by avoiding stalls, such as cache misses and keeping the load on all CPUs balanced, so that no processor needs to wait.

Binding modules to processors has the obvious advantage of minimizing the instruction cache footprint and reducing the likelihood of cache misses. If each processor has a smaller number of modules, it is more likely that the code for each module will be resident in the I-cache the next time the module is executed. It also eliminates any need for synchronization (FLAME isolates all modules from each other, so two distinct modules will not share a common data structure). The only sharing would be between two instances of the same module; but if we bind modules to processors, then only one instance of a module ever executes at a single time.

However, there are two problems with this approach. First, reducing I-cache misses does nothing to reduce misses on the packet in the data cache. Further, the same packet will take misses in *each* processor it migrates to. Second, it is impossible to choose an *a priori* partition of the modules such that the combined execution time of the set on each processor is equal for every packet. Consequently, we either have idle processors or extra bookkeeping and buffering as the packets progress through each processor at different rates.

Consequently, we chose to dispatch each incoming packet to a processor and execute all applicable modules on that processor. This reduces the number of cache misses on the packet and also reduces idle processor time. When a CPU finishes processing a packet, it acquires a new packet from the input queue. The only time a CPU is idle is when the packet queue is empty.

4 Applications

We discuss applications, some of which have been implemented on FLAME to demonstrate the flexibility and investigate the performance of our system. We chose applications that are not currently supported by routers, which makes them likely candidates for deployment using a flexible system like FLAME. We also emphasize applications beyond basic network measurement research: functions that are of practical interest to network operators for performance and security.

4.1 Performance monitoring

4.1.1 Trajectory sampling

Trajectory sampling [23], 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 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.

4.1.2 TCP RTT measurements

We implemented a simple method for obtaining approximate round-trip time measurements experienced by TCP flows on a network link. Round-trip delays provide a reasonably good estimate of end-to-end performance, mostly because of their role in TCP congestion control [24]. Additionally, measuring the round-trip times observed over a specific ISP provides a reasonable qualitative measure of the ISP's infrastructure and the connectivity to the rest of the Internet. Finally, observing the evolution of round-trip delays 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 method to detect service degradation or routing failures in an upstream provider and take appropriate measures, such as re-routing traffic through another peer ISP.

The implementation on FLAME is both simple and efficient. We watch for TCP SYN packets indicating a new TCP flow, and then look 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.[2] For every SYN packet received, we store a timestamp into a hash-table. The first ACK after a SYN usually has a sequence number which is the SYN packet's sequence number plus one. Thus, we can use this number as the key for hashing. In addition to watching for SYN packets, the application needs to perform lookups for every ACK received. The hash-table 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 therefore not appropriate for real-time measurement, as well as an alternative, wavelet-based method are described in [25]. Note that none of these algorithms (including ours) work for parallel paths where SYN and ACK packets are forwarded on different links. Retransmission of the SYN packet does not affect measurement, as the timestamp in the hash-table will be updated. Retransmission of an ACK packet introduces error when the first ACK is not recorded. However, if this happens rarely, then the error is not significant compared to the overall RTT statistics. If this happens frequently, due to a highly congested link, then a cluster of samples around typical TCP timeout values will appear in the overall statistics, and can therefore be detected and interpreted.

[2] 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 may be used to indicate server overload or other phenomena.

4.1.3 Customized flow measurements

The flow abstraction[26] is widely used in the Internet for measuring network usage, through standard router mechanisms such as Cisco's NetFlow[27]. A flow is a collection of packets observed on a network link that has a set of common characteristics.

In most current implementations the classification of packets into flows is based on source and destination address (or prefix), protocol version and application port numbers. The problem with this approach is that it gives very little flexibility to ISPs for adapting flow measurements to their needs. For instance, to support ECN [28] flow measurements, one needs to classify traffic based on network prefixes but also separate marked from unmarked packets. This is not supported by any of the existing router accounting mechanisms: NetFlow allows *either* per-ToS (which is used for ECN marking) *or* per-AS *or* per-network accounting tables.

Customized flow measurement modules are easy to implement in a flexible system like FLAME. To demonstrate the flexibility of FLAME we have implemented a proof-of-concept module for measuring ECN flows, with the low-level kernel module handling packet classification and the user-level code flushing the tables to the application, based on time and memory thresholds.

Other examples of possible customizations to flow measurements include content-based classification for measuring applications that do not communicate over well-known ports (such as FTP and a growing number of peer-to-peer systems) and extracting finer-grained or application-level statistics (such as Web browser usage, object sizes in persistent HTTP/1.1 connections, *etc.*)

4.2 Security applications

The main benefit of implementing internal-network security mechanisms using FLAME is the ability to adapt rapidly and deploy new mechanisms as new threats become known. We discuss some security functions below.

4.2.1 IP Traceback

The current Internet architecture offers little protection against ill-behaved traffic. In recent years Distributed Denial of Service (DDoS) attacks have increased, which has led to the study of appropriate "traceback" mechanisms. A traceback mechanism detects the attack source(s), despite the fact that IP source addresses may be spoofed by the attacker, and responds by confining or blocking traffic from the attacking sites. FLAME allows an implementation of

IP Traceback [29–31] without requiring modification of router functionality to support confinement or traffic blocking. This observation also agrees with the general FLAME philosophy of flexible monitoring on simple IP forwarders. Although these mechanisms have been developed for quite a while, the lack of flexibility in existing routers has so far resulted in limited deployment and thus less effective DDoS protection.

We implement Traceback on FLAME by simply monitoring traffic and recording the upstream router for each packet, with the option of using sampling to reduce the cost of monitoring and the quantity of data collected by each FLAME system. We use out-of-band communication to recursively trigger traceback on neighboring FLAME systems to find the source of the attacks. Once the attackers have been traced, router control commands are initiated from the monitor to an appropriate management interface to block or otherwise confine the attacking traffic. In our experiments we used PC-based routers with the ALTQ [32] QoS API for blocking the attacking sites; in the general case we believe existing router mechanisms are sufficient for reacting to attacks, *once the attack has been identified*. In this work we do not address the question of *how* the attack is identified. We assume that external mechanisms (such as server-, or link-overload detectors) will trigger the traceback mechanism on FLAME-enabled network elements.

4.2.2 Distributed Worm detection

The concept of "worms" and techniques to implement them have existed since the early descriptions in [33,34]. 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 [35]. 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?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN .....
```

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 worm.

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 worm 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 in time.

On the other hand, we know of a mechanism that is able to deliver worm 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.

We have reported some preliminary results in this direction in [36]. We have designed COVERAGE, a *cooperative* immunization system that helps defend against computer worms. The nodes in our system cooperate and inform each other of ongoing attacks and the actions necessary to defend. We use a simple model that uses random polling to evaluate the trustworthiness of worm activity reports from remote nodes, as well as the amount of resources the system should allocate for responding to the various active worms.

The design of FLAME is ideal for implementing this kind of functionality. 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.

5 Performance evaluation

We present experiments evaluating the performance of FLAME, the cost of applications, safety overheads and sustainable workloads.

The test network 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 [37] code under Linux 2.2.14 on the sending host, and the SHAME version of the system with polling support under Linux 2.4.20. All hosts use the Intel PRO/1000SC

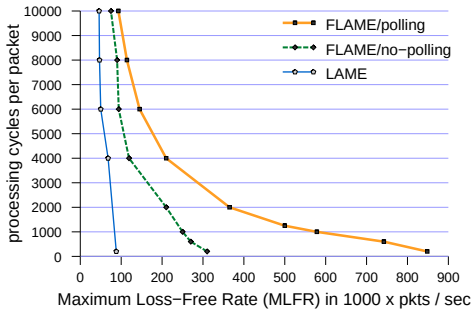


Fig. 3. **Available processing cycles per packet as a function of maximum loss-free traffic rate**

Module	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%

Fig. 4. **Module processing costs (in cycles) and protection overheads.**

Gigabit NIC.

5.1 System performance

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 of 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 3 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.

5.2 Workload analysis and safety overheads

We instrumented the application modules using the Pentium performance counters to obtain an accurate indication of the processing cost for each application. We read the value of the Pentium cycle counter before and after the execution of application code for each packet. Due to 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 specific properties of the traffic, 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.

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). The difference between C and pure Cyclone indicates the incremental cost of basic pointer-safety needed for safe monitoring modules, whereas the difference between pure Cyclone and protected Cyclone reflects the cost of the additional safeguards such as controlling backward jumps. We measure the median execution time of each module, averaged over 113 runs. The results from this experiment are summarized in Table 4. 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, 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.

5.3 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 5.1, and the costs of our experimental applications and the measured safety overheads from Section 5.2.

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 [3], we get that the number of available cycles for processing is $3 \times 10^9 r^{-1.1216}$. The rate of 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 Gbit/s.

The demonstrated processing budget may appear somewhat constrained, assuming that users may require a much richer set of 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 are already common, and processors are

[3] 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.

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.

A straightforward approach is to cycle through the applications in priority order, and monitor the average number of cycles between packets. A threshold slightly below the mean cycle count can be used as an execution limit to abort low priority modules if the system falls behind. The packet buffer should provide adequate cushioning to ensure that the highest priority modules never miss any packets. In this manner, adding non-essential applications (*e.g.,* for research purposes) will not hurt critical functionality (*e.g.,* billing, or security), as increasing traffic rates saturate the system, and may thus be safely admitted for executing on the system. Although our current implementation does not provide this feature, it appears reasonably easy to implement.

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.

5.4 Performance of FLAME-based distributed worm defense

As a final case-study on the ability to do real work under such tight processing constraints, we evaluate the behavior of a FLAME-based cooperative worm defense, COVERAGE [4] , as the per-module budget on the FLAME nodes decreases.

In COVERAGE, nodes throughout the Internet scan locally for potential worm attacks, and periodically probe randomly selected remote sites to exchange information. COVERAGE modules in FLAME are limited to scanning for, or filtering out, only a small number of potential worms per packet. *Scanning* a packet for a worm simply asks whether the worm is present, in an attempt to determine whether this pattern actually represents a dangerous worm. *Filtering* for a given worm occurs if COVERAGE has decided this

[4] The algorithms used in COVERAGE are described more fully in [36].

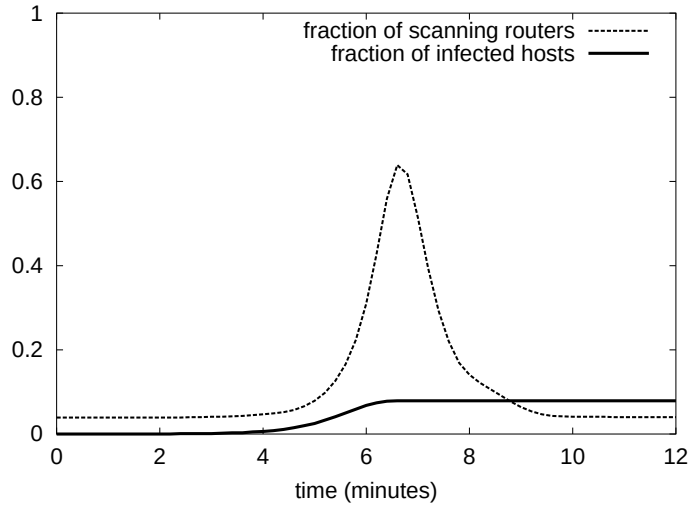


Fig. 5. **Fractions of infected hosts and scanning edge-routers over time.**

worm *is* attacking, and COVERAGE is attempting to destroy all instances of infected messages. The periodic polling, done in a user space process, determines the relative priority of all known worms (as well as potentially acquiring new modules to look for other worms). COVERAGE ranks the worms by *virulence*, an estimate of how long, given their recent rate of growth, it would take them to cover the entire network [5]. The FLAME per-module processing budget limits the number of worms that COVERAGE can scan for.

We model the FLAME budget by setting a threshold value of virulence. COVERAGE will not have enough cycles to care about worms less virulent than `threshold`. Thus, if we say that `threshold` = 5, then COVERAGE will not have sufficient cycles to scan for a particular worm until they are within 5 measurement intervals of covering the entire network. If the measurement is 1 minute, then COVERAGE does not *begin* to react until 5 minutes before a worm is poised to conquer the entire network. If the threshold is 20, then FLAME has enough capacity to allow COVERAGE to consider many more worms, and to catch them at a correspondingly earlier period in their growth.

We evaluate the effectiveness of COVERAGE when under heavy attack by worms, and consider the effect of limited FLAME processing cycles by varying `threshold`.

[5] The ranking actually incorporates some randomness to ensure that COVERAGE has a chance to get some information about low virulence worms in order to determine whether they have exceeded threshold.

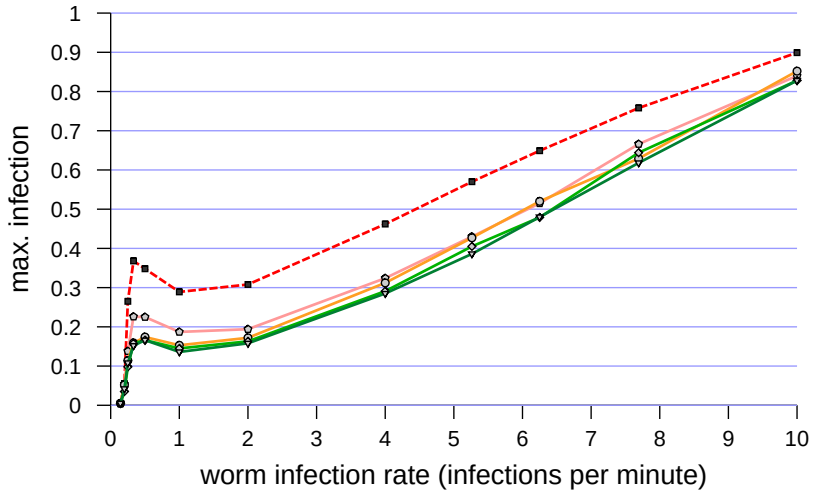


Fig. 6. **Maximum fraction of infected hosts at the high-water-mark of the attack, as a function of worm infection rate.**

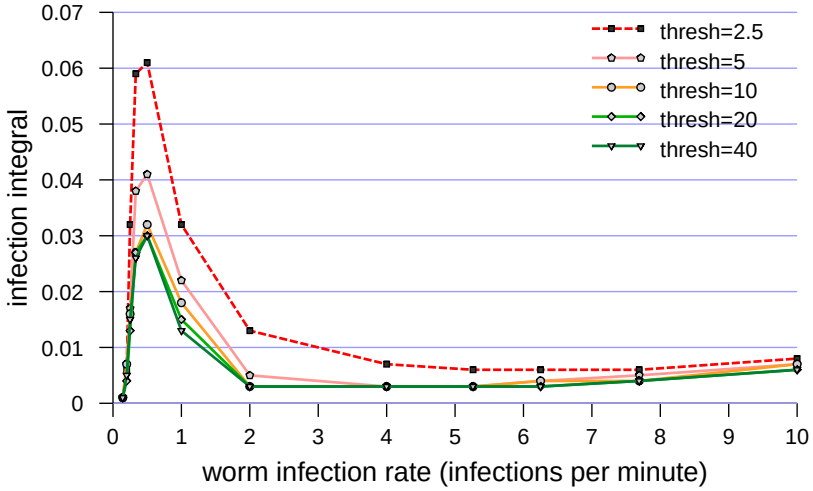


Fig. 7. **Integral of number of infected hosts over entire measurement interval, divided by duration of interval as a function of infection rate of attacking worm.**

5.4.1 Results

We limit our simulation to a simple, relatively small network of 100,000 edge-routers, each connected to 8 hosts. The network contains 2,000 domains consisting of 50 edge-routers each. We set the local-domain polling interval of COVERAGE agents to 1.8 seconds , the maximum and minimum remote polling intervals to 6 seconds and 1.8 seconds, respectively.

Our analysis uses four metrics. First, we estimate the total effect of the attack by integrating the number of infected nodes over time. Second, we compute the maximum number of infected nodes at the height of the attack; this is another indicator of success and a measure of the penalty imposed by inadequate scanning cycles allowed by FLAME. Third, we consider the number

of FLAME-based edge-routers actively scanning/filtering this worm. This is a measure of the computational overhead of the response mechanism. Our fourth metric, the total number of messages sent, measures the communication cost between the (non-FLAME) COVERAGE agents.

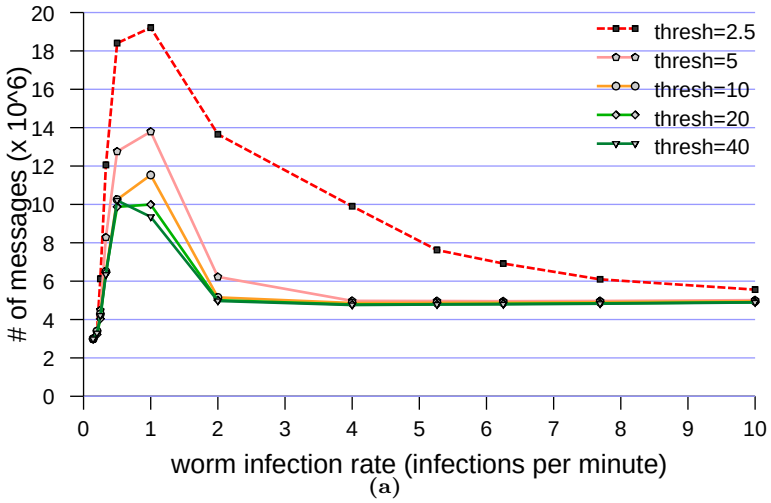
Figure 5 displays a single example run of the COVERAGE algorithm against a worm. One can see the initial stage of the infection and the response of the algorithm: the worm manages to infect roughly 10% of the hosts; cooperation between COVERAGE agents results in a rapid activation of filtering on roughly 60% of the network effectively eliminating the worm. Soon after stopping the attack, most COVERAGE agents deactivate scanning/filtering of this worm (when the worm is quiescent, the background scanning rate has about 4% of the edge-routers permanently scanning for the worm).

The progress of infections of differing virulence is shown in Figures 6 and 7. Figure 6 shows that both very fast and very slow worms are able to infect a large fraction of hosts before COVERAGE halted their growth. Fast worms achieve this by brute force; slow worms by stealth. Fewer available FLAME cycles (lower values of threshold) exacerbate the situation. Figure 7 shows that the net effect of the fast worms is actually less significant than the slow worms; although fast worms infect a larger fraction of nodes, they are also detected much earlier, and the worm is contained far more rapidly than the slow worm.

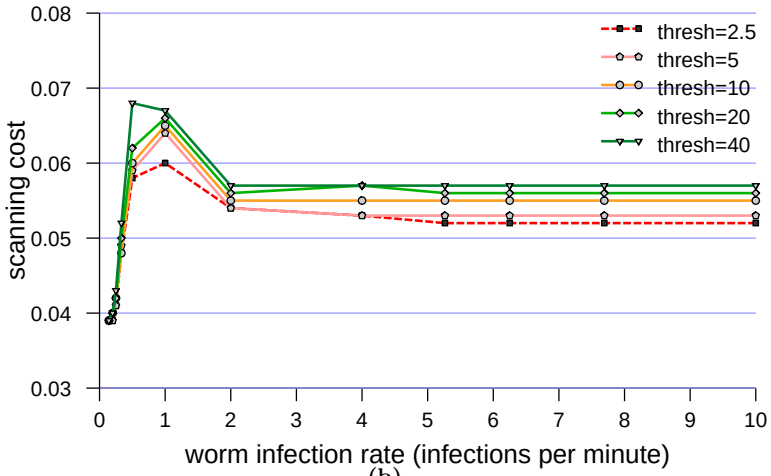
(In both graphs, data points at the far left represent such slow worms that they were still growing at the end of the simulation, and COVERAGE had not yet identified them as malevolent or above threshold.)

Figure 8(a) shows the communication costs of the non-FLAME COVERAGE agents. In general, slower worms impose a larger communication burden because COVERAGE modules need to spend more time and messages convincing each other that this worm is above threshold. We see that when the COVERAGE modules running as part of FLAME (the packet scanners) have fewer available computation cycles, this imposes a larger burden on the non-FLAME agents (the user-space modules that communicate with each other and prioritize the tasks of the FLAME modules).

Figure 8(b) shows the average fraction of nodes scanning for a worm as a function of the virulence of the worm. We see that, in general, slower worms cause a larger fraction of nodes to be actively scanning. This is because a node scans from the time it believes a worm to be dangerous until the danger has abated. For slow worms, the scanning duration for the early detectors is significantly longer than the late detectors; and all continue to run until the worm is contained. For fast worms, there is only a short interval between the time that the first node detects it, and everyone is convinced that the worm



(a)



(b)

Fig. 8. **Total communication cost and normalized integral of scanning activity. Both are plotted vs. infection rate of attacking worm.**

is dangerous, and, finally, the worm is contained. However, regardless of the speed of the worm, there is a minimum scanning duration, and hence scanning cost decreases to a limit as worm speed increases. In the plots of both communication and scanning costs we see that the extremely slow growth worms at the left of the figures were not simulated long enough for the detectors to identify them as a threat.

In summary, we see that COVERAGE degrades gracefully, even when FLAME makes only a small number of computational cycles available to the COVERAGE modules. However, there does appear to be a cliff between a threshold of 2.5 and 5, under which the communication burden climbs rapidly and the effectiveness of the anti-worm measures degrades noticeably. Fortunately, for this application (assuming that the cost of identifying particular worms remains at roughly the current levels), the computing budget offered by FLAME under a reasonable number of modules seems comfortably above the point of collapse.

6 Related work

The main thrust of our research has been to investigate the three-way tradeoff between flexibility, performance, and safety and security in network monitoring applications. Encouraged by the progress in the Active Networking, we designed several extremely flexible network monitoring system, culminating in SHAME.

In other active networking research, Smart Packets [5] and ABLE [38] provide, as does SHAME, programmable platforms for network management applications. SHAME, Smart Packets, and ABLE, all adopt approaches that are similar in principle to Management-by-Delegation (MbD) models [39]. The key difference lies in the level of abstraction. ABLE allows applications to poll SNMP interfaces from inside the network, while Smart Packets operate on a management interface at the language level. In both cases the management information and control settings are pre-defined subsets of the managed element state. In contrast, as we outlined earlier, our work was motivated by the need for more flexibility than is provided by pre-defined subsets, such as exported SNMP interfaces. Therefore we argue for arbitrary packet-level traffic measurement approach instead of relying on the already abstracted SNMP-based metrics. In particular, FLAME, and its successor SHAME, dynamically install arbitrary modules in the monitoring system, close to the information source.

Another active-networking effort related to our work is the SENCOMM project at BBN[40]. The SENCOMM architecture focuses on the use of smart packets for polling monitoring information on active network elements, and also provides support for dynamically loading monitoring modules, written in Java, on the active network elements.

Outside the Active Networking community, numerous techniques have been developed for flexible network monitoring. The first generation of a tools such as `tcpdump` were based on the Berkeley Packet Filter [41] (BPF). The Packet Filter provides operating system functionality for user-level traffic monitoring. Users define filters in a "filter language" and pass them to the system, to execute on a "filter machine" inside the kernel. The filters specify which packets the user is interested in. These packets are then passed to the user-level application for processing. Although BPF provides a programming facility, the filter language has no access to a store. This lack of persistent memory limited the language to stateless filters. As such, we are not aware of any early filter programs that did anything other than simple demultiplexing. Providing a richer programming environment for network monitoring at the packet filter level was proposed in [18], and later in [42]. More recent descendants of BPF, such as `ipf`[43] and `pf`[44], support limited state. The state, however, is hardwired, limited to the necessary fields to support hard-coded rules, such

as NAT, and match TCP packets to ongoing connections.

The work most closely related to our is Windmill [45], an extensible network probe environment which allows loading of "experiments" on the probe, with the purpose of analyzing protocol performance. Windmill includes a facility to demultiplex a single packet stream to multiple applications. The current FLAME implementation does not provide such a facility, since most applications we considered are interested in processing the entire traffic stream. Further, Windmill was designed for a trusted environment and assumes that applications are reliable, and thus does not provide the safety mechanisms of FLAME. In contrast to FLAME, Windmill also does not efficiently scale to higher network speeds or application workloads, as it operates in user space.

OC3MON [46] is a dedicated host-based monitor for snooping on 155 Mbit/s OC3 ATM links. The host is attached to the link using an optical splitter so that the monitoring function does not interfere with regular service. The host monitors packets and records the captured headers in files for post-processing. No real-time functionality for packet processing is considered in the original design. Because OC3MON performs most of its work in an isolated post-processing phase, it finesses the issues of performance and security that FLAME must confront head on. For higher speed networks, the DAG series of monitoring-specific network cards have been developed, implementing some standard header processing functions in hardware [47].

Netramet [48] implements an SNMP-based Traffic Flow Measurement Architecture and Meter Management Information Base. Netramet inherits the limitations of SNMP and standards-based protocol designs with regard to their limited flexibility.

The NIMI project [49] provides a platform for large-scale Internet measurement using Java- and Python-based modules. NIMI only allows active measurements, while the security policy and resource control issues are addressed using standard ACL-like schemes.

7 Summary and Concluding Remarks

In this paper we have reported on our experiences with 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. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, G. Minden, A survey of active network research, IEEE Communications Magazine (1997) 80 – 86.

[2] J. M. Smith, K. Calvert, S. Murphy, H. Orman, L. Peterson, Activating networks: A progress report, IEEE Computer 32 (4).

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

[4] 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), 1999, pp. 64 – 79.

[5] B. Schwartz, A. Jackson, T. Strayer, W. Zhou, R. Rockwell, C. Partridge, Smart packets: Applying active networks to network management, ACM Transactions on Computer Systems 18 (1) (2000) 67–88.

[6] C. Partridge, A. Snoeren, T. Strayer, B. Schwartz, M. Condell, I. Castineyra, FIRE: Flexible intra-AS routing environment, in: Proceedings of ACM SIGCOMM, 2000, pp. 191–203.

[7] J. Mogul, Trace anonymization misses the point, presentation on WWW 2002 Panel on Web Measurements.
URL `http://www2002.org/presentations/mogul-n.pdf`

[8] R. Pang, V. Paxson, A high-level programming environment for packet trace anonymization and transformation, in: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), ACM Press, 2003, pp. 339–351.

[9] J. Coppens, S. V. den Berghe, H. Bos, E. Markatos, F. D. Turck, A. Oslebo, S. Ubik, SCAMPI: A scalable and programmable architecture for monitoring gigabit networks, in: Proceedings of the Workshop on End-to-End Monitoring Techniques and Services (E2EMON), in conjunction with 6th IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MMNS), 2003.

[10] H. Bos, B. Samwel, Safe kernel programming in the OKE, in: Proceedings of IEEE OPENARCH, 2002.

[11] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, T. Stack, Upgrading transport protocols using untrusted mobile code, in: Proceedings of the 19th ACM Symposium on Operating systems Principles (SOSP), ACM Press, 2003, pp. 1–14.

[12] M. Blaze, J. Feigenbaum, J. Lacy, Decentralized Trust Management, in: Proc. of the 17th Symposium on Security and Privacy, 1996, pp. 164–173.

[13] M. Blaze, J. Feigenbaum, J. Ioannidis, A. Keromytis, The KeyNote Trust-Management System Version 2, RFC 2704, http://www.rfc-editor.org/ (September 1999).

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

[15] M. Hicks, J. T. Moore, S. Nettles, Compiling PLAN to SNAP, in: Proceedings of the 3rd International Working Conference on Active Networks (IWAN), 2001, pp. 134–151.

[16] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. M. Smith, Practical network applications on a lightweight active management environment, in: Proceedings of the 3rd International Working Conference on Active Networks (IWAN), 2001, pp. 101–115.

[17] A. Begel, S. McCanne, S. L. Graham, BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture, in: SIGCOMM, 1999, pp. 123–134.

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

[19] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, M. B. Greenwald, J. M. Smith, Efficient packet monitoring for network management, in: Proceedings of the 8th IFIP/IEEE Network Operations and Management Symposium (NOMS), 2002, pp. 423–436, (an earlier extended version of this paper is available as UPenn Technical Report MS-CIS-01-28, September 2001).

[20] K. G. Anagnostakis, M. B. Greenwald, S. Ioannidis, S. Miltchev, Open Packet Monitoring on FLAME: Safety, Performance and Applications, in: Proceedings of the 4th International Working Conference on Active Networks (IWAN'02), 2002.

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

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

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

[24] T. V. Lakshman, 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) 336 – 350.

[25] P. Huang, A. Feldmann, W. Willinger, A non-intrusive, wavelet-based approach to detecting network performance problems, in: Proceedings of the ACM SIGCOMM Internet Measurement Workshop (IMW), 2001, pp. 213 – 227.

[26] K. C. Claffy, H.-W. Braun, G. C. Polyzos, A parameterizable methodology for internet traffic flow profiling, IEEE Journal of Selected Areas in Communications 13 (8) (1995) 1481–1494.

[27] V. Bollapragada, R. White, C. Murphy, Inside Cisco IOS Software Architecture, Cisco Press, 2000.

[28] S. Floyd, TCP and explicit congestion notification, ACM Computer Communication Review 24 (5) (1994) 10–23.

[29] S. Savage, D. Wetherall, A. Karlin, T. Anderson, Practical network support for IP traceback, in: Proceedings of ACM SIGCOMM'2000, 2000.

[30] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, S. Shenker, Controlling high bandwidth aggregates in the network – extended version, `http://www.aciri.org/pushback/`.

[31] V. C. Van, A defense against address spoofing using active networks, Bachelor's Thesis, MIT (1997).

[32] K. Cho, A framework for alternate queueing: Towards traffic management by PC-UNIX based routers, in: Proceedings of the USENIX 1998 Annual Technical Conference, 1998.

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

[34] J. F. Shoch, J. A. Hupp, The "worm" programs – early experiments with a distributed computation, Communications of the ACM 25 (3) (1982) 172–180.

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

[36] K. G. Anagnostakis, M. B. Greenwald, S. Ioannidis, A. D. Keromytis, D. Li, A Cooperative Immunization System for an Untrusting Internet, in: Proceedings of the 11th IEEE Internation Conference on Networking (ICON), 2003.

[37] R. Morris, E. Kohler, J. Jannotti, M. F. Kaashoek, The click modular router, in: Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP), 1999, pp. 217–231.

[38] D. Raz, Y. Shavitt, An active network approach for efficient network management, in: Proceedings of the 1st International Working Conference on Active Networks (IWAN), 1999, pp. 220 –231.

[39] G. Goldszmidt, Y. Yemini, Distributed management by delegation, in: Proc. of the 15th International Conference on Distributed Computing Systems, 1995, pp. 333–340.

[40] BBN, Advanced Networking Dept., Smart Environment for Network Control, Monitoring, and Management (SENCOMM).
URL `http://www.ir.bbn.com/projects/sencomm/sencomm-index.html`

[41] S. McCanne, V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture, in: Proceedings of the Winter 1993 USENIX Conference, 1993, pp. 259–270.

[42] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, G. Portokalidis, FFPF: Fairly Fast Packet Filters, in: Proceedings of OSDI'04, San Francisco, CA, 2004.

[43] G. v. Rooij, Real Stateful TCP Packet Filtering in IP Filter, in: Proceedings of 2nd International System Administration and Network Engineering Conference (SANE 2000), 2000, maastricht, The Netherlands.

[44] D. Hartmeier, Design and performance of the OpenBSD stateful packet filter (pf), in: Proceedings of the USENIX Annual Technical Conference, Freenix Track, 2002, pp. 171–180.

[45] G. R. Malan, F. Jahanian, An extensible probe architecture for network protocol performance measurement, in: Proceedings of ACM SIGCOMM, 1998, pp. 215–227.

[46] J. Apisdorf, k claffy, K. Thompson, R. Wilder, OC3MON: Flexible, Affordable, High Performance Statistics Collection, in: Proceedings of the 1996 LISA X Conference, 1996.

[47] The WAND Group, University of Waikato, New Zealand, DAG4 Architecture, http://dag.cs.waikato.ac.nz/dag/dag4-arch.html (2000).

[48] N. Brownlee, Traffic Flow Measurement: Experiences with NeTraMet, RFC2123, http://www.rfc-editor.org/ (March 1997).

[49] V. Paxson, J. Mahdavi, A. Adams, M. Mathis, An Architecture for Large-Scale Internet Measurement, IEEE Communications Magazine (1998) 48–54.