The Price of Safety in an Active Network

D. Scott Alexander
Bell Laboratories, Lucent Technologies
and
Kostas G. Anagnostakis
University of Pennsylvania
and
William A. Arbaugh
and
Angelos D. Keromytis and Jonathan M. Smith
University of Pennsylvania

This work was supported by DARPA under Contract #N66001-96-C-852, with additional support from the Intel Corporation.
Name: D. Scott Alexander
Affiliation: Bell Laboratories, Lucent Technologies
Address: salex@research.bell-labs.com
Biography: Scott Alexander is currently a Member of Technical Staff at Bell Laboratories. He earned his B.A. at Rice University and his M.S.E. and Ph.D. at the University of Pennsylvania.
Name: Kostas G. Anagnostakis
Affiliation: University of Pennsylvania
Address: anagnost@dsl.cis.upenn.edu
Biography: Kostas G. Anagnostakis is working towards an M.S.E. in Computer and Information Science at the University of Pennsylvania. He earned a B.S. from the CS Department, University of Crete, Greece.
Name: William A. Arbaugh
Affiliation: University of Pennsylvania
Biography: William A. Arbaugh 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.
Name: Angelos Keromytis
Affiliation: University of Pennsylvania
Address: angelos@dsl.cis.upenn.edu
Biography: 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.
Name: Jonathan M. Smith
Affiliation: University of Pennsylvania
Address: jms@central.cis.upenn.edu
Biography: Jonathan M. Smith is an Associate Professor in the Penn CIS Department. Jonathan was previously at Bell Telephone Laboratories and Bellcore, where he focused on UNIX internals, tools and distributed computing technology. He was also a member of a technology transfer team on computer security. He is a member of ACM and Sigma Xi, a Senior Member of IEEE, and has consulted extensively for industry and government. He has patented technology for key-agile encryptors using asynchronous (ATM) networks and ultra high-speed ATM encryptors.

Lack of security is a major threat to "Active Networking," as programmability creates numerous opportunities for mischief. The point at which programmability is exposed, *e.g.,* through the loading of code into network elements, must therefore be carefully crafted to ensure security.

This paper makes two contributions. First, it describes the implementation of a solution, the Secure Active Network Environment (SANE), which is intended to operate on an active network router. The SANE architecture provides a secure bootstrap process, which includes cryptographic certificate exchange and results in execution of a module loader for introducing new code, as well as a packet execution environment. SANE thus permits a direct comparison of security implications of active packets (such as "capsules") with active extensions (used for "flows" of packets).

The second contribution of the paper is a performance study using a combination of execution traces and end-to-end throughput measurements. The example code performs an "active ping" and allows us to break down costs into categories such as authentication. In our SANE implementation on 533 Mhz Alpha PCs, securing active packets effectively increases the time required to process a packet by a third. This result implies that the majority of packets must remain unauthenticated in high performance active networking solutions. We discuss some solutions which preserve security.

1. INTRODUCTION

Programmable network infrastructures offer a variety of possibilities for changing the processes of protocol development, network service deployment, and interoperation with host applications. Perhaps the most aggressive proposal for a programmable infrastructure is "active networks," where "programs" are loaded into network elements on-the-fly, providing rapid dynamic reconfiguration of the network infrastructure, on a per-user or a per-packet basis. The design space for active networks has many dimensions, but the most important are flexibility, security, usability and performance.

Flexibility and usability derive from the choice of programming language and programming environment. Among the available choices are Caml[Leroy 1995], Java[Arnold and Gosling 1996] and purpose-built programming languages such as PLAN[Hicks et al. 1998]. The portability and distributed programming support of Java have made it a popular choice for active networking environments.

The remaining important elements of the design space, performance and security, are both dependent on the language choice and have tradeoffs which are independent of the language choice. For example, a typesafe language such as Caml will preclude certain classes of errors from ever occurring (*e.g.,* stray or intentionally malicious memory references), increasing the security of Caml programs[Leroy and Rouaix 1999]. Authenticating packets of code will require a performance overhead independent of the properties of the language.

Before we discuss the performance versus security tradeoffs further, we will present our security model. At a high level, information security consists of getting the right information to the right person at the right time. At an appropriate and greater level of detail, this statement becomes a statement of a security *policy*, leaving any deviation from the policy defined as insecure. The very flexibility of an active networking infrastructure, since it expands the possibilities for mischief, expands the threat model posed to the network infrastructure. For example, "denial-of-service" attacks are possible against a multiplicity of resources such as CPU cycles, storage and output link bandwidth, since these are used by loaded programs. The essence of security is controlled access to resources, and in the active networking scheme, this means controlling the actions of loaded modules.

Since the programming languages and their support environments largely determine flexibility and usability, and significant performance increases will come from optimizing these environments (*e.g.*, [Hartman et al. 1996]), providing security expands the design space in a significant and necessary way. The goal of this paper is to outline the engineering tradeoffs between performance and security, and to point out the implications of these tradeoffs to architects of programmable network infrastructures.

Our approach was follows. We first outlined the threats, old and new, faced by an active network infrastructure. These are discussed in Section 2. Using this outline, we designed and implemented an infrastructure which provides the basic elements required for security guarantees within and between network elements, namely: a secure bootstrap; key exchange; authentication and identification of network entities; packet confidentiality and integrity; resource and access control; and name-space protection. Section 3 discusses this infrastructure, called the Secure Active Network Environment, (SANE) and explains how SANE is realized in an active networking model such as SwitchWare. Section 4 provides a detailed measurement study of the implementation, with particular attention focused on a cost breakdown for an example application, "active ping." We also report on another experiment, the "active firewall." Section 5 discusses those results and the various costs in our system. Section 6 reviews our contributions and those of related work, and Section 7 concludes the paper with a discussion of the implications of these results for designers of active networks and other systems with mobile code.

2. THREATS

An active network infrastructure is very different from the current Internet. In the latter, the only resource consumed by a packet at a router is the memory needed to temporarily store it and the CPU cycles necessary to find the correct route. Even if IP [Postel 1981] option processing is needed, the CPU overhead is still quite small compared to the cost of executing an active packet. In such an environment, strict resource control in the intermediate routers was considered non-critical. Thus, security policies [Atkinson 1995c] are enforced end-to-end. While this approach has worked well in the past, there are several problems. First, denial-of-service attacks are relatively easy to mount, due to this simple resource model. Attacks to the infrastructure itself are possible, and result in major network connectivity loss. Finally, it is very hard to provide enforceable quality of service guarantees [Braden et al. 1997].

Active Networks, being more flexible, considerably expand the threat possibilities. The security threats faced by such elements are considerable. For example, when a packet containing code to execute arrives, the system typically must:

—Identify the sending network element

—Identify the sending user

—Grant access to appropriate resources based on these identifications

—Allow execution based on the authorizations and security policy

In networking terminology, the first three steps comprise a form of admission control, while the final step is a form of policing. Security violations occur when a policy is violated, *e.g.,* reading a private packet, or exceeding some specified resource usage. In the present-day Internet, intermediate network elements (*e.g.,* routers) very rarely have to perform any of these checks. This is a result of the best-effort resource allocation policies inherent in IP networking.

In an environment where a considerable fraction (and perhaps eventually a majority) of the traffic will be continuous media traffic, security must include resource management and protection with an eye to preserving timing properties. In particular, a pernicious form of "attack" is the so-called "denial-of-service" attack. The basic principle applied in such an attack is that while wresting control of the service is desirable, the goal can be achieved if the opponent cannot use the service. This principle has been used in military communications strategies, *e.g.,* the use of radio "jamming" to frustrate an opponent's communications, and most recently in denying service to Internet Service Provider servers using a TCP *SYN* flood attack [Panix 1996; Daemon9 et al. 1996]. Another very effective (even crippling) attack on a computer system can occur due to scheduling algorithms which implicitly embed design assumptions.

With an active network element, it is easy to imagine situations where user programs (or errant system programs) run amok, and make the network elements useless for basic tasks. The solution, we believe, is to constrain *real* resources associated with active network programs. For example, if we limited the principal (*e.g.,* a "user") invoking the recursive shell script to 10% of the CPU time, or 10% of the system memory, the process would either limit its effects on the CPU to a 10% degradation, or fail to operate (since it could not invoke a new process) when it hit the table space limitation. Fortunately, a number of new operating systems [Montz et al. 1994; Leslie et al. 1996] have appeared which provide the services necessary to contain one or more executing threads within a single scheduling domain.

3. OVERVIEW OF SANE

The following subsections present the components of SANE and explain how they fit together. Figure 1 shows the various components of SANE and their dependencies. SANE provides security from the moment that power is applied to an active network node. This is done by using a secure bootstrap process that provides integrity guarantees for nodes firmware and operating system components. Once the operating system and active network environment have been verified, the static integrity guarantees of the system have been assured and we transition to our dynamic integrity mechanisms.

3.1 AEGIS

All secure systems assume the integrity of the underlying firmware, but typically cannot identify when this assumption becomes invalid. This inability to detect changes in the integrity state of the hardware and firmware results in a significant security problem. The AEGIS Secure Bootstrap architecture reduces the severity of this problem by providing static integrity guarantees of the bootstrap process. We define the static integrity property to mean that an object has not been altered while in storage or transit.

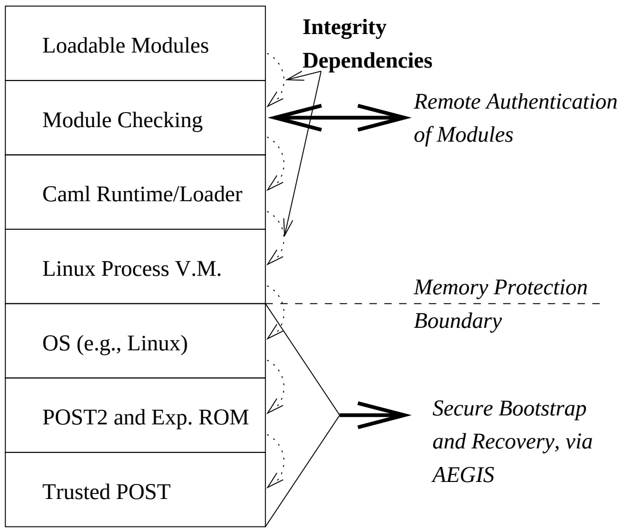

Fig. 1. **SANE Architecture**

AEGIS provides static integrity guarantees first by reducing the size of the firmware assumed as having the static integrity property down to the small section that tests the proper operation of memory and the motherboard. AEGIS then uses induction, digital signatures and modifications to the control transitions from major modules, *e.g.* CALL and JUMP instructions, to ensure the static integrity of the next module. We call the combination of these techniques Chaining Layered Integrity Checks (CLIC).

The goal of AEGIS is to prevent tampering of components that are considered trustworthy by the system administrator. AEGIS verifies the integrity of already trusted components. The nature of this trust is outside the scope of this paper. A more extensive review of AEGIS and a comparison with other secure-bootstrap efforts can be found in [Arbaugh et al. 1997; Arbaugh et al. 1998].

3.2 Cryptographic Primitives

SANE provides access to various cryptographic primitives. These can be used by other applications as-is or as building blocks for more complex protocols. These primitives are also used by SANE's more advanced services. The primitives initially provided are:

—symmetric key encryption (*e.g.,* DES [NBS 1977])

—(keyed) hashes (*e.g.,* SHA-1 [NIST 1995])

—public key signatures (*e.g.,* DSA [NIST 1994])

This set of primitives may be enriched in the future. 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. Support for cryptographic hardware in SANE is being considered. For more details on SHA-1 performance, see Section 5.

3.3 Public Key Infrastructure

In our architecture, every network entity (active switch or user) owns at least one private / public key pair. These keys (and the corresponding certificates) are used to authenticate these entities and authorize their actions. Although SANE depends on a public key infrastructure, it is not tied to a particular one. Certain features, such as selective authorization delegation, user defined authorizations and certificate revocation through expiration are desirable, but they can be simulated in any of they proposed public key infrastructures. In our environment, we use an attribute-based certificate format similar to the original SPKI[Ellison et al. 1997] proposal. For more details on the certificate format, see [Arbaugh et al. 1998].

3.4 Key Establishment Protocol (KEP)

A key element of SANE is the key establishment protocol. The protocol itself is a strengthened variation of the Station-to-Station [Diffie et al. 1992] protocol, which uses Diffie-Hellman [Diffie and Hellman 1976] key exchange and public key signature authentication. The goal of the exchange is to establish a shared secret and authenticate the two protocol participants (node-to-node or user-to-node). Once the key is established, the authorizations of each party are determined, through the exchange of the appropriate certificates. Examples of such authorization is the amount of memory or CPU cycles a user is authorized to use on the active switch. The derived shared key is used to authenticate and / or encrypt further communications between the two parties. For more details on the protocol itself, see [Arbaugh et al. 1998].

3.5 Packet Authentication and Encryption

Once two nodes have established a shared key, they can commence exchanging authenticated and / or encrypted packets. In SANE, we use the ANEP [Alexander et al. 1997] packet format over UDP. We have added an authentication header, which is similar in form and purpose to the one used in the AH IP Security protocol[Atkinson 1995a]. The authentication header offers data integrity and replay protection services. A similar header has been defined for encrypted packet exchange, again similar to the corresponding IPsec protocol[Atkinson 1995b].

It should be noted that the authentication offered by this service is end-to-end. It is fairly straightforward to extend the service to support digital signatures, and thus provide authentication of the initial packet origin.

3.6 Link Keys

When a SANE node boots, it attempts to establish shared keys with each of its neighbors. It does this by running the key establishment protocol already described. In the process, the identity of the neighbors is also verified. The administrator of an active network can essentially “freeze” the network topology by specifying which nodes can be neighbors. There are certain benefits in doing this:

—Certain distributed types of protocols (such as routing) can be secured against outside attacks

—The switch offers secure forwarding services to any active packet that requests them. This is important for mobile agent types of applications that cannot depend on end-to-end security, but require some security guarantees on a hop-by-hop basis.

—Administrative domains and their boundaries can be established through this process. We define an administrative domain as the set of active nodes that are managed by the same entity, have a common set of access and resource management policies and, after the KEP is run, trust each other (or at least some subset of the nodes in the domain) to make some security decisions on each other's behalf.

3.7 Administrative Domains

A user who needs to load a number of modules on a set of active nodes would typically have to contact each node individually and establish security associations (SAs) with each one. This establishment could happen in either a telescopic manner (where the user "explores" the network) or a parallel manner (if the user knows the identities of all the switches in advance). This can prove expensive both computationally (because of the public key operations) and in packet size (since there must be a separate authentication payload for each node that a packet may visit).

We define as an "administrative domain" a set of network nodes under the same administration, having the same security policy and some trust relationships with each other. Figure 2 shows a network consisting of a set of distinct administrative domains. Creation of an administrative domain involves providing the same security policy to all the nodes, and initiating pairwise authentication exchanges at boot time. By taking advantage of the existence of administrative domains, we could make some optimizations:

—Once the user has established an SA with some active node in another administrative domain, that node can act as a key distribution server (KDC) similar to Kerberos [Miller et al. 1987].

—Only nodes at the perimeter of an administrative cloud need verify the cryptographic integrity of packets. They can then specify what the active packet can do in the interior of the domain. In that respect, any machine at the edge of the domain can act as a firewall. In contrast to the Internet firewalls however, policy can be specified but not enforced at the edges; enforcement of access and resource management policies has to take place in the interior.

We implemented the active firewall function in an experimental setup. For more details, see Section 4.

3.8 Resource Control

Resource control on the active switch is imposed by the runtime system, as specified by the certificates exchanged during key establishment. The protected resources include access to standard and loaded modules, use of CPU cycles, access to and allocation of memory, the ability to send and receive packets, latency and bandwidth requirements, and others. It is a subject of further research exactly what the right resources are and how to resolve conflicting resource requests.

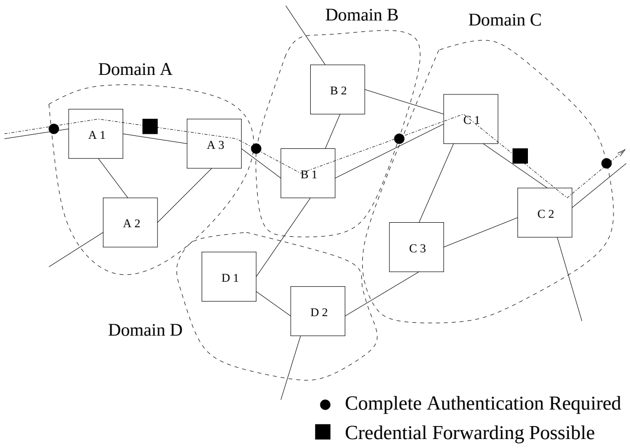

Fig. 2. **Administrative Clouds and Path Setup**

In any case, since a tenet of our approach is controlled loading of modules, SANE must manage loading modules in a secure fashion if it is to be useful in an active network. That is, it must control which 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.

3.9 Naming

Conceptually, loaded modules can be considered as the interfaces to user defined resources. Such resources will generally be shared between different sessions of the same principal, or even between different principals. These principals will need to identify (name) the particular resource they want to use.

The SANE naming service allows for unsupervised but collision-free[1] (secure) identification of programs. The basis of this approach is to combine hashes of the code, public keys (and signatures), and user-defined strings to generate "names" for pieces of code. Thus, a certain program can have different names, each with different semantics and trust dependencies. Such a service is necessary in an active

[1]To the extent that the cryptographic hash functions employed are resistant to collisions.

network environment where different users' modules can explicitly interact with, or even depend on, each other. For more details, see [Alexander et al. 1998].

3.10 Overview of SwitchWare

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[2]. 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.

ALIEN [Alexander et al. 1997; Alexander 1998] is one of the systems built within the SwitchWare project. ALIEN provides the ability to built prototype active networking systems based both active packets and active extensions. Active code is written the Caml language. ALIEN provides restricted access to the underlying system. For SANE, we have extended ALIEN to make use of the security infrastructure.

4. IMPLEMENTATION AND PERFORMANCE OF SANE

We have implemented SANE in the SwitchWare environment. For our experimental network we used a cluster of DEC Alpha 21164SX machines, with 533MHz processors and 64MB memory each, connected via 100Mbit 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 [Alexander 1998].

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 μsec (or about 500 times slower than rpcc).

The activity of the garbage collector varied throughout the 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.

[2]Modern architectures have been optimized to handle memory protection reasonably fast. However, some costs still remain.

4.1 Cryptographic Primitives

Tables 1, 2, and 3 show the costs of the three cryptographic primitives provided 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 tables show, 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 4MB 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. Figure 3 shows the average time to hash different block sizes using the Caml (compiled to native code) and C implementations of SHA-1. Table 2 shows the cost to encrypt a 4MB message using 63 bit integers with either the bytecode or native Alpha code. Finally, table 3 shows the cost in milliseconds of signing and of verifying the message "abc," using DSA. Since a DSA signature consists of computing a SHA-1 digest followed by the signature process itself, for a longer message, one should add the cost of performing the hash.

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

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

Caml	Alpha ints	bytecode	99.331543 s
		native	16.723242 s
C			1.0785348 s

Table 2. **Time to DES encrypt 4MB of data**

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-system errors internal to that code, nor take advantage of the garbage collection mechanism available in the runtime. For these reasons, we

sign	Caml	Int32	bytecode	27.089 ms
			native	12.954 ms
		Alpha ints	bytecode	20.907 ms
			native	11.855 ms
	C			2.800 ms
verify	Caml	Int32	bytecode	41.452 ms
			native	22.121 ms
		Alpha ints	bytecode	35.198 ms
			native	20.664 ms
	C			5.000 ms

Table 3. **Digital Signature Timings**

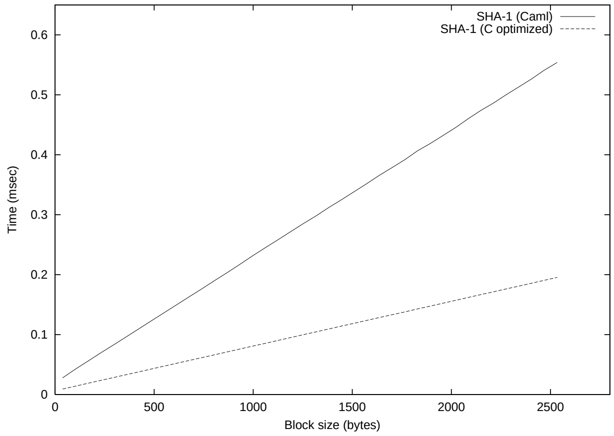

Fig. 3. **SHA-1 Cost**

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 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 Figure 3. We also believe that in the future, "Just In Time" compilation techniques can narrow this gap in performance.

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[Ellison et al. 1997] format has not been fully specified, we designed our own certificate format in the same spirit. The protocol was designed to be fail safe [Gong and Syverson 1995] 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 marshalling 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 [Karn and Simpson] and IKE [Maughan et al. 1996].

The certificate infrastructure we used in our setup is a shallow hierarchy. A small number of keys are considered as trusted to make statements about nodes or, more specifically, what the network topology is. These same keys are also used to certify users and specify their access rights on the active nodes. It is only a matter of policy however what sort of certificate method is followed. A cyclic graph-type (such as in PGP[Zimmerman 1995]) or a hierarchical approach (such as in X.509 [Committee 1989]) or any other method can be used. Furthermore, there is no need for an organization's internal certification policies to be the same as the interdomain and interorganizational policies.

4.2 Cost of Active Ping

To understand the cost imposed by authentication, we measured the cost of sending an active ping (provided as Appendix A for illustration). 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 5.

4.3 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 1-2μsec. This results in an error in measuring the receiving side because we include one extra delay after the last packet in the measurement.) Figure 4 shows the bandwidth in megabits per second.

4.4 Implementing an Active Firewall on SANE

While the task for legacy firewalls is to locally decide on whether to forward or drop packets traveling between an external, untrusted network and an internal, trusted network, we define the operation model of an active firewall as follows: an active firewall is an active node that is connected to multiple domains of different security policies and has to map the set of credentials being carried across domains to match policy and security principals in each domain.

In a simplified yet complete realization, our active firewall has to control (*i.e.,* restrict or explicitly allow) packet code attempting to execute in the secured part

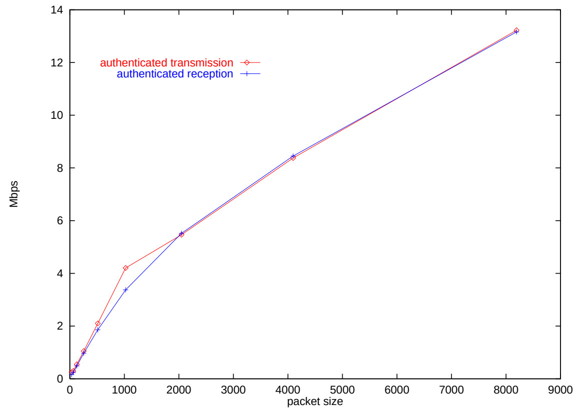

Fig. 4. Bandwidth of authenticated active packets

of the network. The secure nodes trust the firewall, which is expected to provide "guidelines" to the nodes with regards to access policy. Our active firewall then operates by attaching attribute certificates to active packet attempting to enter the trusted network. These certificates define the suggested access and resource usage policy for the internal node. Such policies can be:

—**DATA-ONLY**: the packet can only call data delivery primitives

—**ALL**: giving full access to the execution environment

—a string representing the primitives, modules and globals (including functions) the active code is allowed to use.

The certificate is signed by the firewall, and encoded into the ANEP packet header. Having been signed by the firewall, it is trusted by the internal nodes. While our current implementation supports per-packet authorization only, it is possible to provide more persistent authorization per module or sender (packet header) information. Furthermore, establishing a security association by sending a shared secret from the firewall to the internal node and using additional authenticators for that association in the header would be more desirable if associations have a longer life-cycle. Arriving at the final destination, the packet's encoded certificates are verified and the authorization string is extracted and passed along with the packet code to the Caml dynamic linker. We enhanced the linker to perform access checks on the incoming code relocation list, before proceeding with the actual linking. Since the relocation list is essentially the contact surface between the incoming code and the node execution environment, we believe that this approach offers efficient control and high security.

In our experimental topology, *kenny* is an active firewall between the unsecure network with two nodes (*mrhat*, *mrgarrison*) and the secure network with one node (*chef*). Both external nodes send active pings which, to be evaluated by *chef*, need access to modules "Safeloader" and "Saneproto." The firewall, according to its local policy , issues a certificate for packets coming from *mrhat only*, still forwards packets from both unsecure nodes to *chef*, which executes for the authorized packet

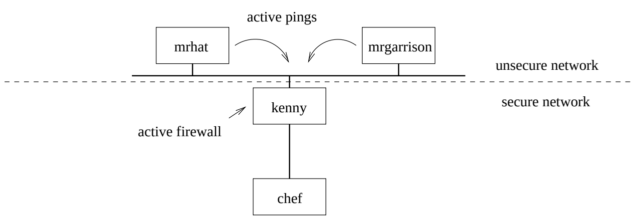

Fig. 5. **Active Firewall Experimental Set-up**

and raises an exception and logs the event for the packet coming from *mrgarrison*. In our case, any certificate issued by a principal other than the firewall would be invalid. The above experiment shows SANE's flexibility in providing security services. It is important to note that, policy can be either *suggested* or *enforced*, so that secure nodes may wish to maintain their own security policy and trust relationships, possibly overriding firewall decisions, a clear advantage over legacy firewalls in certain scenarios.

5. RESULTS

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.

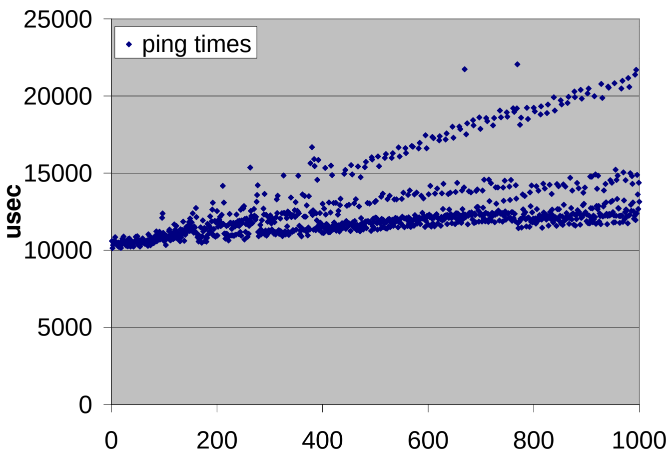

Fig. 6. Times for saneping

Figure 6 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.

5.1 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 [Jain 1991] 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. 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.

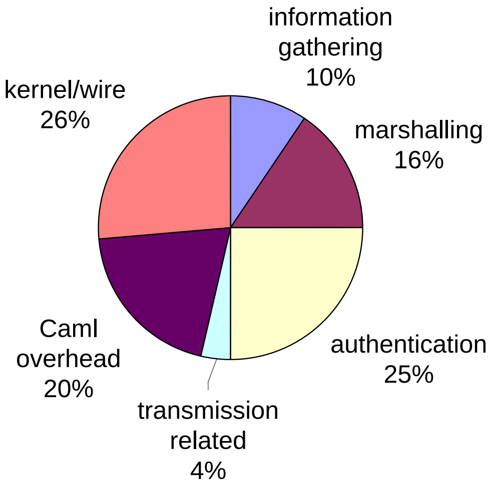

Fig. 7. Categorized costs for saneping

Figure 7[3] 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 3078μsec (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 [Brustoloni and Steenkiste 1996]. 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 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 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 avoids 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 Figure 3, the cost of authentication increases linearly with the size of the data to be authenticated / verified. (For keyed-hash or MAC[4] 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[Schwartz et al. 1998]. Other ways of speeding up the authentication include hardware assistance, cryptographic co-processors (possibly on the network card, or even entirely outside the system), and faster software-authentication algorithms[Halevi and Krawczyk 1997].

In Figure 8, we have removed the kernel / wire and authentication elements to focus on the costs imposed by ALIEN. We examine each of these in turn.

5.1.1 *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

[3]Values rounded and do not total 100%.
[4]Message Authentication Code

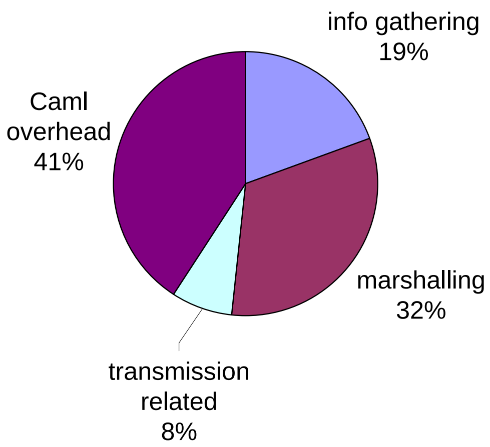

Fig. 8. Controllable costs for saneping

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 1148μsec 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 [Alexander 1998], 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 Figure 6 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 Figure 6. 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 [Appel 1987], 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.

In the case of garbage collection, incremental collection techniques based on Baker's algorithm [Baker 1978] or the techniques from [Nettles and O'Toole 1993] and [O'Toole and Nettles 1994] can be helpful. The major heap is collected using such a incremental technique. Another approach to the problem described in [Shaw 1998] 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 [Wetherall et al. 1998], 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

```
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. 9. Signature for `An_marshal`

we have outlined will be important.

5.1.2 *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.

Figure 9 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 77μsec; `ping_decode` has a median cost of 162μsec. Similarly, `An_marshal.encode` has a median time of 72μsec as compared with 189μsec for `ping_encode`.

5.1.3 *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 1094μsec.

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 39,280μsec. The median cost was 1043μsec. Further, if we calculate tendencies for this cost for the pings after the first, we find a median of 1040μsec and a mean of 1031μsec. The scaled SIQR is 2.55% and the standard deviation is 39μsec.

5.1.4 *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.

6. RELATED WORK

The Secure Active Network Environment has no direct analogues in ongoing work on active networks [Tennenhouse et al. 1997]. While ANTS uses MD5 hashes ("fingerprints") to name on-demand loaded modules, the hashes provide unique names rather than security. The ANTS execution environment depends on the Java programming language for protection, a dependency shared by many active network prototypes. Unfortunately, as [Wallach et al. 1997] notes, Java's security is suspect. The remote authentication and namespace security of SANE address issues ignored in these systems, and could be applied even in cases where Java is used, *e.g.,* to provide integrity checking of the JVM or layers beneath it, as well as on-demand loaded modules.

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.

An architecture which extended a protection model from the local domain to a distributed environment was provided by Sansom, et al. [Sansom et al. 1986], 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.

Of interest is the Proof-Carrying Code (PCC) [Necula 1997] approach, which permits arbitrary code to be executed as long as a valid proof of safety accompanies it. A number of important questions remain to be answered (*e.g.,* cost of proof verification, ability to handle abstract resource types in a dynamic environment) before we can determine exactly how and to what extend it can be used.

The Safetynet Project [Wakeman et al. 1998] at the University of Sussex has also designed a new language for active networking. They have explicitly enumerated what they feel are the important requirements for an active networking language and then set about designing a language to meet those requirements. In particular, they differ from PLAN in that they hope to use the type system to allow safe accumulation of state. They appear to be trying to avoid having any service layer at all.

7. CONCLUSIONS

This paper has made two contributions. First, it has presented the design and implementation of a secure active networking system, SANE, which provides a tight coupling between programming language means of protecting resources and cryptographic means to extend the enforcement of the protection semantics across a network. SANE's realization in SwitchWare allowed us to exploit the ALIEN active loader and its packet execution environment to provide a direct comparison of active packets against active extensions which operate over a multiplicity of packets.

Second, a detailed measurement study was performed using this infrastructure. 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. In an environment where an increasing amount of traffic is multimedia-related, such overhead may be unacceptable for data transmission. However, there are a number of promising avenues of attacking this problem (hardware assistance, faster software algorithms[Halevi and Krawczyk 1997]).

Of more interest is the effect of the overhead to the control-plane scenario of Active Networks[5]. For a secure system, we see two architectural paths to high performance. First is the use of active extensions, as they pay the cost of authentication once for a stream or packets, achieving the benefits of amortizing the

[5]By control-plane scenario we mean using Active Network functionality to control, but not actually perform, data switching.

security costs over a large number of packets. We note in passing that the caching scheme employed in the ANTS architecture is effectively a "soft-state" active extension, and thus enjoys this benefit for its MD5 hashes. The second approach is to sufficiently restrict the actions of each active packet that no authentication is required. This is the approach taken in the PLAN system. However, since many advanced PLAN services utilize active extensions, this approach may simply rely on the existence of the first architectural solution, but exploit it for further gains.

We see three promising research areas 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" or some other compression scheme) may allow a wider range of the active packet/active extension continuum to be exploited by programmers. Second, it would be extremely worthwhile to redo the experiments reported here in a second environment, such as ANTS. The major architectural points will almost surely remain true, but it would be valuable to understand some of the consequences of the language and security mechanisms on ANTS performance. Finally, these questions have to be continually reexamined as applications emerge. If most interesting applications can be written with security enforcement largely in the "control plane" then this model will dominate. If, however, many applications actually require operation in the transport plane, then new approaches following PLAN-like ideas will be needed.

ACKNOWLEDGMENTS

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.

APPENDIX

A. CODE FOR THE ACTIVE PING

```
open Safeloader
open Printf
open Wf
open Safeunix
open Log
open An_marshal

type ping_packet = { start : string ;
     finish : string;
     timestamp : float}

let ping_encode pkt = "start = " ^ pkt.start
  ^ "; finish = " ^ pkt.finish
  ^ "; timestamp = "
  ^ (string_of_float pkt.timestamp)

let decode_regexp = Str.regexp
 "^start = \(.*\); finish = \(.*\)"
```

```
 ^ "; timestamp = \(.*\)$"
let ping_decode str ofs =
  if not (Str.string_match decode_regexp
  str ofs)
  then failwith "bad packet"
  else
    { start = Str.matched_group 1 str;
      finish = Str.matched_group 2 str;
      timestamp = float_of_string
                (Str.matched_group 3 str)
    }

(*
 * This is routine that starts things off by
 * handing the components of an ANEP header
 * to send_wf.
 *)
let send_ping dest name =
  let code =
     Get_bytecode.get_bytecode name in
  let next_hop = Route.get_route dest in
  let hdr =
       {do_forward = true; type_id = 20} in
  let payload = ping_encode
 { start = An.getAddress ();
      finish = dest;
      timestamp = Time.get_time()
  } in
  send_wf next_hop hdr  [] "ping_out"
code payload

let ping_out arg_string =
  let {code=code; data=datastr; func=func} =
decode arg_string 0 in
  let ping_packet = ping_decode datastr 0 in
  if  (ping_packet.finish = An.getAddress())
  then begin
    send_wf
      (Route.get_route ping_packet.start)
      {do_forward=true; type_id=20} []
      "ping_in" code datastr;
    "at the remote machine"
  end else begin
    send_wf
      (Route.get_route ping_packet.finish)
      {do_forward=true; type_id=20} []
      "ping_out" code datastr;
```

```
    "not there yet; forward packet"
  end

let ping_in arg_string =
  let {code=code; data=datastr; func=func} =
      decode arg_string 0 in
  let ping_packet = ping_decode datastr 0 in
    if (ping_packet.start = An.getAddress())
    then begin
      log_msg
       (Printf.sprintf "Success (%f sec)\n"
  ((Time.get_time()) -.
         ping_packet.timestamp));
      "back at the sender"
    end else begin
      send_wf
        (Route.get_route ping_packet.start)
{ do_forward=true; type_id=20 } []
 "ping_in" code datastr;
      "not back yet; forward packet"
    end

let _ = Func.register "ping_out" ping_out
let _ = Func.register "ping_in" ping_in
```

REFERENCES

Alexander, D. S. 1998. *ALIEN: A Generalized Computing Model of Active Networks.* Ph. D. thesis, University of Pennsylvania.

Alexander, D. S., Arbaugh, W. A., Keromytis, A. D., and Smith, J. M. 1998. A secure active network architecture: Realization in SwitchWare. *IEEE Network 12*, 3 (May/June), 37–45. special issue on Active and Programmable Networks.

Alexander, D. S., Braden, B., Gunter, C. A., Jackson, A. W., Keromytis, A. D., Minden, G. J., and Wetherall, D. 1997. Active network encapsulation protocol (ANEP). http://www.cis.upenn.edu/~angelos/ANEP.txt.gz.

Alexander, D. S., Shaw, M., Nettles, S. M., and Smith, J. M. 1997. Active bridging. In *Proc. 1997 ACM SIGCOMM Conference* (1997).

Appel, A. W. 1987. Garbage collection can be faster than stack allocation. *Information Processing Letters 25*, 4 (June), 275–279.

Arbaugh, W. A., Farber, D. J., and Smith, J. M. 1997. A secure and reliable bootstrap architecture. In *Proceedings 1997 IEEE Symposium on Security and Privacy* (May 1997), pp. 65–71.

Arbaugh, W. A., Keromytis, A. D., Farber, D. J., and Smith, J. M. 1998. Automated Recovery in a Secure Bootstrap Process. In *To appear in Network and Distributed System Security Symposium* (March 1998). Internet Society.

Arbaugh, W. A., Keromytis, A. D., and Smith, J. M. 1998. DHCP++: Applying an efficient implementation method for fail-stop cryptographic protocols. Technical report (January), Department of Computer Science, University of Pennsylvania.

Arnold, K. and Gosling, J. 1996. *The Java Programming Language.* Java Series. Sun Microsystems. ISBN 0-201-63455-4.

Atkinson, R. 1995a. IP authentication header. RFC 1826.

ATKINSON, R. 1995b. IP encapsulating security payload. RFC 1827.

ATKINSON, R. 1995c. Security architecture for the internet protocol. RFC 1825.

BAKER, H. G. 1978. List processing in real-time on a serial computer. *Commun. ACM 21*, 4, 280–94.

BRADEN, R., ZHANG, L., BERSON, S., HERZOG, S., AND JAMIN, S. 1997. Resource ReSerVation protocol (RSVP) – version 1 functional sepcification. Internet RFC 2208.

BRUSTOLONI, J. C. AND STEENKISTE, P. 1996. Application-allocated I/O buffering with system-allocated performance. Technical Report CMU-CS-96-169 (August), Carnegie Mellon University, Pittsburgh, PA.

COMMITTEE, C. 1989. *X.509: The Directory Authentication Framework*. Geneva: International Telephone and Telegraph, International Telecommunications Union.

DAEMON9, ROUTE, AND INFINITY. 1996. Project neptune. *Phrack Magazine 7*, 48.

DIFFIE, W. AND HELLMAN, M. 1976. New directions in cryptography. *IEEE Transactions on Information Theory IT–22*, 6 (Nov), 644–654.

DIFFIE, W., VAN OORSCHOT, P., AND WIENER, M. 1992. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography 2*, 107–125.

ELLISON, C. M., FRANTZ, B., RIVEST, R., AND THOMAS, B. M. 1997. Simple public key certificate. Work in Progress.

GONG, L. AND SYVERSON, P. 1995. Fail-stop protocols: An approach to designing secure protocols. In *Proceedings of IFIP DCCA-5* (September 1995).

HALEVI, S. AND KRAWCZYK, H. 1997. MMH: Message Authentication in Software in the Gbit/second Rates. In *Proceedings of the 4th Workshop on Fast Software Encryption* (1997), pp. 172–189. Springer, LNCS vol. 1267.

HARTMAN, J., MANBER, U., PETERSON, L., AND PROEBSTING, T. 1996. Liquid software: A new paradigm for networked systems. Technical Report TR 96-11 (June), University of Arizona. http://www.cs.arizona.edu/liquid/.

HICKS, M., KAKKAR, P., MOORE, J. T., GUNTER, C. A., AND NETTLES, S. 1998. PLAN: A packet language for active networks. In *Proceedings of the Internation Conference on Function Programming (ICFP)* (September 1998).

JAIN, R. 1991. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., New York.

KARN, P. AND SIMPSON, W. A. The Photuris session key management protocol. Work in Progress.

LEROY, X. 1995. *The Caml Special Light System (Release 1.10)*. France: INRIA.

LEROY, X. AND ROUAIX, F. 1999. Security properties of typed applets. In *Secure Internet Programming*, Lecture Notes in Computer Science. New York, NY, USA: Springer-Verlag Inc.

LESLIE, I. M., MCAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R., AND HYDEN, E. 1996. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications 14*, 7 (September), 1280–1297.

MAUGHAN, D., SCHERTLER, M., SCHNEIDER, M., AND TURNER, J. 1996. Internet security association and key management protocol (ISAKMP). Internet–draft (June), IPSEC Working Group.

MILLER, S. P., NEUMAN, B. C., SCHILLER, J. I., AND SALTZER, J. H. 1987. Kerberos authentication and authorization system. Technical report (December), MIT.

MONTZ, A. B., MOSBERGER, D., O'MALLEY, S. W., PETERSON, L. L., PROEBSTING, T. A., AND HARTMAN, J. H. 1994. Scout: A communications-oriented operating system. Technical report (June), Department of Computer Science, University of Arizona.

NBS. 1977. Data encryption standard. Technical Report FIPS-46 (January), U.S. Department of Commerce.

NECULA, G. C. 1997. Proof-carrying code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)* (1997). ACM Press.

NETTLES, S. M. AND O'TOOLE, J. W. 1993. Real-time replication garbage collection. In *SIGPLAN Symposium on Programming Language Design and Implementation* (June 1993), pp. 217–226. ACM.

NIST. 1994. Digital signature standard. Technical Report FIPS-186 (May), U.S. Department of Commerce.

NIST. 1995. Secure hash standard. Technical Report FIPS-180-1 (April), U.S. Department of Commerce. Also known as: 59 Fed Reg 35317 (1994).

O'TOOLE, J. AND NETTLES, S. 1994. Concurrent replicating garbage collection. In *ACM Symposium on LISP and Functional Programming* (June 1994). ACM Press.

Panix. 1996. Cracker Attack Paralyzes PANIX. RISKS Digest. Volume 18. Issue 45.

POSTEL, J. 1981. INTERNET protocol. Internet RFC 791.

SANSOM, R. D., JULIN, D. P., AND RASHID, R. F. 1986. Extending a capability based system into a network environment. In *Proceedings of the 1986 ACM SIGCOMM Conference* (August 1986).

SCHWARTZ, B., ZHOU, W., JACKSON, A. W., STRAYER, W. T., ROCKWELL, D., AND PARTRIDGE, C. 1998. Smart Packets for active networks. http://www.net-tech.bbn.com/smtpkts/smartpkts-index.html.

SHAW, M. 1998. An architecture for an active network node. Master's thesis, University of Pennsylvania, Philadelphia.

TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W. D., WETHERALL, D. J., AND MINDEN, G. J. 1997. A survey of active network research. *IEEE Communications Magazine*, 80–86.

WAKEMAN, I., JEFFREY, A., GRAVES, R., AND OWEN, T. 1998. Designing a programming language for active networks. *submitted to Hipparch special issue of Network and ISDN Systems*. http://www.cogs.susx.ac.uk/projects/safetynet/papers/isdn.ps.gz.

WALLACH, D. S., BALFANZ, D., DEAN, D., AND FELTEN, E. W. 1997. Flexible security architecture for java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (October 1997).

WETHERALL, D. J., GUTTAG, J., AND TENNENHOUSE, D. L. 1998. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings of IEEE OpenArch 98* (April 1998). IEEE Computer Society Press.

ZIMMERMAN, P. 1995. PGP User's Manual.