Secure Quality of Service Handling (SQoSH)

D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, Steve Muir and Jonathan M. Smith*

Abstract

Proposals for programmable network infrastructures, such as Active Networks and Open Signaling, provide programmers with access to network resources and data structures. The motivation for providing these interfaces is accelerated introduction of new services, but exposure of the interfaces introduces many new security risks. The risks can be reduced or eliminated via appropriate restrictions on the exported interfaces, as we demonstrate in our Secure Active Network Environment (SANE). SANE restricts the actions that loaded modules (including "capsules") can perform by restricting the resources that can be named; this model is extended to remote invocation by means of cryptographic certificates.

We have extended SANE to support restricted control of Quality of Service in a programmable network element. The Piglet lightweight device kernel provides a "virtual clock"-like scheduling discipline for network traffic, and exports several tuning knobs with which the clock can be adjusted. The ALIEN active loader provides safe access to these knobs to modules which operate on the network element. Thus, the SQoSH architecture is able to provide safe, secure access to network resources, while allowing these resources to be managed by end users needing customized networking services. A desirable consequence of SQoSH's integration of access control and resource control is that a large class of denial of service attacks, unaddressed solely with access control and cryptographic protocols, can now be prevented.

We provide some performance measurements to illustrate the cost of security, and demonstrate that these costs are minor in the context of managing a multimedia stream.

1 Introduction

In this section, we discuss the notion of security, provide a constructive definition useful for asserting general security properties, and discuss the mapping of this definition onto computer networks, and outline the challenges of securing programmable network infrastructures, which the remainder of the paper addresses.

*This work was supported by DARPA under Contract #N66001-96-C-852, with additional support from the Intel Corporation.

1.1 Security

It is attractive to think of security as a clearcut property of a system, as in the definition of "odd" and "even" numbers. Unfortunately, security is not so easily abstracted, as it is fundamentally a context-dependent property of an engineered system. For example, some applications can tolerate near-infinite delays while others have strict real-time requirements. Thus a system design which guarantees reliable and eventual action may be considered secure in the first case but inadequate (*e.g.*, against "denial-of-service" attacks) in the latter.

In general, a secure system is one which meets or exceeds an application-specified set of security policy *requirements*. So, for example, in message delivery, the high-level requirements may be that the correct information gets to the right person, in the right place, at the right time. The details of "right" are determined by the application's needs; for example timely message delivery is crucial for battlefield or stock-trading tasks.

1.2 Active and Programmable Networks

Active Networks is a proposal for packet-switched networks which are programmable, perhaps on a per-user or even a per-packet basis. The more aggressive proposals [TSS+97] share the property that "programs" are loaded into network elements on-the-fly, providing rapid dynamic reconfiguration of the network infrastructure. Open signaling systems, such as DCAN [vdML97] or XBind [LBL95], restrict the programmability to the *control plane*.

Operational active networking infrastructures have been produced by several initial efforts, such as Active Bridging [ASNS97], ANTS [WGT98] and PLAN [HKM+98]. These efforts are points in a design space which has many dimensions, the most important of which are flexibility, security, usability and performance. Programmable network elements provide flexibility and usability via the choice of programming language and execution environment. For example, the portability and distributed programming support of the Java programming language have made it a popular basis for active networking prototypes[HPB+97].

1.3 Security for Active Networks

Security for active networking is a major challenge, as well as a widespread and legitimate cause for concern. One view of Information security can be characterized as getting the right information to the right person at the right place and time. This is the positive statement of a security policy; other security policies might assert what *cannot* occur. The flexibility of an active networking infrastructure, since it might be exploited for mischief, has the effect of

hugely expanding the threat model for attacks on the network infrastructure. For example, "denial-of-service" attacks can now be made against a variety of resources, such as CPU cycles, output link bandwidth and storage, since these are exposed either wholly or in part to loaded programs.

Typical reasons for deferring consideration of security, aside from simple difficulty, are the negative consequences making a system more secure has for each of flexibility, usability and performance. Since the programming language based approaches to active networking offer advantages in terms of flexibility and usability, and performance optimizations for these environments are ongoing, providing security to such an environment would offer an attractive design point among the various tradeoffs.

1.4 Threat Model for QoS Provision in an Active Network

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 [Pos81] 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 [Atk95c] 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 (QoS) guarantees [BZB+97].

In the context of QoS, a secure system is one which is secure against two types of threats, which we will denote *admission* failures and *policing* failures. Providing QoS is basically about controlled unfairness. Control of the unfairness is accomplished via QoS specifications, which are ultimately realized as queuing disciplines. The queuing disciplines use packet discrimination to apply the policy resulting from a QoS specification. A concrete example of an applied queuing discipline (*e.g.*, Weighted Fair Queuing or Virtual Clock) is allocating bandwidth to flows in an IP Internet, or allocating a more complex resource profile (memory, CPU, input/output port bandwidths) in an active network.

When an "Active 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. A second view is that of static versus dynamic checking. Security violations occur when a policy is violated, *e.g.*, reading a private packet, or exceeding some specified resource usage.

An *admission* violation is one where an unauthorized reallocation of bandwidth/resources occurs. For example, an RSVP-capable router[BZB+97] might be asked to reallocate bandwidth away from one flow to one which has not paid/ has no right to the bandwidth. The policing mechanism is working correctly; it is applying a QoS specification admitted by the network infrastructure. Unfortunately, the specification was unauthorized. This is as much a threat in RSVP or other resource reservation systems as it is in an active network; the greater concern in the active network is a direct consequence of the more complex resource model.

A *policing* violation is one where the specified QoS is correct but the system fails to deliver what is requested. For example, a network element incorporating a computer might be subject to denial of service attacks based on "receive livelock". A second example is aggressive use of bandwidth on a shared output port, which denies bandwidth to a process with QoS requirements. This is a threat to basic IP routers with FIFO queuing disciplines.

The admission/policing division may also be viewed as a control plane versus transport plane partitioning.

1.5 SQoSH Applications

SQoSH provides a powerful new tool for managing resources in a network. It controls access to managed resources, and integrates this control with the resource management mechanisms provided by the Piglet operating system. While the SANE/Piglet combination represents the first instance of the SQoSH architecture, we believe that compelling applications will motivate deployment of SQoSH and SQoSH-inspired architectures. Examples include:

1. Economic algorithms for robust adaptive control. SQoSH is well adapted to this environment for two reasons. First, it is critical that resources sold match the resources delivered if the marketplace is to work. Second, the recovery strategies for flows that are outbid in an auction may be quite complex (*e.g.,* aggregating several flows each of which delivers a portion of the request, searching for a different route, delaying until the resources required become available at the desired prices, or combinations of different strategies). We expect that capturing these complex decisions can be most easily done by active packets (called "switchlets") in a programmable network infrastructure.

2. Trustworthy remote administration and management. A certificate hierarchy produced by SANE can be used in conjunction with PolicyMaker to configure and operate on large distributed pools of network elements, whether Active or not.

3. Military applications, where hierarchical command responsibility maps to multiple classes of service and security. SQoSH ensures that any control requests are authenticated, ensures that autonomous network elements bootstrap into a secure state, and finally that once admission requests are validated that service will be delivered. For example, a command channel of 2% of bandwidth could be preserved at all times. For commercial applications this might be considered wasteful, while military uses might dictate provision of such a no-delay override facility.

1.6 Paper Overview

The rest of the paper begins with descriptions of the Penn/Bellcore *SwitchWare* [SFG+96] project and the Secure Active Network Environment (SANE [AAKS98]). The SANE infrastructure provides security guarantees to the network elements and overlaid services. Additional security services can be built on top of SANE, using the existing primitives. These primitives include secure bootstrapping using the AEGIS architecture [AKFS98]; key exchange; authentication and identification of network entities; packet confidentiality; integrity and resource and access control; and namespace protection.

Section 2 describes the SQoSH architecture and principles. Section 5 presents the current status of the implementation with some experiments and performance results. Section 6 describes the Piglet operating system design, and Section 7 demonstrates the system's

ability to resist denial of service attacks using network bandwidth as an example. Section 8 briefly reviews related projects, and delineates where the SQoSH advances lie. Finally, Section 9 discusses future extensions and directions.

2 The SQoSH Architecture

As we discussed in the Introduction, the goal of Secure Quality of Service Handling is to protect against two types of threats to QoS provision, *admission* and *policing*. Balancing performance and security considerations suggests that we make common operations (*e.g.*, those used to classify packets) cheap, and make less common operations more expensive if this contributes to reducing the cost of common operations. An example of this approach is to provide heavyweight authentication mechanisms at the level of aggregates of packets such as a channel or flow, so that these checks not be done on small groups of packets (*e.g.*, individual packets).

This suggests an architecture where authentication and other resource management decisions are "front-loaded" to reduce the cost of subsequent decisions. We view this scheme as one where expensive static checks are traded for cheaper dynamic checks. Thus, the SQoSH architecture echos similar design decisions made in restricting programmability to the control plane [vdML97, LBL95] and similar, although not equivalent decisons made in the overall *SwitchWare* architecture [AAH+98] and its components such as SANE [AAKS98] (See Section 5, below).

This division of functions into admission/authentication and policing/provision is the approach we have chosen for SQoSH. Figure 1 illustrates the SQoSH architecture at a high level. The SANE system is the only means of access to resource management interfaces provided by Piglet (see Section 6). Heavyweight cryptographic operations required for granting access to Piglet resources are performed by SANE as a front-end. Piglet is thus assured that any resource requests have been authenticated, and thus need focus on whether the resources can be allocated to the validated request. Packets destined for SANE are demultiplexed by Piglet which provides basic packet delivery operations for SANE.

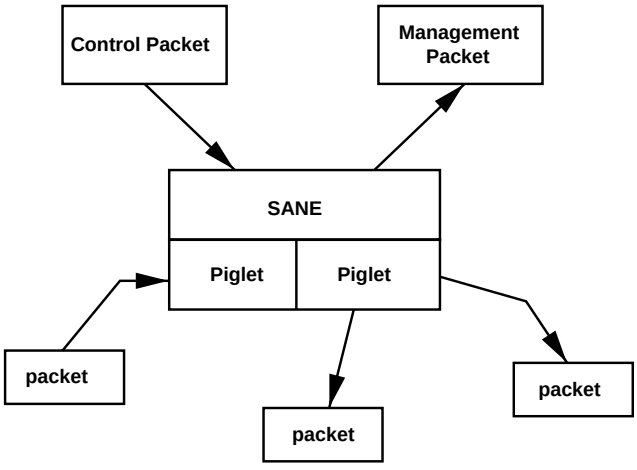

Figure 1: **SQoSH Architecture**

Since Piglet is intended as an asymmetric multiprocessor operating system, multiple instances of Piglet can manage multiple network line cards. A full-scale SQoSH system would consist of a multiprocessor, with a Piglet instance on each device-managing processor (a resizeable subset of all the processors in the system). In this way, all device I/O can be managed by Piglet. Since a typical model for scheduling resources is activity triggered by a device interrupt, Piglet can manage interrupts, buffering, status polling, etc. and protect the host operating system from device-initiated actions.

In subsequent sections of the paper, we describe SANE and Piglet and provide experimental results which illustrate the performance implications of our architectural choices.

3 Security and Safety in SwitchWare

While the SQoSH architecture is portable across many active networking environments, our experimental prototype is constructed in the context of the *SwitchWare* architecture. *SwitchWare* is based on the approach of using restricted semantics to contain the behavior of potentially mischievous programs. This has the benefit that enforcing restrictions can be performed once at compile or link time, resulting in a lower cost than an OS approach such as memory protection which requires repeated checks at runtime. 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.

3.1 The Loader

The loader forms the basis of the dynamic security for our network infrastructure. Once it has been securely started by the AEGIS bootstrap (see section 4.1), the loader provides a minimal set of services necessary to find the Core Switchlet and start it running. It also provides policy and mechanism for making changes to the Core Switchlet, if that is desirable.

The loader is responsible for providing the mechanism by which modules are loaded. Currently, the mechanisms provided are loading from disk or loading from the network. The Core Switchlet governs the policy by which this mechanism may be used and may provide interfaces to the mechanism.

3.2 The Core Switchlet

The Core Switchlet is the privileged portion of the system visible to the user. Through the use of module thinning, it determines which functions and values are visible to which users. The services that it provides are broken into several modules.

The module `Safestd` provides the functions that one would expect to find in any programming language including addition and multiplication as well as more complex abstractions like lists, arrays, and queues. Many functions including the I/O functions have been thinned from this module to make it safe.

The next module is `Safeunix`. This module has been very heavily thinned; it gives access to Unix error information, some time related functions, and some types that are needed for the networking interface that we provide. Access to the rest of the Unix functions has been thinned away.

In order to allow the user to supply error or status messages, we also have a `Log` module.

Access to the network is provided by the `Unixnet` and the `Safeudp` modules. The former provides access to raw Ethernet frames while the latter provides access to the native OS (*e.g.*, Linux) implementation of UDP [Pos80]. This allows switchlets to access network interfaces for either sending or receiving frames or packets. Access to the data packets will be available to any switchlet, assuming said switchlet can prove that it has the authority to access the data. For the work described in this paper, we used the `Safeudp` interface.

Thread support in SwitchWare is provided by a set of three modules: `Safethread`, `Mutex`, and `Condition`. These provide a threads package which helps in the structuring of the system. Each switchlet runs in a thread and is capable of creating

additional threads. When a switchlet is first started, it is given an identifier inside of an opaque type. (An opaque type is one which has no conversion functions to or from any other type. Thus, the identifier cannot be forged.) In order to use additional resources including creating additional threads, the switchlet must provide its identifier which allows the system to check the resources currently consumed and allow or deny the request for additional usage.

Finally, we have a set of modules to support loading of switchlets. The `Aegis` module allows access to the AEGIS public keys. The `An_marshal` module gives interfaces to allow quick transformations between strings and the standard format in which we access our active packets. The `Func` module allows files or strings to be loaded and executed based on the system access policy. Finally, `Route` is a very simplistic static routing scheme which allows us to impose an arbitrary active network topology on top of our physical network without the need to crawl under desks to move cables.

3.3 The Library

The library is a set of functions which provide useful routines which do not require privilege to run. The proper set of functions for the library is a continuing area of research. Some of the things that are in the library for the experiments that we have performed include utility functions for sending and receiving active packets.

4 Secure Active Network Environment – SANE

The following subsections present the components of SANE and explain how they fit together. Figure 2 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, *e.g.* Caml runtime have been verified, the static integrity guarantees of the system have been assured and we transition to our dynamic integrity mechanisms.

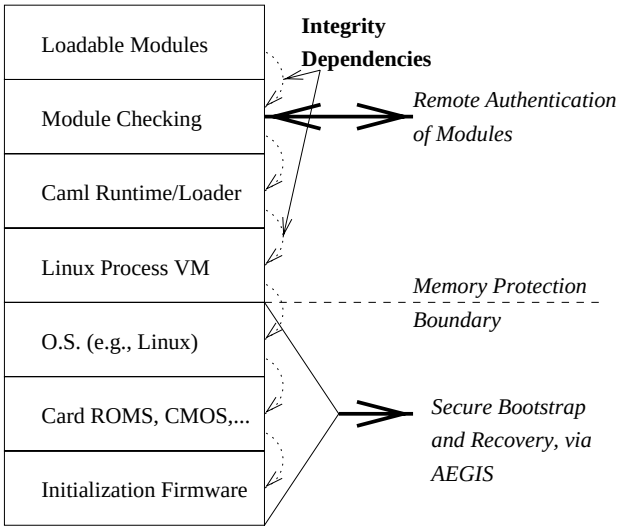

Figure 2: **SANE Architecture**

4.1 AEGIS Architecture

AEGIS modifies the standard IBM PC process so that all executable code, except for a very small section of trusted code, is verified prior to execution by using a digital signature. This is accomplished through modifications and additions to the BIOS (Basic Input/Output System). In essence, this trusted software serves as the root of an authentication chain that extends to the evaluator and potentially beyond to "active" packets. In the AEGIS boot process, either the Active Network element is started, or a recovery process is entered to repair any integrity failure detected. Once the repair is completed, the system is restarted to ensure that the system boots. This entire process occurs without user intervention. AEGIS can also be used to maintain the hardware and software configuration of a machine.

It should be noted that AEGIS does not verify the correctness of a software component. Such a component could contain a flaw or some trapdoor that can be exploited. The goal of AEGIS is to prevent tampering of components that are considered trusted by the system administrator. The nature of this trust is outside the scope of AEGIS.

Other work on the subject of secure system bootstrapping includes [TY91, Yee94, Cla94, LAB92]. A more extensive review of AEGIS and its differences with the above systems can be found in [AFS97].

4.2 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. The services initially provided are:

- public key signatures (DSA [NIS94])
- symmetric key encryption (DES [NBS77])
- (keyed) hashes (SHA1 [NIS95])

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 SHA1. Access to this implementation of SHA1 occurs through a Caml interface, taking care to avoid potential bypassing of the type system. Hardware cryptographic support is being considered.

4.3 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 intend to use a combination of SPKI [EFRT97] and PolicyMaker [BFL96] or KeyNote [BFK98]. For more details on the certificate format, see Section 5.

4.4 Key Establishment Protocol (KEP)

The protocol we use throughout this paper and in our architecture is based on the Station to Station protocol [DvOW92]. The basis of the protocol is the Diffie-Hellman exchange [DH76] for key establishment, and public key signatures for authentication (to avoid man-in-the-middle attacks). In our architecture we use DSA (a NIST-approved digital signature algorithm), but other (*e.g.*, RSA [Lab93] etc.) algorithms can be used.

Briefly, this protocol allows each participant to establish the identity of the other, discover the operations that the peer is authorized to perform, and allows the two parties to establish a shared

secret to be used for a variety of purposes including the authentication and encryption of future traffic. This is accomplished by having each party send the other both an authentication certificate and an authorization certificate and using Diffie-Hellman key exchange to establish the shared secret. The protocol is carried out with a total of three messages transmitted. For more details on the protocol, see [AKFS98].

A node that has detected an integrity failure can establish this secure channel with a repository. It can then request a new version of the failed component. The repository will send the new component protected by the shared key to prevent tampering from an attacker. The component can additionally be signed by some trusted authority using a digital signature algorithm, to prove its validity (*e.g.*, a signature by company X).

4.5 Packet Authentication

Once a key has been established between two nodes, they can commence exchanging authenticated and / or encrypted packets. In SANE, we use the ANEP [ABG+97] packet format over UDP, although in a homogenous active network a packet format would be unnecessary. We've added an authentication header, as shown in Figure 3, similar to the one used in the IPsec Authentication Header protocol [Atk95a]. The SPI is negotiated during the key establishment protocol exchange, and is used to identify the security association and corresponding cryptographic material used. The $Replay\ Counter$ is a monotonically increasing value, used to prevent packet replay attacks. The $authenticator$ is the keyed hash (HMAC [KBC97]) computed over the SPI, $replaycounter$ and packet payload. We can similarly define an encryption header similar to the IPsec ESP [Atk95b] protocol.

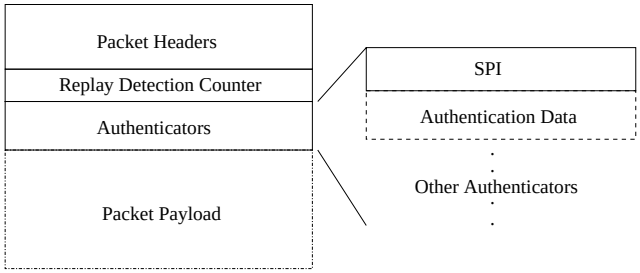

Figure 3: **Authenticator Header**

4.6 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 to make trust decisions on their behalf.

4.7 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).

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 [MNSS87].
- 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 [KBIS98].

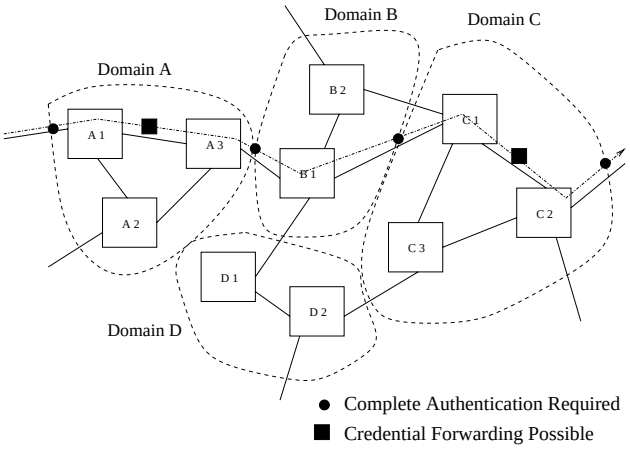

Figure 4: **Administrative Clouds and Path Setup**

4.8 Resource Control

Resource control on the active switch is imposed by the runtime system, as specified by the certificates exchanged during key establishment. In SQoSH, these are used to control the behavior of Piglet. The protected resources will include access to standard and loaded modules, CPU cycles, memory allocated, number of packets, latency and bandwidth requirements, and others. For the Piglet experiments we report below in Section 7 we have limited the resource management to network bandwidth. There is a great deal of further research necessary to determine what the right resources to manage are, and how to resolve conflicting resource requests.

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. For system resources (such as memory or bandwidth), the certificates can also specify the allowed usage patterns (*e.g.*, "no more than 4 Mbit/sec"). As a result, this scheme allows fine-grained control of switch resources.

4.9 Dynamic Resource Naming

Conceptually, loaded modules can be considered as the interfaces to user defined resources. Other resources potentially include memory, CPU cycles, bandwidth, disk space, real-time guarantees etc. 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 "naive" way of naming (using some user-defined value) would not work well, because names need to be unique across the Active Network. If users arbitrarily assign names to their resources, it is conceivable that there will be accidental naming collisions; worse yet, forging names is possible, allowing for resource-in-the-middle attacks. Alternatively, some centralized authority could assign names per request, making sure these remain unique; this solution is unattractive because it does not scale well as the number of names required increases.

SANE provides a decentralized mechanism for resource naming that does not allow name collisions (accidental or malicious). It does this by generating the name of a dynamic resource from the cryptographic credentials used to authenticate/authorize it. This prevents "trojan horse" type of attacks. See [AAKS98] for more details.

5 Implementation and Performance of SANE

We have implemented SANE in the *SwitchWare* environment. For our experimental network we used a cluster of DEC Alpha PC 164SX 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 some of our throughput tests, we modified the Linux kernel to allow allocation of a buffer larger than 64KB per socket.

5.1 SANE Performance Measurements

Tables 1, 2, and 3 show the costs of the three cryptographic primitives provided by SANE. 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. Additionally, it shows the difference in cost between compiled and interpreted code. 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	bytecode	36.027246 s
	native	2.477051 s
C		0.333212 s

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

Caml	bytecode	99.331543 s
	native	16.723242 s
C		1.0785348 s

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

sign	Caml	bytecode	20.907 ms
		native	11.855 ms
sign	C		2.800 ms
verify	Caml	bytecode	35.198 ms
		native	20.664 ms
verify	C		5.000 ms

Table 3: **Digital Signature Timings**

In practice, to use the dynamic loader in Caml, we must use the bytecode interpreter. This imposes a very high overhead on authenticating packets, an operation which relies on the SHA-1 hash function, so we have resorted to a C implementation. While this greatly speeds the HMAC generation and verification operations, it may interfere with the Caml runtime thread scheduler. 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 tried to limit the amount of non-Caml code in our system. 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. Compilation techniques such as "Just In Time" should help narrow this gap in performance.

The key exchange protocol was also implemented in Caml. The protocol was designed to be fail safe [GS95] 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). 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 [KS] and ISAKMP/Oakley [MSST96].

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) or a hierarchical approach (such as in X.509 [Com89]) 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.

5.2 Cost of Active Ping

To understand the cost imposed by authentication, we measured the cost of sending an active ping both with and without authentication. 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. An unauthenticated ping took an average of 5.084 ms versus 8.052 ms for the authenticated ping.

As described in [AAKS98], we believe that these numbers can be improved by changes to the Caml runtime system. In particular, caching of switchlets, improvements to the thread scheduler, and improvements to the thread linker could help with this performance. Fortunately, in the SQoSH environment, these costs are likely to be incurred only during connection setup and so are less critical than they are generally in SANE. Since we are modifying the queuing strategy and can reasonably expect a given strategy to run for minutes at least, we have the opportunity to amortize away the milliseconds of overhead necessary for authentication.

6 Piglet

While the SANE architecture presents a secure resource-management interface to switchlets, the underlying system must be capable of providing the appropriate capabilities to implement that interface. In the SQoSH architecture those capabilities are provided by Piglet [MS98], a functionally-partitioned network operating system derived from Linux.

6.1 Piglet: Structure and Architecture

Piglet is designed around an asymmetric, functionally-partitioned architecture, where system processors are dedicated to particular functions rather than having each processor execute user applications, as is typically the case in a symmetric multiprocessor OS. While the symmetric model is attractive in computationally-intensive applications, we believe that the functional model is more appealing in systems where a high proportion of the workload consists of I/O, as in a network element.

Piglet is implemented as a *lightweight device kernel (LDK)* which runs on one or more processors of a multiprocessor system. In a network element, this LDK is responsible for managing one or more of the network interfaces of the system. Those interfaces are then made accessible to the host operating system, in this case Linux, via *virtual device interfaces (VDIs)*. These VDIs can extend the capabilities of the physical network interfaces with additional services provided by the LDK, including various resource management functions, such that Piglet can present QoS guarantees to applications.

6.2 Resource Management in Piglet

Piglet's resource management functions are based around an abstract data structure known as a *frameset*. The implementation of the frameset data structure is not relevant or important to the discussion of SQoSH—for our purposes it suffices to say that a frameset consists of independent transmit and receive queues into which *frames*, each corresponding to a network packet, can be placed. Services can be associated with framesets in order to manipulate and process frames—since services form an integral part of the Piglet OS they have full access to the host system and can thus perform almost any conceivable function.

For the purposes of network element applications each frameset is logically associated with a network *flow*. The exact details of a flow specification are bounded only by the packet-filtering

```
frameset_t *
piglet_create_frameset(int tx_bufs,
                       int rx_bufs,
                       unsigned options);

int
piglet_set_filter(frameset_t *flow,
                  filter_t *filter);

int
piglet_set_vclock(frameset_t *flow,
                  unsigned period,
                  unsigned limit);
```

Figure 5: Flow management functions in Piglet

mechanism employed by Piglet to classify received network packets into flows (and hence demultiplex to the appropriate frameset). A flow can be defined as broadly as "All packets received on this network interface", or as specifically as "All packets sent by host 158.130.6.140 to TCP port 5005 on host 158.130.4.4".

Figure 5 shows prototypes for the principal functions used to manage framesets. `piglet_create_frameset` is used by an application to create a frameset, specifying the transmit and receive buffer sizes and an option field indicating, among other things, which services should be used to process this frameset. Once the frameset has been created, `piglet_set_filter` is called to associate a flow specification with the frameset.

An example of a service provided by Piglet is the Virtual Clock algorithm, a mechanism for scheduling a network link across multiple flows and controlling the rate at which each flow sends data. Virtual Clock is parameterised by the clock period and maximum amount of data to be sent in that period—tuning these two parameters allows an application to specify not only its bandwidth requirement but also the degree of burstiness of its traffic. The function `piglet_set_vclock` is used to convey these parameters to Piglet.

7 SQoSH Resource Multiplexing using Piglet

The specific problem we address is that of multiplexing a single network interface between a number of uncooperative applications, each of which may or may not have specific requirements. This is exactly the challenge faced in secure multiplexing of resources once a policy has been validated by checking its certificates. In this example, the resource we wish to divide among these applications is network bandwidth.

7.1 Experimental Description

The experimental setup for this test consists of two PCs connected to an AsanteFast 100Mb/s Ethernet hub by 3Com 3c905 network interface cards. The sender is a 200MHz dual-processor Pentium Pro PC, running RedHat Linux 5.0 with the Piglet kernel replacing the standard Linux kernel. The receiver is a 200MHz uniprocessor Pentium Pro PC, running RedHat Linux 4.2 with the Linux kernel 2.0.31. Both machines are idle apart from the test applications, and the test network has no other traffic.

The experiment consists of three applications all trying to send a large amount of data from the sender to the receiver. The results are plotted in Figure 6. Each application has different bandwidth requirements:

1. **A** - an unconstrained sender which uses as much bandwidth as is available. This can be viewed as the source of a "denial-of-service" attack in the context of SQoSH.

2. **B** - a sender constrained to run at 40Mb/s [1].

3. **C** - a sender constrained to run at 10Mb/s.

The application used to send the data is the standard *ttcp* augmented with an option to set the Virtual Clock parameters (period and time). These parameters are passed to the Linux TCP/IP stack by `setsockopt()` system calls, where they are then passed to Piglet to create application-specific framesets with those parameters. This is the only modification made to the Linux networking code.

Each of applications **A**, **B**, and **C** start and stop sending their data at different times, and the per-application bandwidth is measured every second and plotted in Figure 2 as the three heavy lines. The three thinner lines show reference bandwidth measurements for each sender with no competing applications over a 30 second period.

- After 1s, **B** starts sending at 40Mb/s[2].
- After 5s, **A** starts sending as fast as possible. Piglet's guarantee of 40Mb/s to **B** limits **A** to approximately 40Mb/s also.
- After 9s, **C** starts sending at 10Mb/s, causing **A**'s bandwidth to decrease by approximately 10Mb/s.
- After ≈15s, **B** stops sending, **A**'s bandwidth thus increases to approximately 10Mb/s below the absolute limit.
- After ≈20s, **A** stops sending.
- After ≈29s, **C** stops sending.

7.2 Experimental Results

We see from the graph that Piglet's queue scheduling mechanism provides controlled multiplexing of the shared network resource, despite the fact that the applications are not cooperating to share the resource, and neither they nor the host O.S. (Linux) are aware of the constraints imposed upon them.

The applications which have specific bandwidth requirements receive exactly that amount of bandwidth, even when an application with no specified constraint is competing for the same resource. This ability to add resource management to a standard host O.S. is one of the key strengths of Piglet, and enables its easy integration into SQoSH.

8 Related Work

8.1 Quality of Service Provision and Management

QoS provision and management has a wide-ranging literature. Much of the early work was stimulated by the promise of Asynchronous Transfer Mode (ATM) networks[dP91]. The demand for these services was stimulated by multimedia traffic[PS95]. The relevant promise was the control of multiplexing behavior in both endpoints and network elements, with the idea that ATM hardware-supported queuing disciplines such as Fair Queuing or its many variants could be used to allocate bandwidth resources, and for the most part provide delay bounds. While such hardware support remains attractive, the signaling software (Q.2931) has proved sufficiently unwieldy that the potential for managed bandwidth remains largely unrealized.

The attraction of integrated services did serve, however, to revitalize and stimulate research into integrated services in the IP Internet community[Sch96]. This research program resulted in the RSVP[BZB+97] proposal for signaling resource reservations to network elements by endpoints.

Neither ATM signaling protocols (*e.g.*, Q.2931 or UNI 3.1) [dP91] nor RSVP[BZB+97] provide the integrated admission control and policing of SQoSH. It is presumed that administrative entities are trusted in either system, while policing is delegated; to hardware in the ATM setting and to some lower layer through the Internet Subnet-Specific Layer (ISSLL) in the RSVP case. Some extensions for securing signaling are discussed by Schuba [SLS97]. An additional limitation of these systems (although we believe them extensible) is that their policing is limited to bandwidth management, rather than the more general resource model inherent in an Active Network.

8.2 Secure Resource Control in Active and Programmable Networks

The Secure Active Network Environment has no direct analogues in ongoing work on active networks [TSS+97]. 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., [WBDF97] note, 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 all enhanced services are added to the node as PLAN extensions, any such extensions 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.* [SJR86], where protection was enforced 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.

8.3 SQoSH and other environments

The Cambridge University Nemesis [BBDS97] operating system has considerable potential for supporting SQoSH functionality, as its single-layer multiplexing model can be readily adapted to the SQoSH policing requirements. Using the SANE architecture, Nemesis could be enhanced with automated scheduling domain setup and

[1] Here and in our experimental results we use the convention that 1Mb/s = 10^6 bits per second

[2] ttcp actually tries to send as fast as possible but Piglet constrains packet transmission to 40Mb/s

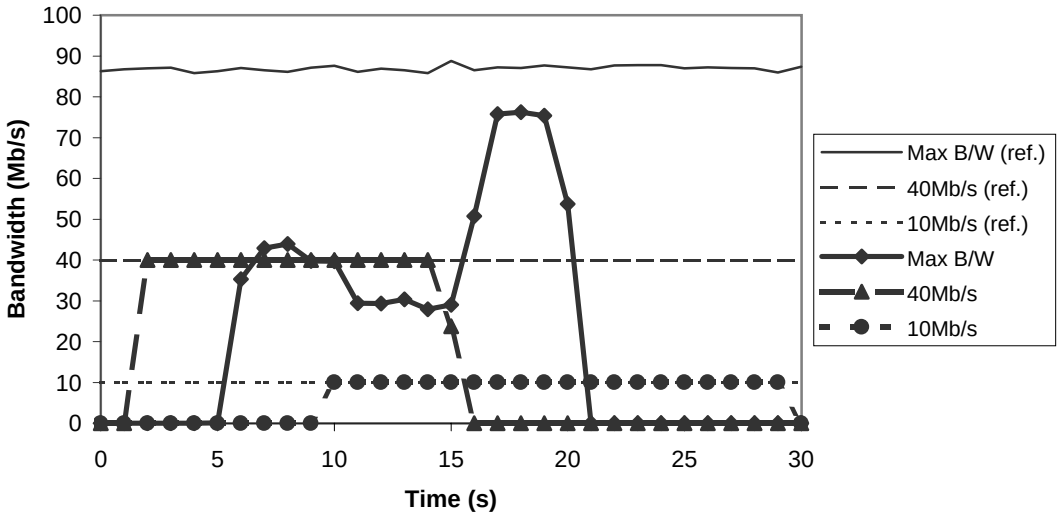

Figure 6: Per-channel bandwidth as a function of time

adjustment to service Active Network needs.

Another system with great potential is the Arizona Scout/Escort [MMO+94] operating system with its support end-to-end resource allocations called "paths". Paths, in spirit, are the right idea for end-to-end allocation in an active network. The security infrastructure is not nearly as complete as SANE, with its secure initialization, public-key infrastructure, ALIEN active loader and remote module authorization certificates. We believe that, like Nemesis, Scout could be readily adapted to support SQoSH.

8.4 Limitations of this work

The SQoSH architecture we described provides a generalized solution to trustworthy control and management of resources in an integrated services network.

The Piglet and SANE software systems are implemented but further integration must take place before the extension to Active Networks is completely validated. In particular, our Piglet experiments were limited to bandwidth provisioning for a single network adapter. Ideally, we would be able to provision a variety of resources, such as CPU cycles or regions of main memory, and demonstrate how the SQoSH system copes with co-scheduling these resources. This will be necessary in future active networks; consider for example a Java program which has a very small CPU and bandwidth allocation, but allocates new memory at a high rate, inducing garbage collection overheads which deny CPU, buffering and bandwidth to other activities.

Referring to our introduction, we made it clear that "security" is very application-dependent. Thus, SQoSH is not really "secure"; rather, it provides the tools and infrastructure necessary to build systems that are secure. Like any other infrastructure it remains vulnerable to misconfiguration and administrative mischief, such as inappropriate grants of resource access certificates.

9 Conclusions and Future Work

The Secure Quality of Service Handling (SQoSH) architecture provides controlled access to allocations of system resources in an Active Network element. It is more generally applicable to any resource allocation or policing scheme where remote allocation and deallocation of resources is required. We believe, for example, that this new architecture is well suited to providing secured resource allocation in an integrated services internetwork.

An example of the general architecture can be constructed using the Secure Active Network Environment (SANE) as a protective mechanism for resource allocations available from the Piglet operating system. As an example of resource policing, we have demonstrated Piglet partitioning bandwidth using a Virtual Clock-like queue-scheduling discipline. The measurements showed the effectiveness of Piglet on this task.

Since SANE is used to control access to Piglet operations, we measured the cost of the cryptography for commonly performed tasks. The results indicated that, not surprisingly, cryptographic transformations incur a major performance cost on SANE functions. Since SANE operations are in the SQoSH "control plane" they are perfomed infrequently relative to the policing functions, and thus their cost has a minor effect on overall performance. An additional benefit of SANE's use of a public-key infrastructure is the presence of this infrastructure for preserving privacy and integrity of media streams if required.

We believe that SQoSH represents a practical advance in automating and securing the administration of remote network elements of any type. We presented the threat model which SQoSH addresses, and showed that these attacks (admission failures and policing failures) can be thwarted. Typical architectures and implementations provide no protection against such attacks, except perhaps via inflexibility. In any environment where resources and resource allocations have value, SQoSH ensures that the resources are allocated as intended.

A major concern of ours is the translation from security policy to admission and policing actions. The approach we are pursuing in SQoSH is to integrate PolicyMaker in the runtime system, while continuing to use Piglet as the means of low-level policy enforcement.

10 Acknowledgements

We'd 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.

References

[AAH+98] D. S. Alexander, W. A. Arbaugh, M. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M.

Nettles, and J. M. Smith. The switchware active network architecture. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 1998.

[AAKS98] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. A Secure Active Network Environment Architecture. *IEEE Network*, August 1998.

[ABG+97] D. Scott Alexander, Bob Braden, Carl A. Gunter, Alden W. Jackson, Angelos D. Keromytis, Gary J. Minden, and David Wetherall. Active network encapsulation protocol (anep). http://www.cis.upenn.edu/~angelos/ANEP.txt.gz, August 1997.

[AFS97] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[AKFS98] William A. Arbaugh, Angelos D. Keromytis, David J. Farber, and Jonathan M. Smith. Automated Recovery in a Secure Bootstrap Process. In *Network and Distributed System Security Symposium*. Internet Society, March 1998.

[ASNS97] D. S. Alexander, M. Shaw, S. M. Nettles, and J. M Smith. Active bridging. In *Proc. 1997 ACM SIGCOMM Conference*, 1997.

[Atk95a] R. Atkinson. IP authentication header. RFC 1826, August 1995.

[Atk95b] R. Atkinson. IP encapsulating security payload. RFC 1827, August 1995.

[Atk95c] R. Atkinson. Security architecture for the internet protocol. RFC 1825, August 1995.

[BBDS97] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol implementation in a vertically structured operating system. In *Proc. 22nd Annual Conference on Local Computer Networks*, 1997.

[BFK98] M. Blaze, J. Feigenbaum, and A. D. Keromytis. The keynote trust-management system, April 1998.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[BZB+97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation protocol (RSVP) – version 1 functional sepcification. Internet RFC 2208, 1997.

[Cla94] Paul Christopher Clark. *BITS: A Smartcard Protected Operating System*. PhD thesis, George Washington University, 1994.

[Com89] Consultation Committee. *X.509: The Directory Authentication Framework*. International Telephone and Telegraph, International Telecommunications Union, Geneva, 1989.

[DH76] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, Nov 1976.

[dP91] Martin de Prycker. *Asynchronous Transfer Mode*. Ellis Horwood, 1991.

[DvOW92] W. Diffie, P.C. van Oorschot, and M.J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[EFRT97] Carl M. Ellison, Bill Frantz, Ron Rivest, and Brian M. Thomas. Simple Public Key Certificate. Work in Progress, April 1997.

[GS95] Li Gong and Paul Syverson. Fail-Stop Protocols: An Approach to Designing Secure Protocols. In *Proceedings of IFIP DCCA-5*, September 1995.

[HKM+98] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Plan: A programming language for active networks. Technical report, Department of Computer and Information Science, University of Pennsylvania, February 1998.

[HPB+97] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck. Joust: A platform for communications-oriented liquid software. Technical report, Department of Computer Science, University of Arizona, November 1997.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC:Keyed–Hashing for Message Authentication. Internet RFC 2104, February 1997.

[KBIS98] Angelos D. Keromytis, Matt Blaze, John Ioannidis, and Jonathan M. Smith. Firewalls in active networks. Technical report, University of Pennsylvania, February 1998.

[KS] P. Karn and W. A. Simpson. The Photuris Session Key Management Protocol. Work in Progress.

[LAB92] Butler Lampson, Martin Abadi, and Michael Burrows. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, v10:265–310, November 1992.

[Lab93] RSA Laboratories. *PKCS #1: RSA Encryption Standard*, version 1.5 edition, 1993. November.

[LBL95] A. A. Lazar, S. Bhonsle, and K.-S. Lim. A binding architecture for multimedia networks. *Journal of Parallel and Dsitributed Computing*, 30:204–216, 1995.

[MMO+94] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical report, Department of Computer Science, University of Arizona, June 1994.

[MNSS87] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Technical report, MIT, December 1987.

[MS98] S. J. Muir and J. M. Smith. Functional divisions in the piglet multiprocessor operating system. In *SIGOPS European Workshop*, September 1998.

[MSST96] Douglas Maughan, Mark Schertler, Mark Schneider, and Jeff Turner. Internet Security Association and Key Management Protocol (ISAKMP). Internet–draft, IPSEC Working Group, June 1996.

[NBS77] Data Encryption Standard. Technical Report FIPS-46, U.S. Department of Commerce, January 1977.

[NIS94] Digital Signature Standard. Technical Report FIPS-186, U.S. Department of Commerce, May 1994.

[NIS95] Secure Hash Standard. Technical Report FIPS-180-1, U.S. Department of Commerce, April 1995. Also known as: 59 Fed Reg 35317 (1994).

[Pos80] Jon Postel. User datagram protocol. Internet RFC 768, 1980.

[Pos81] Jon Postel. INTERNET protocol. Internet RFC 791, 1981.

[PS95] G. Pacifici and R. Stadler. Integrating resource control and performance management in multimedia networks. *Proc. ICC*, June 1995.

[Sch96] H. Schulzrinne. The impact of resource reservation for real-time internet services. *NRC Wkshp on Information System Trustworthiness*, 1996.

[SFG+96] J. M. Smith, D. J. Farber, C. A. Gunter, S. M Nettles, D. C. Feldmeier, and W. D. Sincoskie. SwitchWare: Accelerating network evolution. Technical Report MS-CIS-96-38, CIS Dept. University of Pennsylvania, 1996.

[SJR86] R. D. Sansom, D. P. Julin, and R. F. Rashid. Extending a capability based system into a network environment. In *Proceedings of the 1986 ACM SIGCOMM Conference*, August 1986.

[SLS97] Christoph Schuba, Bryan Lyles, and Eugene Spafford. A reference model for firewall technology, Mar. 1997. SPARTAN Symposium.

[TSS+97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, pages 80–86, January 1997.

[TY91] J.D. Tygar and Bennet Yee. Dyad: A system for using physically secure coprocessors. Technical Report CMU–CS–91–140R, Carnegie Mellon University, May 1991.

[vdML97] J. E. van der Merwe and I. M. Leslie. Switchlets and dynamic virtual ATM networks. In *Proc. of the Fifth IFIP/IEEE International Symposium on Integrated Network Management*, San Diego, CA., May 1997.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Flexible security architecture for java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[WGT98] David J. Wetherall, John Guttag, and David L. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols. In *To appear in IEEE OpenArch*. IEEE Computer Society Press, April 1998.

[Yee94] Bennet Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.