The Price of Safety in an Active Network

D. Scott Alexander, Paul B. Menage, Angelos D. Keromytis,
William A. Arbaugh, Kostas G Anagnostakis, and Jonathan M. Smith

Abstract: **Security is a major challenge for "Active Networking," as accessible programmability creates numerous opportunities for mischief. The point at which programmability is exposed, e.g., through the loading and execution of code in network elements, must therefore be carefully crafted to ensure security.**

The SwitchWare active networking research project has studied the architectural implications of various tradeoffs between performance and security. Namespace protection and type safety were achieved with a module loader for active networks, ALIEN, which carefully delineated boundaries for privilege and dynamic updates. ALIEN supports two extensions, the Secure Active Network Environment (SANE), and the Resource Controlled Active Network Environment (RCANE). SANE extends ALIEN's node protection model into a distributed setting, and uses a secure bootstrap to guarantee integrity of the namespace protection system. RCANE provides resource isolation between active network node users, including separate heaps and robust time-division multiplexing of the node.

The SANE and RCANE systems show that convincing active network security can be achieved. This paper contributes a measurement-based analysis of the costs of such security with an analysis of each system based on both execution traces and end-to-end behavior.

Index Terms: **Active networking, security, performance.**

I. INTRODUCTION

The design space for active networks has many dimensions, but the most important are flexibility, security, usability and performance. While flexibility and usability follow from the choice of a programming environment, the tradeoff between performance and security is the major architectural focus in active networking research. As we have previously discussed our approaches to these tradeoffs [1]–[5], the interested reader may consult those references for further background. Here, we focus on the implications of our choices. Thus, we provide a detailed exposition on the architectural decisions supporting security, and extensive and detailed measurements of performance resulting from our choices.

Broadly, security is the restriction of actions within a system to protect the operation of the system. Safety and security are closely related, and are often supported by the same mechanisms within a system. For our purposes, we distinguish safety from security using the rule of thumb that safety is to protect you from yourself, while security protects you from others.

Information security consists of getting the right information to the right location at the right time. In a given context, a security *policy* is specified reflecting details (location, information) appropriate to the context. The flexibility of an active networking infrastructure expands the possibilities for mischief. For example, "denial-of-service" attacks are possible against a multiplicity of resources such as CPU cycles, storage and output link bandwidth, which are used by loaded programs. Controlling access to resources in an active networking requires controlling the actions of loaded modules.

The remainder of the paper is as follows. Section II discusses security threats to an active network infrastructure. Section III briefly discusses the Secure Active Network Environment (SANE)[1], and its services. SANE provides basic security elements such as: a secure bootstrap; key exchange; authentication and identification of network entities; packet confidentiality and integrity; resource and access control; and name-space protection. Section IV discusses the design principles and realization of the RCANE architecture. Section V provides a detailed measurement study of the implementation, with particular attention focused on a cost breakdown for an example application, "active ping," and Section VI evaluates the effectiveness of RCANE. Section VII discusses those results and the various costs in our system. Section VIII reviews our contributions and those of related work, and Section IX concludes the paper with a discussion of the implications of these results for designers of active networks and other systems with mobile untrusted code.

II. THREATS

An active network infrastructure is very different from the current Internet. The latter was conceived as a network giving only best-effort delivery. In the presence of congestion, protocols such as TCP throttle back their output, so as not to overload network switches. Token support for QoS was provided in the *Type of Service* (ToS) field, which allowed packets to be marked according to their traffic type and precedence. Such information is typically ignored by network routers, leading to a best-effort service for all network users. Security policies are enforced at the endpoints.

The potential resource load at a node caused by the activities

Manuscript received month date,year.

D. S. Alexander is with Activium, Inc., in New York, NY, e-mail: salex@activium.com.

P. B.Menage is with Ensim Corporation, in Sunnyvale, CA, e-mail: pmenage@ensim.com.

K. G. Anagnostakis, A. D. Keromytis, and J. m. Smith are with the University of Pennsylvania, in Philadelphia, PA, e-mail: anagnost@dsl.cis.upenn.edu, angelos@dsl.cis.upenn.edu, jms@dsl.cis.upenn.edu.

W. A. Arbaugh is with the University of Maryland in College Park, MD, e-mail: waa@cs.umd.edu.

This work was supported by DARPA under Contract #N66001-96-C-82, with additional support from the Intel Corporation, the UK Engineering and Physical Sciences Research Council, and the US National Science Foundation under Grants #ANI 98-13875, ANI 99-06855, and ANI 00-82386.

1229-2370/01/$10.00 © 2001 KICS

```
let hostileForwardPacket pkt =
    while (true) do
         allocateSomeMemory ();
         sendPacketToNeighbours (pkt)
    done
```

Fig. 1. A hostile forwarding routine (an example of a forwarding routine potentially capable of consuming all available resources at a node.).

of a particular end-user is likely to be roughly proportional to the bandwidth offered to that user. There is a direct correlation in the case of buffer storage and link utilization; the CPU cycles required for forwarding a packet are likely to involve a constant per-packet cost for the routing, and a copying cost proportional to the bandwidth. Thus, by limiting the bandwidth that a user receives, a network provider may limit the amount of resources consumed by that user on a network node.

Breslau and Shenker [6] address the question of whether resource reservations are required in passive networks. Different classes of applications and load distributions are considered, and expressions are derived for the additional bandwidth required for a best-effort network to provide equivalent service to a reservation-capable network. No definitive conclusions are presented about whether future networks should be reservation-capable. However, the authors note that the greater the unpredictability of the offered load, the greater the performance advantage of a reservation-capable network over a best-effort network; in particular, with an exponential or algebraic [1] load distribution, the additional factor of bandwidth required by a best-effort network can increase without bound as the base bandwidth increases.

In [7] traffic traces were studied, with the conclusion that much of the WAN traffic in the Internet could not be modeled with a Poisson inter-arrivals process, but instead exhibited distributions with much larger variances and self-similarity [8], [9]. Although it is difficult to predict the load distributions faced by future networks, such results suggest that reservations will be necessary, at least for certain classes of traffic.

Providing programmable platforms within the network greatly increases the ways in which end-users may consume resources—the loads generated on resources such as CPU cycles, memory and outgoing link bandwidth may be totally unrelated to the incoming link bandwidth. At the extreme, hostile, greedy, or careless forwarding code could potentially consume all available resources at a node. For example, within an active network node, allowing arbitrary customization of the forwarding code used to process packets would permit (if specific controls were not put in place) a single packet to potentially consume all available CPU, memory and link bandwidth at a node (see Fig. 1).

The solution, we believe, is to constrain *real* resources associated with active network programs, based on the authentication offered by the principal (e.g., a "user") supplying the program, and the amount that the principal is willing to pay for resources. For example, a user may request a certain fraction of the bandwidth on a link or cycles of a CPU, in each case with a specified maximum level of jitter. Such control requires support from the operating system; fortunately, a number of new operating systems [10], [11] have appeared which provide the services necessary to contain one or more executing threads within a single scheduling domain.

[1] An algebraic distribution has a high variance in offered load; the probability that k flows are requesting service is $P(k) = \frac{\nu}{\lambda + k^z}$.

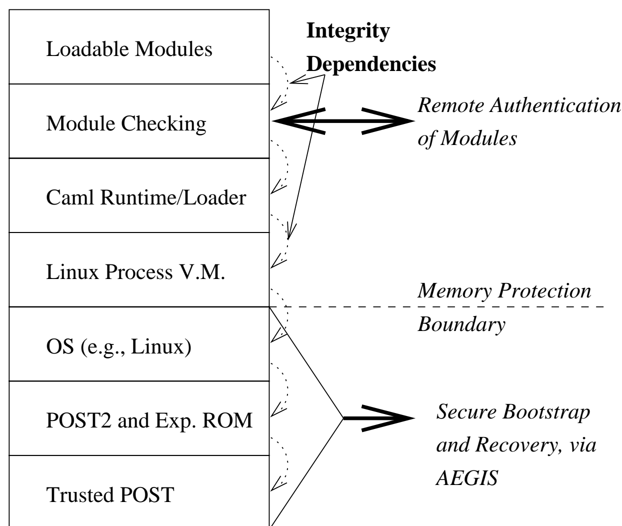


Fig. 2. SANE architecture.

We designed SANE/RCANE trying to neutralize, or at least mitigate, some of these threats. We made use of cryptographic primitives, language mechanisms, and appropriate software engineering and protocol design principles. The following section present a brief overview of SANE/RCANE.

III. OVERVIEW OF SANE

This section presents a brief overview of the architecture of SANE. For detailed information, please refer to [1]–[3]. The components of SANE are illustrated in Fig. 2. SANE provides security from the moment power is applied to the active node. This is accomplished using the AEGIS Secure Bootstrap Architecture which is able to detect alterations in the firmware and within the operating system. See [12], [13] for further detail on AEGIS.

Clients of SANE have access to several cryptographic primitives. These are DES [14] for symmetric key encryption, SHA-1 [15] for keyed hashes, and DSA [16] for public key signatures. (Other equivalent primitives could be provided.) All the algorithms have been implemented in Caml but due to performance degradation, we use a C version of SHA-1. Access to this implementation of SHA-1 occurs through a Caml interface, taking care to avoid potential bypassing of the type system. For more details on SHA-1 performance, see Section V-E.

Our architecture is based on public and private keys. Through the use of certificates, these allow specification and enforcement of security policies. Once two nodes have established a trust relationship, they can commence exchanging authenticated and/or encrypted packets. Please see the references for more details.

The basis of SANE's control of the system is its control of what modules are loaded, and by whom. SANE associates cryptographic certificates with modules. SANE can either require

a certificate for loading a particular module, or may allow universal loading of the module. Examples where such universal loading may be useful include low-cost operations like `ping`, as well as the security operations used for bootstrapping the security relationship with remote switches. There are two classes of certificate which can be presented by a user packet requesting access to a resource via a module. An *administrative* certificate allows loading of any or all modules into the system; it is intended for management and emergencies as might arise, and can be thought of as analogous to a "master key" granted by the switch administrator. More commonly, certificates are used to permit loading of selected modules. Once loaded, the certificates can then be used by the runtime system to allocate the specified amount of resources; for instance, the thread scheduler may terminate a thread that has executed longer than its certificate values allow.

IV. OVERVIEW OF RCANE

RCANE allows providers of nodes in a programmable network to permit untrusted clients to run code on their nodes, without the risk of Denial of Service attacks or excessive consumption of resources. It provides abstractions to control access to CPU cycles, network bandwidth and memory, and allows lightweight and flexible communication between clients.

A. Architectural Principles

RCANE is designed to provide resource isolation between multiple independent applications on a node in a programmable network, with the resources consumed by each application being paid for by a remote principal. The following aims underlie the design of the architecture:

- To provide guarantees to applications that they will receive the Quality of Service that they require in order to complete their tasks in a timely manner.
- To accurately account resource consumption to the client that causes such consumption to occur, in order that the client may later be billed.

B. System Structure

RCANE employs both horizontal layering (between different layers of trust) and vertical isolation (between different clients).

B.1 Layering

RCANE follows the principles proposed in [17] to partition the system into multiple layers:

- The *Runtime* is written in unsafe native code and provides access to—and scheduling for—the resources on the node. Services such as garbage collection (GC) and thread synchronization primitives are also provided by the Runtime.
- The *Loader* is written in a safe language (as are all higher levels). The Loader is entered early in system initialization. It is responsible for:
  — completing the initialization of the Runtime,
  — loading the higher levels of the system, and
  — linking user-supplied code into the system.

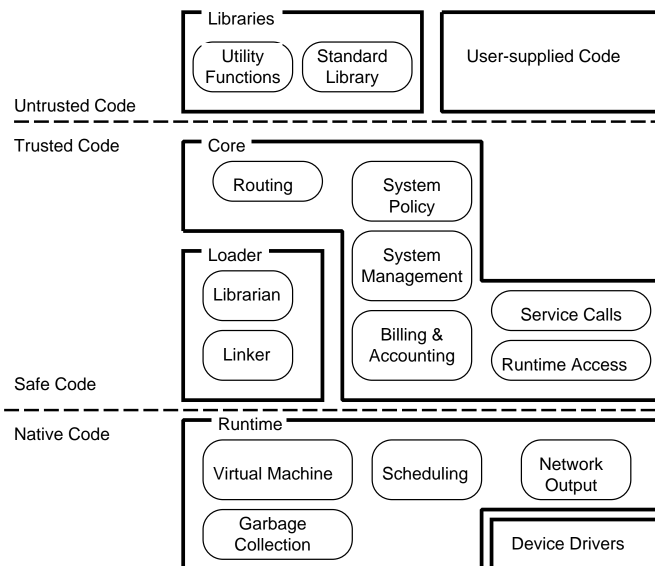


Fig. 3. RCANE architecture overview.

- The *Core*, loaded at system initialization time, provides safe access to the Runtime and the Loader and performs admission control for the resources on the node. The interface to the Core represents the "Red Line" identified in [18] as a requirement for security in a system using software protection.
- *Modules* are units of untrusted code. They include standard *libraries*, supplied by the system and loaded at system initialization time, and code supplied by remote users. They have access to the interfaces exported by the Core, but no direct access to the Runtime or the Loader except where permitted by the Core.

System modules in the Core are linked against entry points in the (unsafe) Runtime; these are then exported through safe interfaces to which the untrusted modules can link directly. The Runtime performs a policing function on the use of the node's resources. An overview of the RCANE architecture is shown in Fig. 3. The Safe/Native code boundary indicates the division between unsafe native code (written in a language such as C) and code written in a safe language supported by the Runtime's virtual machine. The Trusted/Untrusted code boundary indicates the division between code that is known to respect the security properties of the node and other code; such code may be regarded as untrusted if it is supplied by an untrusted source, or if it is from a trusted source but has not been sufficiently checked to ensure that it would not compromise the system. Within the Core and the Loader, although all code is written in a safe language, the interfaces exported by the Runtime permit complete control over the node[2]. Thus it is important that malicious code not be permitted to execute within the Core.

A safe language is one in which the language's typesystem is strongly enforced by the compiler, hence making it practical to use software, rather than hardware, mechanisms to protect the host machine from malicious code. For example, in Caml or

[2]In particular, some of the low-level features of the Runtime may permit language safety to be compromised.

Java, an untrusted program may be passed a pointer to an object containing system state; the abstract interface exported by the object will limit the operations that the program may perform on the object to that set approved by the system implementor. Such an approach could not work in C, as the compiler would permit you to cast the object pointer into a byte array pointer, and hence read or modify any part of the object. In order to make use of the type-safe properties of a language in this way, it is necessary to be able to verify that the code supplied by the untrusted user does indeed respect the typesystem of the language; such verification is often done by constructing a formal proof for the supplied code, or by having a trusted compiler sign the generated code.

B.2 Sessions

RCANE uses the abstraction of a *Session* to represent a principal with resources reserved on the node. A session is the analog of a *process* in a conventional OS that uses hardware protection, and is similar to the concept of a *flow* in the Active Networks NodeOS [19]. Sessions are isolated, so that activity occurring in one session cannot prevent other sessions from receiving their guaranteed QoS, except in situations where explicit interaction is requested (e.g., due to one session using services provided by another session).

To provide guaranteed levels of QoS to remote principals, RCANE allows sessions to reserve resources in advance. Requests for resource reservations are processed by the System session (see below) and, if accepted, are communicated to the Runtime's schedulers. In general, data-path activity—e.g., sending packets—is carried out within the originating session.

In other active network systems, the main resource principal is the execution environment (EE). This can lead to QoS crosstalk between the different clients of an EE. The use of sessions in RCANE makes the end-user the resource principal, allowing guarantees to be made more easily to individual end-users. An EE then becomes a library that a session may use to provide a convenient programming abstraction, and a client may make use of more than one EE in a single session if desired.

At system initialization time two distinguished sessions are created. The *System* session represents activity carried out as housekeeping work for RCANE and has full control over the Runtime. Many of the control-path services exported from the Loader and the Core are accessed through communication with the System session. The *Best-Effort* session represents activity carried out by all remote principals without resource reservations. Packets processed by the Best-Effort session supply code written in a very restricted language and are given minimal access to system resources.

Fig. 4 shows how RCANE sessions are orthogonal to the layering described in Section IV-B.1. The horizontal dashed lines indicate boundaries of *trust*; the vertical dashed lines indicate boundaries of *resource isolation*. It can be seen that some portions of the Core—such as those dealing with data-path activity—are directly accessible to user sessions; untrusted code may call them directly, possibly resulting in a direct call to the Runtime. The majority of the Core code executes in the System Session and thus is not directly accessible to the user sessions; such separation is achieved by isolating the heaps of the different sessions.

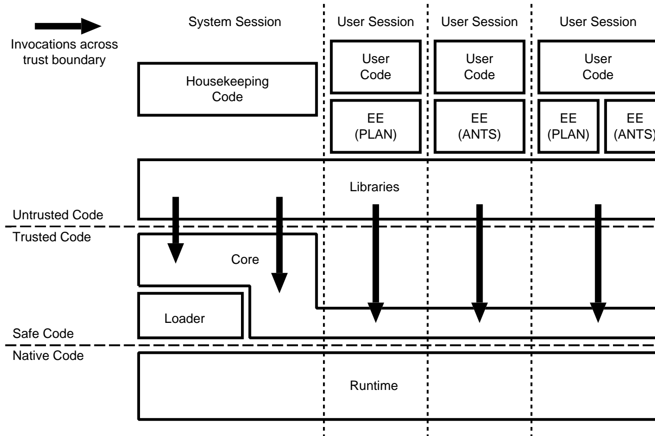


Fig. 4. Orthogonality of sessions and layering in RCANE.

It can be seen that each user session has created an instance of either the PLAN [20] or ANTS [21] execution environments, or both. Note that although multiple sessions are using each EE, these instantiations are operating independently and without QoS crosstalk, since resource scheduling occurs in the Runtime, below the level of the EEs.

C. CPU Management

RCANE's CPU management abstractions are structured so as to allow sessions to split their tasks between multiple scheduling classes, control the level of concurrency used for different sets of tasks, and to service those tasks efficiently. Three important abstractions employed are:

1. A *virtual processor* (VP) represents a regular guaranteed allocation of CPU time, according to some scheduling policy (e.g., EDF [22] or WFQ [23]). All activities carried out within a single VP share that VP's CPU guarantee. A session may have one or more VPs; by requesting multiple VPs, a session may organize its tasks into multiple independently-scheduled classes.
2. A *thread* is the basic unit of execution, and at any time may be either *runnable* (working on computation), *blocked* (e.g., on a semaphore, or awaiting more resources to become available) or *idle* (in a quiescent state, awaiting the arrival of further work items).
3. A *thread pool* is a collection of one or more threads. Each thread is a member of one pool. A thread pool acts as a queueing and dispatch point for *events*. An event represents a callback into one of a session's functions, either due to an incoming packet or an application-specified alarm. Each pool is associated with a particular VP; its threads are only eligible to run when its VP receives CPU time.

D. Network I/O

Access to network flows is essential to allow mobile code to communicate both with its original source and with the other resources in the network with which it wishes to interact.

RCANE allows sessions to open *channels* to give access to network resources. A channel is a simplex or duplex flow of packets to and/or from the network. A channel may have a guaranteed level of buffering and transmission bandwidth[3], to provide QoS for both incoming and outgoing streams.

Each channel that is capable of receiving packets has associated with it a demultiplexing specification (to select the incoming flow of packets to be directed to that channel) and a VP, which is used for RX protocol processing on that channel.

One of the significant sources of QoS crosstalk in a traditional operating system is the networking stack. In particular, the use of kernel threads to perform protocol processing can make it difficult to correctly account the resources consumed by a particular flow, and can lead to livelock situations[4] [24].

To prevent crosstalk between the network activity of different clients, all packets are demultiplexed to their receiving pools by the Runtime using a packet filter. Protocol processing is not performed on packets before demultiplexing. Each channel contains a FIFO queue of received packets – if this queue fills up, subsequent packets arriving for that channel will be discarded until such time as space becomes available in the queue. At a later time, when the VP associated with the channel is eligible to receive CPU time, protocol processing occurs:

1. The packet is removed from the channel's packet queue.
2. Any required processing such as defragmentation or checksum validation is performed.
3. If specified by the application, further demultiplexing into sub-flows may be performed, based on the value of specified key fields within the packet data.
4. The contents of the packet are encapsulated in a callback event to the function specified for the particular flow or sub-flow. Incoming packets for each thread pool are stored in a per-pool packet queue.

The time required for this protocol processing is entirely accounted to the VP associated with the channel, and thus to the session that opened the channel.

E. Memory

The RCANE memory architecture is based around the model of multiple garbage collected heaps in a single virtual machine. Each session is given its own independent heap. The maximum reserved size for this heap may be configured by the session, by requesting a particular size from the Core.

An incremental garbage collector—which processes a small portion of the heap each time it is invoked, rather than processing the entire heap in a single invocation—is employed to prevent excessive interruptions to execution. Such a property is essential to prevent the unpredictable nature of garbage collection from causing clients to miss their deadlines. Each session may tune the parameters of GC activity—such as frequency and duration of collection slices—in order to trade off responsiveness against overhead.

[3]In its current incarnation, RCANE schedules link bandwidth and buffering at a given node. A topic for future work is to extend the reservations provided by RCANE to permit end-to-end network QoS between different active nodes.

[4]Livelock occurs when little or no useful work is performed processing received data, as the system is too busy processing (and throwing away) more incoming data.

Charging is based on the size of the reserved memory blocks that comprise the heap, rather than the amount of live memory within those blocks, simplifying the accounting process and more accurately reflecting the load placed on the system's memory by each client.

Since each session executes in its own heap, for security direct communication between sessions via procedure calls is forbidden. Instead RCANE provides the mechanism of *inter-session services*, which allow secure control switching between sessions. Due to space constraints this mechanism is not discussed here.

More detail on the architecture and implementation of RCANE may be found in [5], [25].

V. IMPLEMENTATION AND PERFORMANCE OF SANE

A. The SwitchWare SANE Implementation

While the Secure Active Network Environment (SANE) architecture is portable across many active networking environments, our experimental prototype is constructed in the context of the SwitchWare active network architecture. SwitchWare is based on the approach of using restricted semantics to contain the behavior of potentially mischievous programs. This has the benefit that enforcement of restrictions can be performed once at compile or link time, resulting in a lower cost than an operating systems approach such as memory protection which requires repeated checks at runtime[5]. These semantic restrictions depend on the integrity of other system components such as the operating system, shared libraries, etc. The semantic restrictions are enforced with a strongly-typed language which supports garbage collection and module thinning [26].

ALIEN [27], [17] is one of the systems built within the SwitchWare project. ALIEN provides the ability to build prototype active networking systems based both active packets and active extensions. Active code is written the Caml language. For SANE, we have extended ALIEN to make use of the security infrastructure.

For our experimental network we used a cluster of DEC Alpha 21164SX machines, with 533 MHz processors and 64 MB memory each, connected via 100 Mbps switched Ethernet. All the test machines were running RedHat Linux, kernel version 2.0.33, and a modified Caml 1.0.7 runtime system. For additional details of the configuration, see [17].

To generate accurate timing measurements, we use the cycle counter available via the `rpcc` instruction. This is a 32-bit counter which increments once each clock cycle, thus giving a period slightly over 8 seconds. Back to back calls to `rpcc` will show a difference of two cycles. In C code, we call `rpcc` directly and attempt to print out results at times when the system is not being measured to minimize overhead. In Caml code, we use the locally written `Time` library. A start call immediately followed by a stop call averages between 1.5 and 2 μs (or about 500 times slower than `rpcc`).

The activity of the garbage collector varied throughout the

[5]Modern architectures have been optimized to handle memory protection with reasonable efficiency. However, some costs are unavoidable.

Table 1. Time to SHA-1 hash 4 MB of data.

Caml	Int32	bytecode	86.446289 s
		native	61.991894 s
	Alpha ints	bytecode	36.027246 s
		native	2.477051 s
C			0.333212 s

tests. Most of our tests were made without regard to the actions of the garbage collector, as we would expect to be the case if our system were being used in a production environment. However, in seeking the causes of some of the behaviors we observed, we did alter the behavior of the garbage collector to understand its contribution to the costs of the system. In the descriptions below of the tests that we ran, we mention any special settings of the garbage collector.

B. Cryptographic Primitives

Table 1 shows the costs of the main cryptographic primitive used by SANE. Each was implemented twice based on two different sets of integer primitives. This is because the garbage collector requires a bit from each integer to use as a tag bit. Thus, we have made use of a package called Int32 which supplies full 32 bit integers on both Pentium and Alpha platforms (with additional space overhead); using this package allows a single implementation of our cryptographic routines which will run on either platform. (As the table shows, this portability can come at a substantial cost in performance.) Finally, in addition to the bytecode interpreter which we use, the Caml distribution also provides a native code compiler which produces Alpha executables. Table 1 gives the average time in seconds to hash a 4 MB string using either the Int32 package and using the 63-bit integers provided by Caml on the Alpha. Additionally, it shows the difference in cost between compiled and interpreted code. Fig. 5 shows the average time to hash different block sizes using the Caml (compiled to native code) and C implementations of SHA-1.[6]

In practice, to use the dynamic loader in Caml, we must use the bytecode interpreter. Because the byte code version of SHA-1 has such a high cost and because that cost would be borne at least twice by every authenticated packet, we have resorted to a C implementation of the hash algorithm. While this greatly speeds the authenticator generation and verification operations, it may interfere with the Caml runtime thread scheduler. Specifically, when the end of a quantum occurs, if the current thread is executing C code, no call to the scheduler occurs and the thread will get an extra quantum. Furthermore, when using a C code implementation, we cannot catch type errors internal to that code, nor take advantage of the garbage collection mechanism available in the runtime. For these reasons, we tried to limit the amount of non-Caml code in our system, so we opted to keep the Caml DSA and DES implementations. In the future, we intend to investigate the feasibility of statically integrating

[6]Since these measurements were originally taken, the Caml developers have added their own Int32 package to the standard library. The result is substantially faster results on the native code version of the Int32 test (under 10 s). Nonetheless, it is still not nearly fast enough to change our conclusions about using C rather than Caml for this critical function, particularly since we do not see such a speedup for the bytecode version of the function.

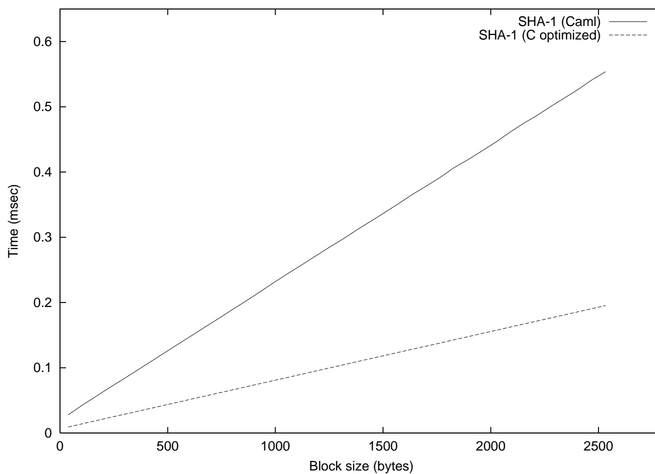


Fig. 5. SHA-1 cost.

Caml native code into the bytecode interpreter in the same way that we currently are able to integrate C code. This would allow us to regain the advantages of strong types and garbage collection with a more acceptable overhead, as can be seen in Fig. 5.

The key exchange protocol was also implemented in Caml as a three step protocol. In the first two messages, a list of SPKI-like certificates encoded as a string is exchanged. The third message contains a single certificate. Since the SPKI [28] format has not been fully specified, we designed our own certificate format in the same spirit. The protocol was designed to be fail safe [29] under all circumstances. In the presence of loosely synchronized clocks, it becomes fail stop (meaning that active attacks, including replays, on the protocol, are always detected). We encode all fields in the certificates as strings before transmission, and for signing and verification purposes. This allows us to avoid complicated marshaling issues. The average execution time of KEP with a 256-bit Diffie-Hellman exponent is 2.4 seconds, and with a 1024-bit exponent, 4.8 seconds. In both cases we used a 1024-bit modulus. This time is comparable to that of the IPsec key management protocols, Photuris [30] and IKE [31].

C. Cost of Active Ping

To understand the cost imposed by authentication, we measured the cost of sending an active ping. This ping was generated at a source machine, transmitted over a crossover cable via 100 Mbps Ethernet to the target machine, loaded and evaluated, then sent back to the source machine, where it was again loaded and evaluated.

The compiled byte code file, `saneping.cmo`, is 2230 bytes long. (When we are timing `saneping.ml`, it is slightly longer as the code to call the timing routines is added.) This results from about 60 lines of code. Transmission requires two Ethernet frames; fragmentation and reassembly is taken care of by UDP which we use as a "link layer" in this experiment. The results are described in Section V-E.

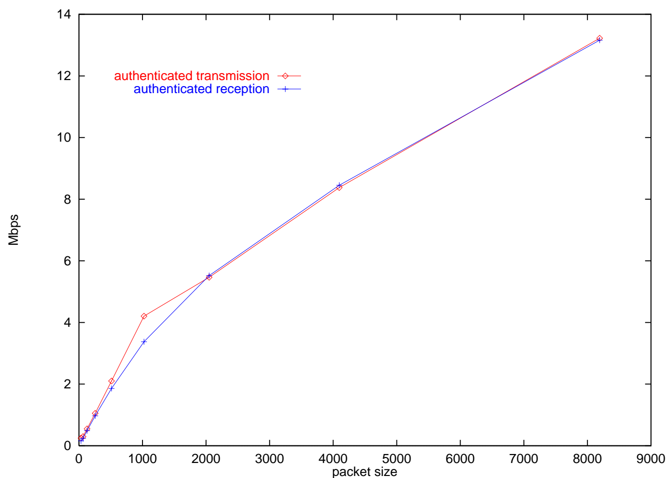


Fig. 6. Bandwidth of authenticated active packets.

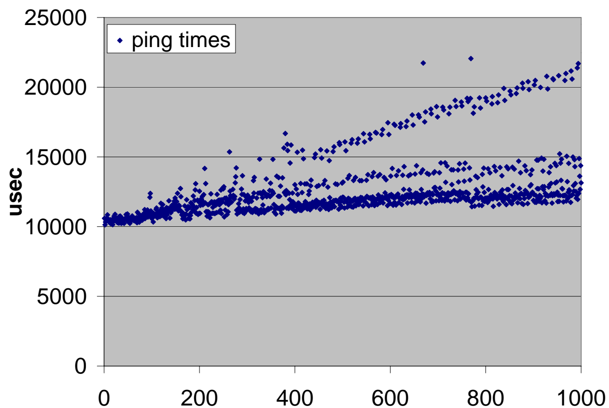


Fig. 7. Times for saneping.

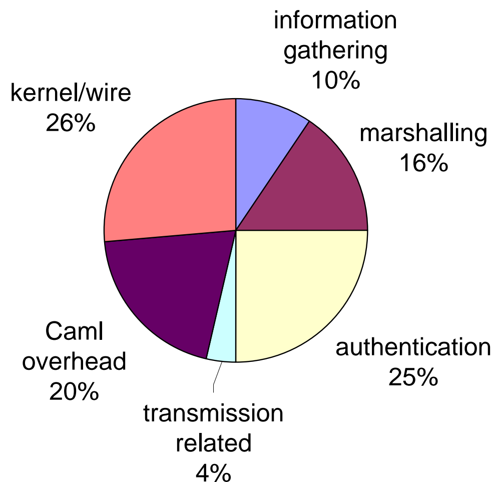


Fig. 8. Categorized costs for saneping.

D. Bandwidth Testing

We have also built an experiment to test the bandwidth available with active packets. This experiment sends 101 packets from the originator to the receiver. On the receiver, we check to see if all packets were received. If not, we add delays between the transmission of each packet until we do receive each packet. We then examine the time from when we started sending data until the kernel had accepted all the data on the sender and the time from when the first packet arrived until the time the last packet arrived on the receiver. (For all packet sizes greater than 32 bytes, we had to use a delay of 1–2 μs. This results in an error in measuring the receiving side because we include one extra delay after the last packet in the measurement.) Fig. 6 shows the bandwidth in megabits per second.

E. Performance of SwitchWare SANE

We have inserted timing points into ALIEN and into saneping to find out where the costs in the system are. We then classify these costs into several divisions. Based on these divisions, we can make an assessment of how these costs can be ameliorated.

Fig. 7 shows the timings for a set of 1000 pings. Both the slowly rising nature of the data and the line of outliers are described below. The next several sections describe how the tests were conducted and the results that we observed.

E.1 Breakdown of Costs

To better understand the behavior of our system, we used our timing infrastructure to time sections of the code. For each test, we arranged to time a single section of the code. We then started ten pings with sufficient delays between them to ensure that no two pings were being processed simultaneously.

Times reported are medians unless we specify otherwise. As Jain [32] describes, for a skewed distribution, the median is the best indicator of central tendency. Generally speaking, we tend to see data which clusters with very few points below the cluster, but with outliers above the cluster. These outliers occur because of events like resizing of tables or garbage collection. We also report the Semi-Interquartile Range (SIQR) or scaled SIQR when warranted.[7]

Fig. 8[8] shows the breakdown of these costs into categories. We have divided the code executed to process saneping into six categories. We then categorized the timing results based on which of these categories best described the activity being timed. Note that because of the additive nature of the errors introduced by our test infrastructure, smaller elements of the graph may be overrepresented. Nonetheless, the ordering of the elements should be correct.

The largest of these categories is "kernel/wire" which is the time spent in the kernel and in transmission between the two systems. We measured a median time of 3078 μs (with scaled SIQR 1.65%) for this value. We believe that some of this cost could be reduced either by running the Caml runtime in kernel mode or by using the techniques proposed by Brustoloni and Steenkiste [33]. Additionally, we believe that it would be possible to create a compiler which optimized for byte code size, which would also reduce this cost somewhat. A motivated programmer could also optimize the byte codes by hand (but this seems contrary to the advances that compilers have made since

[7] Recall that the quartiles Q_1 and Q_3 are the points such that 25% is less than or equal Q_1 and 75% of the data is less than or equal to Q_3. The SIQR is $(Q_3 - Q_1)/2$; it gives some notion of how dispersed the data are. The scaled SIQR is the SIQR divided by the median and can be expressed as a percentage. A low value indicates tight clustering of the quartiles around the median; conversely, a high value indicates dispersed data.

[8] Values rounded and do not total 100%.

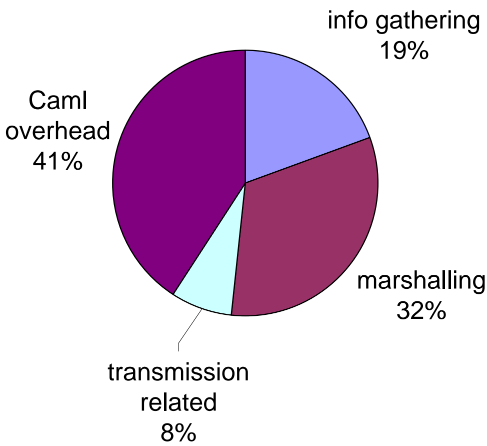


Fig. 9. Controllable costs for saneping.

the 1970s). This is probably the area over which the programmer has the least control.

The next largest overhead is due to authentication. Therefore, each time a packet is sent and received, the SHA-1 hash is computed, which accounts for practically all of the authentication cost. In fact, of the total cost of our active ping, 33% is spent on authentication. As long as this is the case, authentication needs to be a rare operation in high performance applications. Two primary approaches to avoiding too frequent authentications are caching as in the ANTS architecture or designing a domain specific language as in the PLAN architecture. Caching has the advantage that, to the extent that active code is reused by a flow, the cost of authentication (and several of the other costs we discuss here) can be amortized. A domain specific language such as PLAN can reduce the need for authentication and thus can be used by applications that have very dynamic needs that result in constantly changing active code.

Another approach to reducing the cost of authentication and transmission is a reduction in the size of the transmitted active code. As seen in Fig. 5, the cost of authentication increases linearly with the size of the data to be authenticated/verified. (For keyed-hash or MAC (Message Authentication Code) type of authentication, the process of "signing" and verifying is the same.) Thus, if we can reduce the size of the transmitted packets, we can reduce the cost of authentication. This can be achieved by data compression or (in the case of active code) by a compact bytecode, such as Spanner [34]. Other ways of speeding up the authentication include hardware assistance, cryptographic coprocessors (possibly on the network card, or even entirely outside the system), and faster software-authentication algorithms.

In Fig. 9, we have removed the kernel/wire and authentication elements to focus on the costs imposed by ALIEN. We examine each of these in turn.

E.2 Caml Overhead

The largest of the costs from ALIEN is the category which we have called Caml overhead. This consists of the cost to link a byte code file into the running system. When using active packets, this cost occurs on every node visited and so is particularly important. We have not evaluated less easily measured costs such as the overhead of using byte code and this interpreter or the relative efficiency of the Caml libraries for the task at hand. Overall, we measured the cost of loading a byte code file (on each of the two machines along the path) to have a median time of 1148 μs with a scaled SIQR of 2.29%.

We also ran tests to more tightly characterize some effects of the runtime system. These tests were run with an enlarged minor heap and invocations of the major garbage collector at times which minimize garbage collection activity during the timing period.

In particular, as described in [17], we divided the code used to load a byte code file into sections and timed these sections. We found several undesirable characteristics. One of the most significant features of Fig. 7 is that the ping time rises in later tests. This is caused by the growth of the symbol table, which is stored as a hash table. Each new active packet declares the same set of symbols, which shadow the previous definitions. This causes a problem because, whenever a new entry is inserted into the hash table, the insertion routine counts the number of elements in the bucket by traversing a linked list. This is done to determine whether it is time to resize the hash table. The occasions when the hash table is resized cause the lower line of outliers in Fig. 7. Additionally, if a symbol from a library should hash to the same bucket as one of the symbols from ping, it will be at the "bottom" of the bucket.

The upper set of outliers is caused by a similar problem. There is an array which contains global information used by the linker, to which each ping adds new entries. (In fact, the entries shadow the entries from the previous ping in our case. Because this is not typical behavior for a program, the runtime system makes no attempt to find this sort of shadowing.) Every 12 to 13 pings, the array fills and must be resized. Moreover, the data in the old array must be copied to the new array which accounts for the steep slope of the line through this set of outliers.

This example illustrates one of the weaknesses of our approach in ALIEN: by choosing an existing language, we gain a large infrastructure, but elements of that infrastructure are designed for a different problem than ours. In several cases, the implementors of Caml have assumptions that are different from ours. We have identified two categories of differing assumptions: assumptions about amortization and assumptions about time scales. Garbage collection of the minor heap illustrates both of these points. While copying garbage collection can take less time than comparable dynamic memory management using an explicit system like malloc [35], this is often accomplished by "freeing" many allocations simultaneously. For a long running, computationally bound program, it is reasonable to assume that the user is only concerned with time to completion and so this sort of amortization is appropriate. Even with an interactive program, if the length of the pauses can be made short enough so that the user does not perceive them, this approach works well. The key here is that with a single user, amortization reduces the costs paid by the user. In contrast, in ALIEN with multiple users, amortization may mean that some switchlets see an *increased* cost as they pay not only for freeing their own allocations, but also those of other switchlets. Moreover, because switchlets may be sensitive on much finer time scales than a human can perceive, the level of acceptable jitter in ALIEN is lower than would generally be the case.

```
module An_marshal = struct
  type packet_data = { code : string; data : string; func : string }
  let encode pkt = Marshal.to_string pkt []
  let decode str off = Marshal.from_string str off
end
```

Fig. 10. Signature for An_marshal.

In the case of garbage collection, incremental collection techniques based on Baker's algorithm [36] or the techniques from [37] and [38] can be helpful. The major heap is collected using such a incremental technique. Another approach to the problem described in [39] uses a separate minor heap for each thread thus solving the amortization problem (as well as allowing access to multiple processors). Nonetheless, any language to be used for Active Networks must be examined closely for instances of these assumptions. This is particularly true for languages designed for general purpose use where such assumptions are generally reasonable.

More generally, if the same program is to process different data repeatedly, if we adopted a means of caching and reusing active packets such as the facilities provided by ANTS [21], this would also solve both of these problems. Additionally, it would reduce our kernel/wire costs, our authentication costs, our Caml overhead, and some of our marshaling costs. Since these are our four most expensive activities, such an approach seems warranted. Moreover, it would reduce some of our unmeasured costs. For example, with more reuse, we should reduce the amount of memory allocation and hence the frequency of our garbage collections. If dynamic code generators or just-in-time compilers cause the automatic generation of active packets to become common, however, the costs that we have outlined will be important.

E.3 Marshaling

The next largest set of costs are due to marshaling data. The difficulty comes in making sure that the switchlet can only do this in a secure manner while at the same time striving to deliver adequate performance.

For example, Caml provides a module called `Marshal` which allows complex data objects to be transformed into strings and back again. However, these functions do not perform checking to ensure that data objects created this way are valid on this machine. Thus, they provide a means to subvert the type system which undermines our security. More concretely, thread IDs are internally a pointer to a (C) struct containing information about that thread. If an attacker were able to guess the address of the descriptor for a thread that he wanted to attack then it becomes simple to create a string which has the bit pattern which corresponds to that address. A call to `Marshal.from_string` would transform this into a valid `Thread.t`. Since opaque types are intended to be unforgeable, we use them as capabilities; the attacker could now kill the thread described by the forged ID. For this reason, we do not make the `Marshal` module available to switchlets except in limited circumstances.

Fig. 10 shows `An_marshal` which is a module we provide which allows transformations between structures containing three strings and a string which can be sent over the network. Because the only data objects which can be created are strings, we avoid any security holes.

In cases where we need to marshal other data, we use *ad hoc* methods usually based on `Printf.sprintf`. For example, when SANE sends a certificate, the SHA-1 values which describe the programs which are authorized are first converted from a binary value to ASCII strings. Similarly, to send the starting time for saneping, we use `sprintf` with a `%f` conversion to create an ASCII string that we can send over the network. When the ping returns to the originator, we use a combination of regular expression routines and string to float conversions to recapture the original value so that we can find the latency.

Unfortunately, this has a significant performance penalty. `An_marshal.decode` has a median cost of 77 μs; `ping_decode` has a median cost of 162 μs. Similarly, `An_marshal.encode` has a median time of 72 μs as compared with 189 μs for `ping_encode`.

E.4 Information Gathering

This category consists of the time spent by the active packet gathering information about its environment. Thus, asking the `Route` module for the next hop on the route to the destination or finding the address of the current host falls into this category. We have also included the cost of reading from disk the bytecode file for the program portion of the active packet. Overall, these operations have a median cost of 1094 μs.

The cost of reading-in the bytecode file, however, is driven by whether it is in the buffer cache of the host. (Linux maintains an in-memory cache of recently used file blocks. Our test file is small enough and our test machine has enough memory that we believe the first read of the file caused the kernel to go to the disk to get the file whereas for subsequent reads of the file, the kernel was able to find the file in memory.) A typical cost for reading the byte code file for the first ping was 39,280 μs. The median cost was 1043 μs. Further, if we calculate tendencies for this cost for the pings after the first, we find a median of 1040 μs and a mean of 1031 μs. The scaled SIQR is 2.55% and the standard deviation is 39 μs.

E.5 Transmission Related Costs

The final section is transmission-related costs. This is the cost of calling `Udp.sendto_udp` and queuing a packet for processing on the receiving side. Some of the time from the former is also measured in the kernel/wire category.

While this is the section with the lowest cost, there is a potential source of improvement. We use queues to pass packets up to the thread that has registered an interest in these packets. It has been suggested that if we instead allow the thread to register an upcall, we save the cost of queuing and dequeuing the packet and possibly of a context switch. Our concern is that doing so allows a malicious or buggy thread to "capture" the thread

that is intended to retrieve packets from the interface. Finding a scheme that allows recovery in the case of such a misbehaving thread, but which normally has only the cost of the upcall approach is an interesting area for continued study.

VI. IMPLEMENTATION AND PERFORMANCE OF RCANE

A prototype of RCANE was developed over the Nemesis Operating System [11]. Nemesis was chosen as the base platform for its support for low-level of abstraction, allowing better control over the resources available to the RCANE system. Experiments were performed on Intel Pentium II machines running at 300 MHz connected to 100 Mbps Ethernet and Intel PentiumPro machines running at 200 MHz, connected to 10 Mbps Ethernet.

A. CPU Scheduling

The fundamental advantage of a programmable network over a passive network is that end-users may perform computations at nodes within the network, and hence receive lower latency than if all interactions were required to take place between the end-user and the ultimate destination. In order to provide low latency, it is vital that the user-supplied applications running on the programmable node receive timely access to the CPU. Alternatively, if the application is attempting to process or filter some form of multimedia data, it should receive regular access to the CPU in order to prevent excessive jitter in its results. Thus an important feature of RCANE that must be shown is that it allows users to request and receive the guaranteed access to the CPU that they require.

To demonstrate the effectiveness of the CPU guarantees provided by RCANE, the time received by a set of sessions loaded over the network on to an RCANE node was logged. Four sessions were created at various times through the experiment. Each session used a single VP and was CPU bound.

All four sessions were started with no particular guarantee, but with access to best-effort time. Sessions B, C, and D were created at 3 second intervals following the creation of session A, and requested 10%, 20%, and 30% shares of the CPU, respectively, each with a period of 4ms and with no access to extra time.

Fig. 11 shows the percentage of the total CPU that each VP received over each scheduling period (i.e., between consecutive deadlines). The scheduling period for each VP is used since this is provides an accurate view of whether the contractual guarantee made to the session is being honored. Since session A's VP is running without a guarantee, it is assigned a notional scheduling period of 100 ms; hence, the trace shown is clearly coarser than the traces for the other sessions, whose VPs have fine-grained guarantees.

B. Network Transmission

Fig. 12 shows a trace of network output from three sessions, each attempting to transmit continuously.

- Session D has no guaranteed bandwidth.
- Session E has a transmission scheduling period of 6ms, during which it receives 2 ms worth of link bandwidth. This 33% share equates to 33 Mbps on the 100 Mbps link used for this experiment.
- Session F has a 6 ms transmission scheduling period, and a dynamically changing guarantee (see below).

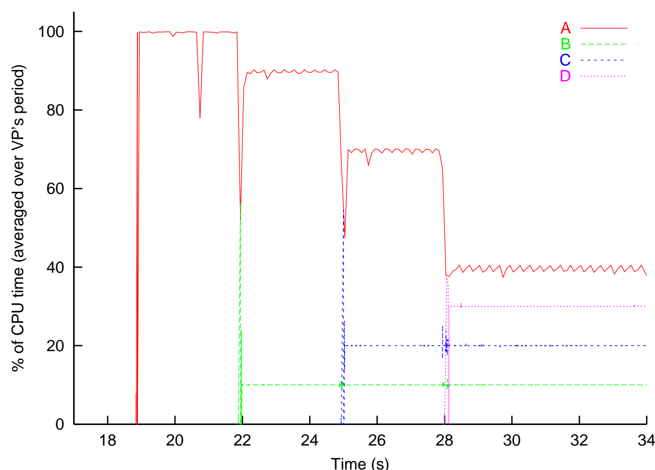


Fig. 11. CPU consumption by a set of sessions.

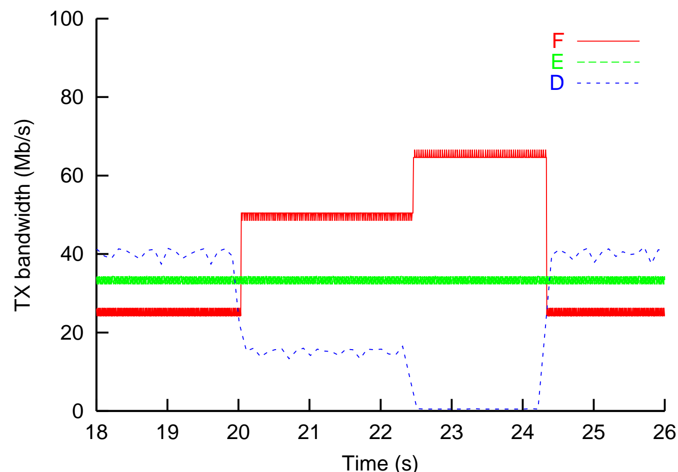


Fig. 12. Network transmission rates.

Session F starts with a guarantee of 25%. 20 s after the start of the experiment it requests 45%, thus reducing the best-effort bandwidth received by D. After another 2 s, it requests 65%. Now the link is saturated and there is no best-effort transmission time available for D. After a further 2 s it returns to 25%, allowing D to begin transmitting again. It can be seen from the trace that the desired resource isolation is achieved.

The apparent noisiness of the traces shown for the sessions with guaranteed resources is due to the quantization effects caused by the use of large (1500 byte) packets and small transmission periods. The network transmission guarantee given to session E, 33 Mbps guaranteed over a 6 ms period, is equivalent to $16\frac{2}{3}$ packets per scheduling period. Since it is not possible to preempt access to the network while a packet is in the process of being transmitted, sessions alternately overrun their guarantee in one period, and then receive correspondingly less in the following period, such that when the traces are averaged over two or

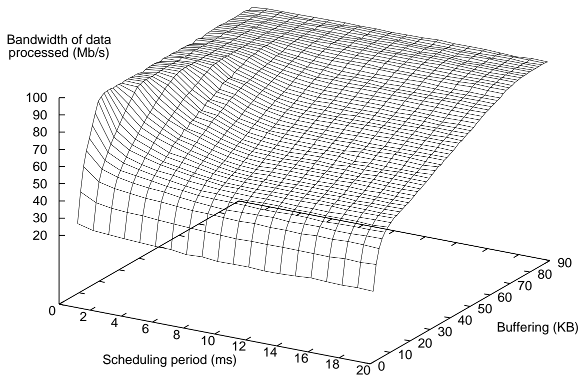


Fig. 13. Network receive bandwidth as a function of CPU scheduling period and buffering.

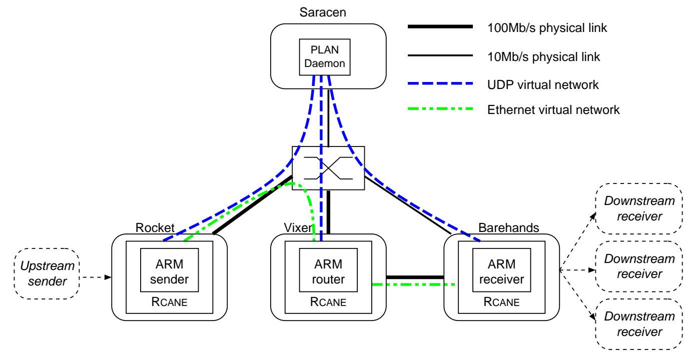


Fig. 14. Experimental topology for ARM testing.

more periods, the oscillation becomes insignificant. In the case of session E, it manages to transmit either 16 or 17 packets in each period. This effect could be reduced through the use of a networking technology that uses a smaller maximum packet/cell size.

C. Network Reception

On a shared, unscheduled medium such as Ethernet it is not straightforward to provide guarantees on the number of packets received at a node for a particular client. However, the level of guaranteed buffering and CPU time that a particular client receives will affect the amount of incoming data on a network stream that actually reaches the client. To demonstrate this, the bandwidth of data that could be processed by a session running on an RCANE node was measured.

A session on a neighboring node (belonging to the same principal, and bearing the same session ID) transmitted approximately 97 Mbps across the intermediate link. A CPU bound session with a 40% guarantee and a 10 ms period is also running on the receiving node. The buffer space reserved by the receiving session was varied between 1.5 KB and 88 KB; it was guaranteed 10% of the CPU, with a scheduling period varying between 1 ms and 20 ms. 10% of the CPU had been observed to be more than sufficient to process the entire incoming data stream on an otherwise unloaded node.

Fig. 13 shows how the level of buffering and CPU scheduling period affect the amount of data that can be processed. It can be seen that when the session runs with a short scheduling period, it is scheduled sufficiently frequently to process all the incoming data even when its buffering is relatively low. Similarly, when the session has large amounts of buffering, it is able to process all the data even if its scheduling period is long.

However, the CPU-bound session has a guaranteed slice of 4 ms every 10 ms; as the receiving session's scheduling period increases, it finds itself interrupted for longer periods of time, and thus for modest levels of buffering the sustained receive bandwidth falls—the receiving session loses access to the CPU for long enough stretches of time that its buffers are filled up, resulting in incoming packets being dropped in the network device driver. As the receiving session's period increases significantly beyond the 4 ms slice that is being received by the CPU-bound session, no further reduction in processed bandwidth occurs.

It may also be seen that at very low levels of buffering, the amount of data that can be processed drops off rapidly due to the inherent latency in the Nemesis I/O channels between the network device driver and RCANE.

D. Memory Isolation

D.1 Memory Consumption

In [40], the Active Reliable Multicast protocol (ARM) was proposed to improve the performance of reliable multicast flows. ARM caches packets at intermediate nodes within the network, trading memory on the nodes for reduced bandwidth requirements and latency; as such, some form of resource control is required to ensure that greedy clients do not consume excessive amounts of memory. A simple implementation of ARM was developed to run over RCANE.

RCANE provides callbacks to indicate to a session that it is close to exhausting its reserved heap size; when ARM receives such callbacks, older packets and older structures describing NACKs received and pending retransmissions, are recycled to reduce garbage collection overheads or released to be reused as a different type of object. ARM records the numbers of NACKs received, and the number of times that the packet requested by a NACK was not found in the cache. At intervals it checks the ratio of these; if the ratio is too high, it deduces that the amount of caching it is using is insufficient to effectively deal with the loss on the downstream links, and (assuming that the client is willing to pay for an additional memory reservation) requests a higher memory reservation from RCANE.

Fig. 14 shows the experimental setup used to demonstrate the use of ARM over RCANE. Rocket, Vixen and Barehands are each running RCANE and are connected by 100 Mbps Ethernet. Saracen is running a PLAN daemon and is used to control the RCANE machines. The link between Vixen and Rocket is assumed to be non-lossy. The link between Vixen and Barehands is artificially made very lossy, by dropping packets at Barehands with a given probability. Sessions on all three nodes are started by a client who wishes to transfer an ARM flow from upstream of Rocket to downstream of Barehands. Thus Rocket is acting as

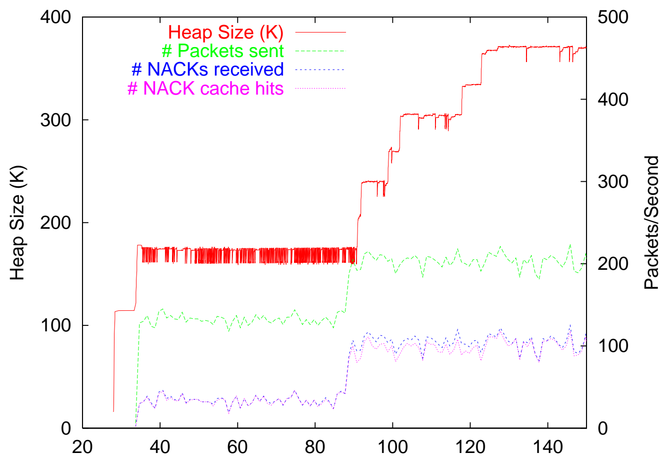
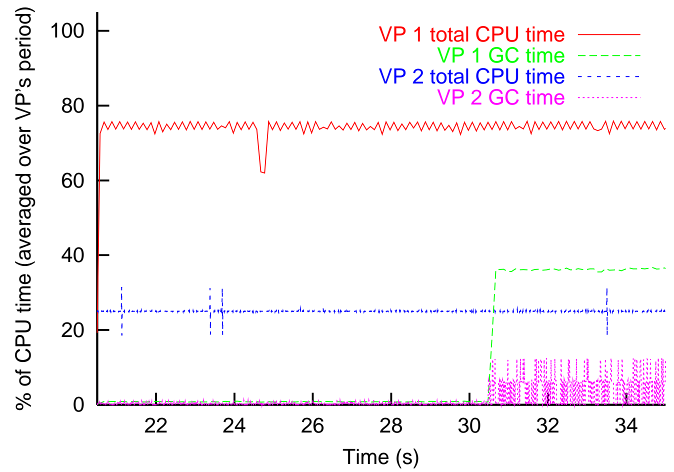
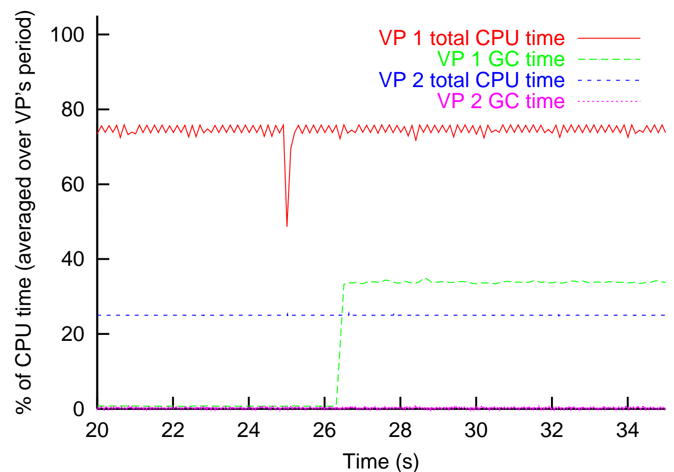


Fig. 15. Memory usage, packet counts and cache hits for an ARM stream.



(a)



(b)

Fig. 16. Avoiding QoS crosstalk due to garbage collection.(a) Single heap; (b) Separate heaps

an ARM sender, Vixen as an ARM router and Barehands as an ARM receiver[9]. Initially, 25% of packets from Vixen to Barehands are dropped; after 90 seconds 50% are dropped. Vixen is configured to attempt to increase its reserved heap size if it discovers that it is servicing less than 90% of the NACKs from its cache over a 1 second period.

Fig. 15 shows the memory usage of the ARM router on Vixen, along with the total number of packets sent, the number of NACKs received from Barehands and the number of cache "hits" (NACKs that could be served from the cache). It can be seen that the heap consumed by ARM quickly grows to approximately 180 KB; at this level, Vixen is able to service over 90% of the NACKs that it receives from Vixen directly from its cache. After 90 seconds the loss on the link from Vixen to Barehands doubles to 50%. At this point, the number of NACKs received from Barehands increases significantly, and the percentage of NACKs serviced from the cache falls noticeably below 90%. Thus ARM increases its reserved heap size in an attempt to reduce the number of misses; the heap size grows to approximately 370 KB over the course of 40 seconds, at which point the level of NACK hits reaches the target of 90%, and the heap size stabilizes.

This demonstrates the ability of RCANE to constrain the amount of memory used by different clients on a node—an essential property of any practical programmable network platform.

D.2 Garbage-Collection Isolation

RCANE runs each session in a separate heap in order to eliminate crosstalk caused by garbage collection—a session should not be inconvenienced due to the allocation behavior of other sessions. To demonstrate the utility of this approach, two scenarios were considered. In Fig. 16(a), VPs A and B are running in the same session. Initially both are generating small amounts of garbage. After a period of time, A begins generating large amounts of garbage. Scenario Fig. 16(b) is the same, but with the two VPs running in separate sessions (and hence having separate heaps). Fig. 16 shows the outcome of these scenarios. In Fig. 16(a), both VPs are initially doing small amounts of GC work. A is running best-effort, whereas B has a guarantee of 1ms in each 4ms period. When A switches to generating large amounts of garbage, the time it spends garbage collecting increases substantially. However, as shown by the noisy region at the bottom right of the graph, B also ends up doing an irregular but substantial amount of GC work. Although B has its own independent CPU guarantee, at times critical GC activity (such as root tracing) is taking place when its thread is due to run; this GC work must be completed before normal execution can be resumed. In Fig. 16(b), B is unaffected by the extra GC activity caused by A, since it is running in a separate session and hence does not share its heap; thus B's own GC activity remains

[9]Clearly in a real situation Rocket would be receiving from an upstream node, and Barehands would be sending to a set of downstream nodes; this was not practical due to lack of suitable networking infrastructure.

minimal.

VII. DISCUSSION

We have described the architecture of two active networking systems, SANE and RCANE, which have a shared heritage in ALIEN. The key contributions of ALIEN were its proof that a general purpose computing model was applicable to active networking, and for security, that privilege boundaries were distinct from mutability boundaries. This resulted in a system with three distinct levels: 1) an immutable privileged loader; 2) a mutable privileged "Core Switchlet," the loading of which is the major role of the loader; and 3) unprivileged active code. Loaded modules are constrained via "module thinning" which can be viewed as a form of namespace sandboxing; the major idea is that the restriction of the namespace serves (along with privilege) to restrict access to named resources.

ALIEN's form of protection is effective on a closed active node, ignoring competition for bandwidth and assuming the integrity of the node. SANE's advances were in two areas: 1) guaranteeing the integrity of control transfers until ALIEN gains control of the node, and 2) providing remote access to the namespace via ALIEN-authorized cryptographic credentials. We evaluated the cost of implementing SANE using a small active application, that of an active network echo packet ("saneping"), and discovered that the cost of the cryptographic operations on either a per-byte or a per-packet basis comprised a major portion of the measured delays, up to about 25%. SANE pays about a 25% delay penalty for its extended security. SANE's mechanisms depend on a modified BIOS and operating system boot block for the chained layered integrity checks, but do not otherwise depend on the operating system, other than demanding that it be available as a target for lower layers of the system. SANE intends to pass control to ALIEN in our current implementation, but could as easily use a JVM or ANTS. SANE is completely oblivious to the use of Linux as a platform for its Caml implementation.

RCANE has addressed the availability of service to applications by its management of resources in the time domain. Our measurements have clearly demonstrated that the system effectively partitions active applications from each other, with its use of separate heaps and Nemesis scheduling technologies. Further, active applications are protected from many forms of denial of service attacks and other bandwidth-starvation schemes, as we showed in our end-to-end measurements. RCANE is the first active networking system which pulls together the namespace protection schemes of ALIEN and the management of real resources derived from Nemesis.

The way forward in systems development is the augmentation of the RCANE implementation of an active networking environment on Nemesis with the security features of SANE. This implies that the BIOS of machines so configured would be modified to support the secure bootstrap, the boot block of the Nemesis boot disk would be modified, etc. The bootstrap would end in the Nemesis system executing RCANE. The cryptographic support and credentials used by SANE are easily imported into RCANE, as RCANE's active networking programming environment is essentially Caml. The result is a threat-secure active networking node implementation.

VIII. RELATED WORK

Ongoing work on active networks has been surveyed elsewhere [41], [42] and thus we focus on issues germane to SANE and RCANE.

An architecture which, like from ALIEN to SANE, extended a protection model from the local domain to a distributed environment was provided by Sansom, *et al.* [43], who enforced protection locally with memory-protection enforced capabilities. (It is notable that capabilities can be viewed as a namespace-based protection mechanism). The capabilities were extended to remote nodes via cryptographic means. SANE provides more general mechanisms and could thus be specialized to such an application (moving memory-protected objects about the network) but more importantly guarantees local integrity before extending itself into the network.

ANTS [44] uses MD5 hashes ("fingerprints") to name on-demand loaded modules, the hashes provide unique names rather than security. ANTS performance is limited by its use of the Java programming language for protection; ANTS use of "code caching" can be used to amortize the cost of cryptography across multiple uses of a module. The remote authentication and namespace security of SANE can be used to support ANTS and similar systems, and RCANE supports ANTS, making its resource control available with the ANTS namespace isolation model.

Another, quite different, approach to providing secure active networking is that used by the Programming Language for Active Nets (PLAN). PLAN is a special-purpose programming language appropriate for per-packet programs. PLAN's semantics are purposely restricted to operations which are safe and bounded in resource usage, with the intention of being so lightweight that any node would be willing to run PLAN packets, including those from remote nodes, and thus would not require the security of SANE. However, as any enhanced services are added to the node as PLAN services, such services would require a SANE-like approach for security. RCANE supports PLAN, and thus its features and model can be coupled with robust resource control of memory, bandwidth and processing. The interesting issue in such a construction would be the coupling of PLAN's language-supported "resource bound" with the controls supported by RCANE.

Finally, the Active Network Node Operating System ("NodeOS") provides support for Execution Environments such as ALIEN, PLAN and ANTS. The architecture provides considerable support for resource management, and in fact has incorporated ideas such as the thread model from RCANE. It may be interesting to compare the performance of our current SANE and RCANE systems with versions using implementations of a NodeOS when they become available.

IX. CONCLUSIONS

This paper has made two contributions.

First, it has presented the design and implementation of two active networking systems, SANE and RCANE, which have ex-

plored different aspects of security. Each was based on a common framework, ALIEN, developed as part of the SwitchWare project. SANE added a secure bootstrap and cryptographic protocols to ensure node integrity and preserve ALIEN's namespace protection. RCANE used the Nemesis operating system as a basis for resource control unavailable in ALIEN's initial Linux implementation, while adding support for separate heaps and garbage collection.

The second contribution is detailed measurement studies of SANE and RCANE, showing the measured costs of their security enhancements. The costs of various cryptographic operations were studied independently, and then the entire SANE system was instrumented and measured using applications such as an "Active Ping." Using PCs with extremely high performance arithmetic (533 MHZ Alphas) and native mode cryptographic implementations to make these operations as fast as possible, we discovered that the basic operations required for authenticating packets require a 33% overhead relative to unauthenticated packets. The RCANE measurements confirm that a flexible high performance active network node architecture can provide good resource control and isolation with appropriate operating system support.

We see three promising research areas, beyond the system development discussed in the previous section, that have been exposed by our results. First, much of the cost of authentication is a "per-byte" cost, and thus reducing the size of the packet (*e.g.,* by use of a "very-high-level language" (such as PLAN) or some other compression scheme) may allow a wider range of the active packet/active extension continuum to be exploited by programmers. (An alternative is the clever code-caching scheme of ANTS, which on average would pay the authentication cost once.) Second, it would be of interest to replicate the experiments reported here in other environments, such as those used in related work. The major architectural points we have made will almost surely remain true, but it would be valuable to study other environments to split out their detailed cost/performance tradeoffs. Finally, performance versus security tradeoffs must be reexamined as each new active application is developed.

X. ACKNOWLEDGEMENTS

We would like to thank Bill Marcus for his help in writing some of the original ANEP code, and Mike Hicks for the discussions and tools he provided us for performance analysis.

We would also like to thank Simon Crosby, Scott Nettles, and Marianne Shaw for their discussions and insights into this work.

REFERENCES

[1] D. S. Alexander *et al.*, "A secure active network architecture: Realization in SwitchWare," *IEEE Network, (Special issue on Active and Programmable Networks)*, vol. 12, pp. 37–45, May/June 1998.

[2] D. S. Alexander *et al.*, "Safety and security of programmable network infrastructures," *IEEE Commun. Mag., (Special issue on Programmable Networks)*, vol. 36, pp. 84–92, Oct. 1998.

[3] D. S. Alexander *et al.*, "A secure active network environment architecture," *Technical Report MS-CIS-97-17*, Pennsylvania Univ., Nov. 1997.

[4] D. S. Alexander and J. M. Smith, "The Architecture of ALIEN," in *Proc. the 1st International Working Conf. Active Networks (IWAN '99)*, Lecture Notes in Computer Science, July 1999.

[5] P. Menage, "RCANE: A resource controlled framework for active network services," in *Proc. the 1st International Working Conf. Active Networks (IWAN'99)*, Lecture Notes in Computer Science, July 1999.

[6] L. Breslau and S. Shenker, "Best-effort versus reservations: A simple comparative analysis," in *Proc. ACM SIGCOMM*, vol. 28, Sept. 1998.

[7] V. Paxson and S. Floyd, "Wide-area traffic: The failure of poisson modeling," in *Proc. ACM SIGCOMM*, vol. 24, Sept. 1994, pp. 257–268.

[8] W. E. Leland *et al.*, "On the self-similar nature of Ethernet traffic," in *Proc. ACM SIGCOMM*, vol. 23, Sept. 1993, pp. 183–193.

[9] M. Crovella and A. Bestavros, "Explaining world wide web traffic self-similarity," *Technical Report 95-015*, Boston Univ., Aug. 1995.

[10] A. B. Montz *et al.*, "Scout: A communications-oriented operating system," *Technical Report*, Department of Computer Science, Arizona Univ., June 1994.

[11] I. M. Leslie *et al.*, "The design and implementation of an operating system to support distributed multimedia applications," *IEEE J. Select. Areas Commun.*, vol. 14, pp. 1280–1297, Sept. 1996.

[12] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proc. IEEE Symposium on Security and Privacy*, May 1997, pp. 65–71.

[13] W. A. Arbaugh *et al.*, "Automated recovery in a secure bootstrap process," in *Symposium on Network and Distributed System Security*, Internet Society, Mar. 1998.

[14] "Data encryption standard," *Technical Report FIPS-46*, U.S. Department of Commerce, Jan. 1977.

[15] "Secure hash standard," *Technical Report FIPS-180-1*, U.S. Department of Commerce, Apr. 1995. Also known as: 59 Fed Reg 35317, 1994.

[16] "Digital signature standard," *Technical Report FIPS-186*, U.S. Department of Commerce, May 1994.

[17] D. S. Alexander, *ALIEN: A Generalized Computing Model of Active Networks*, Ph.D. Dissertation, Pennsylvania Univ., Sept. 1998.

[18] G. Back and W. Hsieh, "Drawing the Red Line in Java," in *Proc. the Seventh Workshop on Hot Topics in Operating Systems (HOTOS-VII)*, Mar. 1999.

[19] "*L Peterson, ed.* NodeOS Interface Specification, AN Node OS Working Group," Jan. 2000. Draft.

[20] M. Hicks *et al.*, "PLANet: An active internetwork," in *Proc. IEEE INFOCOM'99*, Mar. 1999.

[21] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse, "ANTS: A toolkit for building and dynamically deploying network protocols," in *Proc. OPENARCH'98*, Apr. 1998.

[22] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. the ACM*, vol. 20, pp. 46–61, Feb. 1973.

[23] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *Proc. ACM SIGCOMM*, vol. 19, Sept. 1989, pp. 1–12.

[24] P. Druschel and G. Banga, "Lazy receiver processing (LRP): A network subsystem architecture for server systems," in *Proc. the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, USENIX, Oct. 1996, pp. 261–275.

[25] P. Menage, *Resource Control of Untrusted Code in an Open Programmable Network*, Ph.D. Dissertation, Cambridge Univ. Computer Lab., June 2000.

[26] F. Rouaix, "A web navigator with applets in Caml," *Fifth WWW Conf.*, May 1996. Available at http://pauillac.inria.fr/mmm/papers/mmm.ps.gz.

[27] D. S. Alexander *et al.*, "Active bridging," in *Proc. ACM SIGCOMM*, Sept. 1997.

[28] C. M. Ellison *et al*, "Simple public key certificate." Apr. 1997, work in progress.

[29] L. Gong and P. Syverson, "Fail-stop protocols: An approach to designing secure protocols," in *Proc. IFIP DCCA-5*, Sept. 1995.

[30] P. Karn and W. Simpson, "Photuris: Session-key management protocol," RFC 2522, Mar. 1999.

[31] D. Maughan *et al.*, "Internet security association and key management protocol (ISAKMP)," internet–Draft, IPSEC Working Group, June 1996.

[32] R. Jain, *The Art of Computer Systems Performance Analysis*, New York: John Wiley & Sons Inc., 1991.

[33] J. C. Brustoloni and P. Steenkiste, "Application-allocated I/O buffering with system-allocated performance," *Technical Report CMU-CS-96-169*, Carnegie Mellon Univ., Pittsburgh, PA., Aug. 1996.

[34] B. Schwartz *et al.*, "Smart packets for active networks," Jan. 1998. Available at http://www.net-tech.bbn.com/smtpkts/smartpkts-index.html.

[35] A. W. Appel, "Garbage collection can be faster than stack allocation," *Information Processing Lett.*, vol. 25, pp. 275–279, June 1987.

[36] H. G. Baker, "List processing in real-time on a serial computer," *Commun. ACM*, vol. 21, no. 4, pp. 280–94, 1978.

[37] S. M. Nettles and J. W. O'Toole, "Real-time replication garbage collec-

tion," *SIGPLAN Symposium on Programming Language Design and Implementation*, pp. 217–226, ACM, June 1993.

[38] J. O'Toole and S. Nettles, "Concurrent replicating garbage collection," *ACM Symposium on LISP and Functional Programming*, ACM Press, June 1994.

[39] M. Shaw, "An architecture for an active network node," Master's Thesis, Pennsylvania Univ., Philadelphia, Dec. 1998.

[40] L. Lehman, S. J. Garland, and D. L. Tennenhouse, "Active reliable multicast," in *Proc. IEEE INFOCOM'98*, Mar. 1998.

[41] D. L. Tennenhouse *et al.*, "A survey of active network research," *IEEE Commun. Mag.*, pp. 80–86, Jan. 1997.

[42] J. M. Smith *et al.*, "Activating networks: A progress report," *IEEE Computer*, vol. 32, pp. 32–41, Apr. 1999.

[43] R. D. Sansom, D. P. Julin, and R. F. Rashid, "Extending a capability based system into a network environment," in *Proc. ACM SIGCOMM*, Aug. 1986.

[44] D. Wetherall, "active network vision and reality: Lessons from a capsule-based system," in *Proc. the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, vol. 33, Dec. 1999.

D. Scott Alexander is currently Chief Architect at Activium, Inc. where he is involved in bringing Active Networking to the commercial market. Previously, he has worked for Bell Laboratories and the Jet Propulsion Lab. He earned his B.A. at Rice University and his M.S.E and Ph.D. at the University of Pennsylvania.

Paul B. Menage is a software engineer at Ensim Corporation, working on scalable ISP/ASP hosting technology. He received his B.A., M.A., and has recently completed his Ph.D., at the University of Cambridge. He is a member of the IEEE and ACM.

Angelos D. Keromytis is a Ph.D. candidate at the University of Pennsylvania. He earned a M.S. in computer science from University of Pennsylvania, and a B.S. from University of Crete, Greece. He is a member of IEEE, ACM, USENIX, and IACR.

William A. Arbaugh is an Assistant Professor of Computer Science at the University of Maryland, College Park, where his current research focus is computer and network security. Bill received his Ph.D. at the University of Pennsylvania. He has served as a senior computer scientist with the Research Group of the U.S. Department of Defense, and as a senior software engineer and a tactical communications platoon leader with the U.S. Army. He earned a M.S. in computer science from Columbia University, and a B.S. from the United States Military Academy. He is a member of the IEEE and ACM.

Kostas G. Anagnostakis is a Ph.D. candidate at the University of Pennsylvania. He earned a M.S. in Computer and Information Science from the University of Pennsylvania and a B.S. in Computer Science from the University of Crete, Greece.

Jonathan M. Smith is a Professor in CIS at the University of Pennsylvania, where he leads research in networking and security.