The SwitchWare Active Network Implementation *

D. Scott Alexander, Michael W. Hicks, Pankaj Kakkar,
Angelos D. Keromytis, Marianne Shaw, Jonathan T. Moore,
Carl A. Gunter, Trevor Jim,
Scott M. Nettles, and Jonathan M. Smith

University of Pennsylvania

September 1998

1 Introduction

This is an overview of work on the SwitchWare active network project, which began two years ago based on ideas about how to improve the flexibility of networks by making the network programmable. The original ideas for active networks as a whole and some comparative analysis of possible architectures are surveyed in [33]. A variety of technology trends in computing power, communication speeds, programming languages, and security have made it worthwhile to investigate network programming interfaces that allow code to be downloaded into routers within the network and invoked by the packets passing through them. At the current time there are at least a dozen AN prototype architectures under development [34, 8, 22, 2, 19, 35], a few of which have released software.

Our SwitchWare perspective was first described in [16] and has been considerably refined as we gained deeper insight into active networking. It was the first active network prototype to be publically released, and is implemented largely in the Caml [12] dialect of the ML programming language, using the OCaml implementation. We were instigated to use Caml because of its success in several other distributed computing and networking projects such as Ensemble [15] and MMM [26]. We found ourselves able to achieve good results quickly because of the flexible and rigorous ML programming model, and the quality of the OCaml compiler has allowed us to consistently outperform AN prototypes based on Java.

This note provides a very brief overview of the SwitchWare architecture, then provides short explanations of the major parts of SwitchWare. These include: a dynamic loader, which serves as the basis of code mobility in SwitchWare; an active bridge, which is able to download code that allows it to adapt to changes in standards for network topology; the PLANet active internet, in which all packets contain programs written in PLAN, the Packet Language for Active Networks; the SANE (Secure Active Network Element) system for secure active network elements; and the QCM (Query Certificate Manager) system, which provides a security policy language and certificate retrieval mechanism.

Most of the SwitchWare source code and papers can be found on the WWW home page for the project: `www.cis.upenn.edu/~switchware`.

2 SwitchWare Architecture

SwitchWare is based on a three-level architecture which is described in detail in [2]. The three layers are depicted in Figure 1. At the top level, there are lightweight mobile programs that fit within a packet, which we call *active packets*. We have experimented

*Presented at the 1998 ML Workshop in Baltimore

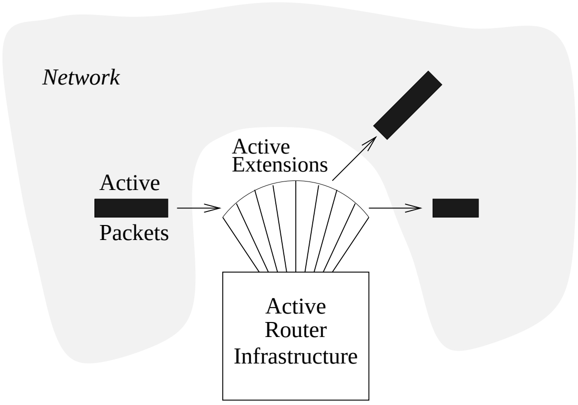

Figure 1: SwitchWare Layers

with both PLAN (Packet Language for Active Networks) and Caml as the language at the active packet level. PLAN is a scripting language that was developed by the SwitchWare project. A complete PLAN program must fit within each packet. However, using a fragmentation/reassembly service, larger programs can be transfered in small pieces, reconstructed at the destination, and subsequently be executed. Programs in active packets are very limited in capabilities, for example they cannot leave state at active nodes or communicate with other packets without using services, since they are untrusted.

The second layer is called the *active extension* layer. This layer consists of Caml programs that can be dynamically loaded (over the network, or from the local disk on an on-demand basis) into SwitchWare active nodes. The *extensions* are used to provide *services*, which can be invoked by active packets. A number of *services* are also offered by the underlying runtime system.

The lowest layer of the SwitchWare architecture is the *infrastructure* which supports resource allocation and enforces the rules for downloading *switchlets (active packets and extensions)*. This is implemented by a modified version of the OCaml runtime system running on top of an operating system such as Linux.

Using this basic architecture, we've built a number of standalone systems and system components. PLANet is our active internet in which all packets contain PLAN programs; basic network services (such as routing, address resolution, etc.) are implemented as active extensions. PLANet may be run directly on top of Ethernet without the aid of IP [32]. ALIEN is node architecture which differs from PLANet in that active packets consist of Caml bytecodes. As our first experiment with ALIEN we built an Active Bridge [4] to experiment with node extensibility.

We have also developed a security-services infrastructure, called SANE [3] which has been nearly fully implemented in the context of ALIEN, and partially implemented in the context of PLANet. Among the services offered by SANE are secure bootstrapping [5, 6], cryptographic primitives (digital signatures [29], hashes [30], secret key encryption [28]), key establishment and principal[1] authentication [14], a safe distributed naming scheme, and policy-controlled resource and access control through language restrictions and use of KeyNote [9] and PolicyMaker [10, 11] policy certificates.

We have also implemented a second security infrastructure called QCM. Like KeyNote and PolicyMaker, QCM provides a general language for expressing security policies and a verifier for checking that policies are satisfied. In addition, QCM can automatically retrieve certificates from the network using PLAN packets. QCM thus supports transparent, application-independent, distributed security policies in the active network.

3 ALIEN: An Active Loader for Caml

ALIEN is a program that we have built to allow loading Caml programs into a running Active Network system and to provide the support necessary for the loaded switchlets to access (carefully limited) parts of the system. In essence, it consists of Objective Caml's `Dynlink` library wrapped by the additional functions needed by our system. Logically, ALIEN is

[1] By principal we mean the holder of a public key, in some public key cryptosystem, such as DSA[29].

divided into three pieces: the Active Loader, the Core Switchlet, and a set of libraries.

The Active Loader is the small, fixed part of the system which takes control initially and coordinates the activities of ALIEN and its loaded switchlets. Its core is a loop which reads from its standard input looking for directives which will load a new switchlet from disk or execute a 'well-known' function from a previously loaded switchlet. A load request calls Dynlink which evaluates the byte-code file. In a manner similar to MMM, we provide a function which can be called to arrange to add a mapping between a string and a function into a hash table. When a request to execute a function is entered, we use this table to map from the name of the requested function to an actual function which is then executed. The loading and executing functionality along with a function that we added which will dynamically load from memory instead of from disk are then available to functions within the Core Switchlet and the Active Loader once the system is running.

The Core Switchlet is the set of modules we have found necessary to support our Active Networks applications. It is privileged in the sense that it can access any facility provided by Objective Caml. It provides an interface which borrows heavily from the structure of MMM: we use module thinning to control the functionality available to loaded switchlets. (We discuss the safety aspects in Section 6 when we describe SANE.) We have five major modules in this part of the system: Safestd, Safeunix, Log, Unixnet, and Safethread. The Safe* modules are slightly modified versions of those provided with MMM. In particular, we go further in excising I/O functions since we do not have a user to whom a dialog box can be presented. The Log module is a simple facility that allows a switchlet to log a message. Where, how often, and by what method such messages are logged in is deliberately unspecified. While debugging, we log all messages to a file. Obviously, an attacker could use such a mechanism to fill the disk, so we anticipate a production version which limits the number of messages per unit time by either discarding messages or delaying the thread that is overproducing messages.

The Unixnet module is our facility for accessing the network. To provide the access we required, we had to make some small changes to the Objective Caml runtime system. The first of these changes allows access to raw Ethernet frames as provided through the (idiosyncratic) Linux socket interface. This is critical for those times that we do not use IP as an infrastructure. We also provide some additions that allow changing characteristics of network interfaces (*e.g.*, putting an interface into promiscuous mode so that it will receive all packets on the Ethernet instead of only those addressed to it).

The libraries are the least well-defined portion of the system because they are unprivileged and thus completely loadable. Some of the pieces that we have built range from a (skeletal) IP implementation to implementations of some cryptographic routines. Switchlets that are considered libraries differ from other switchlets only in the intent of the programmer. For more details on the architecture, see [1].

Objective Caml provided several characteristics that were needed for ALIEN: the ability to load code dynamically, strong typing, module thinning, and a homogeneous representation for switchlets moving between different machine architectures. Dynamic loading and architecture independence simplify the implementation of any Active Network system dramatically; given the time constraints for implementing ALIEN, they were required. The importance of strong typing has been widely discussed generally; we particularly were interested in using it to prevent unwanted interactions between switchlets and to ease the distributed debugging problem. Finally, module thinning allows us to control what pieces of the Objective Caml runtime system are accessible to an anonymous switchlet. With the advent of SANE, we now have switchlets which are not (necessarily) anonymous and intend to explore extending our current two level privilege scheme into a scheme where the privileges available to a switchlet vary based on the credentials presented by that switchlet.

Some performance issues are discussed in Section 5.

4 An Active Bridge

The Active Bridge [4] is an experiment with active extensions that provides a bridge for 100 Mbps Ethernet LANs and an automated mechanism for installing new software versions with minimal disruption. We build the bridge in three layers, echoing the description of a bridge in [31]. These layers are a buffered repeater, a learning algorithm, and a spanning tree algorithm. See [31] for more details on each of these.

To show the utility of building the bridge piecewise, we demonstrate the ability to transition between incompatible protocols, in this case spanning tree algorithms (STAs). We have implemented two different STAs, an *old* and a *new*. With the old STA running, we load (but do not start) the new STA. We also load a control switchlet. The control switchlet watches for a specific signal, 'suspends' the old STA, collects information from it, and starts the new STA. The network is given time to restabilize and then the information collected is used to check if the new STA is running properly. If not, the control switchlet kills it, restarts the old switchlet, and signals the control switchlets running on the other nodes of the network that they should also transition back to the old (stable) STA.

With the Active Bridge, we have done a set of experiments to understand both the costs of loadable modules and the cost of our demultiplexing architecture. To test the first case, we have a version of the Active Loader which supports only the Bridge. In this case, we see a throughput as measured by `ttcp` of 61 Mbps. Our second experiment used the general Active Loader and gave a throughput of 54 Mbps. Of the per-packet processing time, approximately 70 μs is calling `recvfrom()`, 70 μs is processing in the Bridge, and 30-40 μs is calling `sendto()`.

5 PLANet

PLANet is our active internet in which all packets contain programs written in PLAN, the Packet Language for Active Networks. A PLAN program can conceptually be viewed as a replacement for a packet header. If one thinks of IP headers as programs and IP routers as interpreters of these programs, then one can see that IP defines a very restricted programming language. Replacing the header with a small program seems feasible, but once one has gone that far there seems to be less need to distinguish between the header and the payload, since the program can contain the payload as a datastructure. Thus PLAN allows a packet to simply contain a program to be evaluated by a PLAN active node; the evaluation may cause the packet to be routed or delivered, or may have some more interesting effect, like changing the router behavior or creating new packets. With this increased flexibility comes increased risk since a packet that could replicate or execute itself without bound could swamp the network. Privacy is clearly also a concern, depending on what the more general kinds of packets are allowed to do.

	Packet level	Service level
Language	PLAN	flexible
Code location	in packet	on node
Expressibility	limited	general purpose
Authentication?	no	when needed

Table 1: Comparison of the Packet and Service levels

We designed PLAN to be essentially a scripting language; it can be viewed as a special-purpose language providing an API for remote evaluation. What PLAN 'scripts' together are calls to more general-purpose, router-resident service routines. In PLAN, calls to service routines resemble normal function calls except that service routine names are resolved dynamically, at PLAN evaluation time. In PLANet, service routines are implemented as active extensions written in Caml. This two-level architecture of PLAN and service routines is summarized in Table 1, which is drawn from [22]. This combination provides a powerful programming style which we describe in [21] and have illustrated in PLANet. PLAN provides very efficient, light-weight mobile agents that invoke services to achieve flexible network utilization that can be customized to the needs of particular users or applications.

PLANet [23] is an internetwork in which PLAN forms the interoperability layer over various (virtual) link layers (currently we support Ethernet and IP). In order to implement the PLAN semantics of remote evaluation, we provide active extension-based im-

plementations of various internetworking tasks, such as routing, address resolution, name resolution, etc. Each PLANet active node has the following components:
(1) packet processing core
(2) network functions
(3) the PLAN interpreter
(4) library of service routines

Packet Processing Core

The operation of a PLANet node is depicted in Figure 2, which is drawn from [23]. In its idle state, an active node has one thread running for each network interface type, waiting for input on that interface[2]. Once a packet arrives, the thread makes an upcall to the appropriate *handler* to process the packet (currently, all packets that are not PLAN packets are discarded). This code is a slight extension to the Unixnet module described earlier (which makes use of a queue rather than an upcall). If the packet has reached its evaluation destination, it is passed to the PLAN interpreter to be evaluated; otherwise, it is routed onwards. Evaluation consists of unmarshalling the packet into a structured type followed by the actual interpretation. During interpretation, PLAN programs may make service calls, perform remote evaluations, recursively call the PLAN interpreter, etc. If a remote evaluation occurs, a PLAN packet is constructed, the next hop determined, and is finally sent out the appropriate network interface. Once packet processing completes, the upcall returns, and the network-interface thread continues to look for new packets.

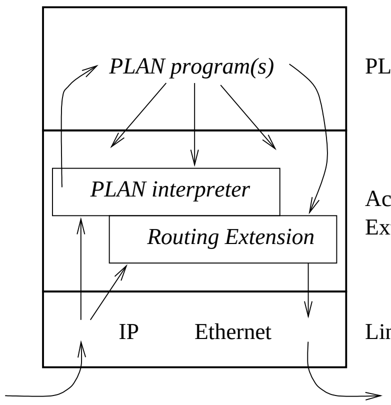

Figure 2: PLANet node architecture

[2]We would rather have one thread per interface, as opposed to interface-type, but our use of the Linux special socket for receiving Ethernet packets would make that inefficient.

This architecture arose after some analysis which shed light on our misimpressions of the costs of certain OCaml operations. For instance, we initially thought that thread overheads would be fairly low, since the byte-code interpreter employs a user-level thread scheduler. For this reason, our initial packet-processing approach forked a thread for every new packet that arrived, rather than performing an upcall. This approach is preferable in that the amount of time required to execute a PLAN program is not tightly bounded, which means that in our current implementation we could lose packets because the network interfaces are not being serviced fast enough. However, we found this potential problem to be far less than excessive overhead of context-switching among many (> 20) threads; on our 300 mHz Pentium-II's, context switching typically took about 110 μs. Since a context-switch is favored to occur whenever I/O is ready, this meant that the interpretation threads rarely got the CPU under high packet loads: the majority of the CPU time was spent reading in packets and context-switching.

We also found garbage collector overhead to be more than we had suspected. The fact is that the OCaml collector is very good, but that *any* additional overhead seriously impacts performance—each 130μs or so spent out of the packet processing loop meant a lost packet. We managed to cut GC cost dramatically by not incurring additional copies where possible, meaning that we would GC less often. Caml's mutable strings were helpful in this regard.

Network Functions

In order to maintain the routing tables needed to implement remote evaluation, we needed to implement a routing protocol. We based our implementation on RIP [20] with the exception that rather than using special packet formats to advertise routes, we used PLAN programs. Thus, the majority of the routing protocol implementation was done in Caml.

The routing software updates its routing table based on two forms of events: received advertisements and entry expirations. To handle the expirations, we simply wrote a thread which wakes up periodically and times out expired entries. Another thread broadcasts the node's route advertisements periodically: either every 30 seconds, or when some other change occurs in the table (such as a route times out, or a new route is received). Here it would have been useful to have a version of `wait` which could time out, as is supplied with Java. As it was, we had to simulate this by forking a timing thread (which as we have indicated is expensive).

The implementation of the routing protocol shares common themes with other table-based network services we've written, such as address resolution, fragmentation/reassembly, reliable transport, etc. In all of these, a maintenance thread is used to time out expired entries, while the contents of the table are modified via service calls from PLAN programs. In the address resolution protocol, a similar technique for broadcasting remote evaluations is used.

The PLAN Interpreter

The PLAN interpreter itself is fairly simple – it was written in only a few days with the help of OCaml's `ocamllex` and `ocamlyacc`. Programs are compiled at the source into abstract syntax trees which are marshalled and transmitted to the evaluation destination. This approach improves both space and time efficiency on the routers. Type-checking is done dynamically during interpretation (since the AST is already type-tagged). This was originally to simplify our implementation, but there is some belief that the overall packet processing time might improved this way.

The interpreter is augmented by a hierarchical symbol table which maps identifiers to either PLAN or service functions. The topmost portion of the table is implemented as a `Hashtable` and maps to service routines (stored as first-class functions). As the PLAN program evaluates, association lists are used to perform local bindings.

Perhaps the biggest payoff to using Caml was in the development of the interpreter. The implementation was greatly simplified through the use of structured types (for the AST) and higher-order functions (to store the service routines). A subsequent development of the interpreter in Java took 3 weeks as opposed to 3 days, and required about 3 times as much code.

Service Routine Libraries

In order to augment the limited abilities of PLAN programs, we needed to provide a fairly rich library of service routines, all of which were implemented in Caml. Typically this was a straightforward affair in which we wrapped an Caml function with a wrapper that allowed it to recieve and return PLAN arguments. Some of the services that we provide include packet interrogation functions (get the source of the current packet), network service functions (show me the routing table, get the name of the current host), cryptographic functions for security (as described in Section 6), and a service loading new services. In many cases, OCaml libraries existed to allow easy implementation of these services. The `Unix` module was especially useful. In the case no library function was available, we found the interface to C to be very convenient. In fact, it was these features that made programming PLANet possible – while ML provides many language level advantages, it would have been of no use to us without adequate libraries and a solid foreign-function interface.

6 SANE

The SANE architecture aims to provide security services and guarantees to mobile programs and switches in an active network. It achieves this through use of a number of different mechanisms, cryptographic and otherwise.

Arguably the most important Caml features used in SANE are strong typing and module thinning. Since switchlets run in the same address space, implemented as threads, we need to guarantee process isolation through mechanisms other than virtual memory protection. Caml's strong typing and dynamic checks (such as array bound checking), and the lack of arbitrary memory pointers, allow us to make that guarantee.

Module thinning allows us to control access to parts of the runtime system based on privileges. Those privileges take the form of KeyNote [9] and PolicyMaker [10] assertions (roughly corresponding to certificates). Privileges are checked once when the Caml switchlets are linked into the system (to determine whether a switchlet has the right to access a facility), and then as necessary (*e.g.,* when more memory is needed, or after a certain number of CPU cycles has been consumed, etc.).

Caml was very useful in developing the key management protocol used in SANE in a very short time. Implementations of similar (albeit somewhat more featureful) protocols [27, 24] in the past have taken considerably more time to develop and debug, and required about one order of magnitude more C code. For comparison purposes, developing a roughly equivalent in capabilities version of the ISAKMP/Oakley protocol (the key management protocol in IPsec [7]) took a full month, and resulted in about 10000 lines of C code; developing the SANE key management protocol took half as long, with the code being less than 1000 lines of code. Performance was not perceptibly impacted either; the ISAKMP protocol performed a Diffie-Hellman [13] key agreement and a number of cryptographic hash operations for authentication, in an average of 5 seconds. The SANE protocol performed the same Diffie-Hellman operation and two additional DSA [29] signatures/verifications (but less cryptographic hash operations) in 4.8 seconds.

We also implemented and measured the performance of a number of cryptographic primitives. In particular, we implemented the DSA digital signature algorithm, the DES [28] encryption algorithm, and the SHA1 [30] cryptographic hash. The tables 2, 3, and 4 show some performance measurements among the bytecode and native Caml versions and some fully optimized C implementations. The measurements were taken on an Alpha 21164SX at 533MHz. The SHA1 and DSA algorithms were implemented using both the Int32 package (which allows the same code to run on both Alpha and Pentium platforms) and the native Alpha integers.

Caml	Int32	bytecode	86.4 s
		native	61.9 s
	Alpha ints	bytecode	36.0 s
		native	2.4 s
C			0.3 s

Table 2: **Time to SHA-1 hash 4MB of data**

Caml	Alpha ints	bytecode	99.3 s
		native	16.7 s
C			1.0 s

Table 3: **Time to DES encrypt 4MB of data**

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 authentication, 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 tried to limit the amount of non-Caml code in our system. Thus 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. We also believe that in the future, 'Just-In-Time' compilation techniques can narrow this gap in performance.

sign	Caml	Int32	bytecode	27.0 ms
			native	12.9 ms
		Alpha ints	bytecode	20.9 ms
			native	11.8 ms
	C			2.8 ms
verify	Caml	Int32	bytecode	41.4 ms
			native	22.1 ms
		Alpha ints	bytecode	35.1 ms
			native	20.6 ms
	C			5.0 ms

Table 4: **Digital Signature Timings**

7 QCM

Any large scale security architecture that uses certificates to provide security in a distributed system will need some automated support for moving certificates around in the network. This is especially likely for active networks, where authorization for the installation and use of services is likely to involve many administrative domains and users. We have developed a system called a *Query Certificate Manager (QCM)* which supports automated retrieval of certificates in a distributed context, driven by a QCM policy verifier. Like other verifiers, QCM takes a policy and certificates supplied by a requester and determines whether the policy is satisfied. However, if the policy is not satisfied, QCM can examine the policy to decide what certificates might help satisfy it and obtain them from remote servers on behalf of the requester. The QCM policy language is based on set comprehensions and standard distributed query optimizations are used to minimize network traffic. Retrieval is secured through the use of digital signatures, which are checked automatically. A novelty of QCM is its ability to combine certificates that it retrieves with certificates supplied by a client; supplied certificates are used to short-circuit remote retrieval or deal with QCM when the verifier has its automatic retrieval ability 'turned off'. A network of QCM nodes can contain a mixture of servers that do online and offline signing of the certficates they provide. QCM is designed to deal with failure on a 'best effort' model: a subset of the desired certificates are delivered if some servers are not responding.

We have implemented several variants of QCM; the primary implementation uses IP sockets to send messages between QCM servers. Another variant runs in a single-machine mode, simulating distributed computing using threads. This variant is useful for prototyping, debugging, and simulating QCM computations conveniently. A third variant of QCM was built to run on top of the PLANet. In this implementation QCM carries out network communication using PLAN active packets. We have used this system to implement a security infrastructure for access control of PLANet switchlets. We have also implemented a graphical user interface for instigating and observing QCM computations. The GUI allows QCM nodes to be registered and report events; these are used to create a 'movie' tracing the QCM queries. The GUI was built using CamlTk, Caml's interface to the Tk graphics toolkit. Since both Caml and Tk run on a number of systems, the QCM system and its GUI are portable; we have it running under Windows and several variants of Unix. This wouldn't be possible if we were forced to use an OS-specific graphics package.

The QCM implementation is essentially an interpreter, so as you might expect, Caml's data types, pattern-matching, and automatic memory management allowed us to quickly build the system with relatively little code. Although it offers both verification and certificate retrieval, QCM is about the same size as other systems that only offer verification. The IP, PLANet, and simulated variants of QCM combined take up about 9,000 lines of Caml; this includes approximately 2,000 lines for basic cryptographic algorithms (SHA, DSA, and key generation). The GUI adds another 2,500 lines of code. In comparison, the SDSI 2.0 distribution [25] is about 13,000 lines of C and Perl, not including basic cryptographic algorithms or GUI support.

Thread management in Caml, which caused many performance problems for PLANet, is also crucial for QCM. QCM must be multi-threaded: in processing a request the verifier might have to make time-consuming remote queries for certificates. Other threads should not be stopped while QCM waits for a reply. Lowering the thread overhead will be important since we are competing with verification and retrieval systems written in C.

Work on QCM is described in [17, 18].

8 Summary

Caml has been an effective tool for producing reliable and efficient networking software quickly. We have mentioned a number of areas where we think improvements in Caml would translate directly into benifits for our software, and we hope that some of these opportunities can be pursued in the future. Developments like the Ensemble group enabling Caml bytecode to run in the Windows NT kernel may well yield other substantial gains.

9 Acknowledgements

We would like to acknowledge William Arbaugh, who was a co-designer of the SANE system, and architect of our secure bootstrapping method.

References

[1] D. Scott Alexander. ALIEN*: A Generalized Computing Model of Active Networks*. PhD thesis, University of Pennsylvania, to appear December 1998.

[2] D. Scott Alexander, William A. Arbaugh, Michael Hicks, Pankaj Kakkar, Angelos Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The switchware active network architecture. *IEEE Network Magazine*, 1998. To appear in the special issue on Active and Controllable Networks.

[3] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromyts, and Jonathan M. Smith. A secure active network environment architecture: Realization in SwitchWare. *IEEE Network Magazine*, 1998. To appear in the special issue on Active and Controllable Networks.

[4] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active Bridging. In *Proceedings, 1997 SIGCOMM Conference*. ACM, 1997.

[5] 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.

[6] William A. Arbaugh, Angelos D. Keromytis, David J. Farbe r, and Jonathan M. Smith. Automated Recovery in a Secure Bootstrap Process. In *To appear in Network and Distributed System Security Symposium*, pages 155–167. Internet Society, March 1998.

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

[8] Smart packets. `http://www.net-tech.bbn.com/smtpkts/ smtpkts-index.html`.

[9] M. Blaze, J. Feigenbaum, , J. Ioannidis, and A.D. Keromytis. The keynote trust management system. Work in Progress, June 1998.

[10] 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.

[11] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the policymaker trust management system. In *Proc. of the Financial Cryptography '98 Conference*. Springer, 1998.

[12] Caml home page. `http://pauillac.inria.fr/caml/ index- eng.html`.

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

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

[15] Ensemble home page. `http://simon.cs.cornell.edu/Info /Projects/Ensemble`.

[16] Dave J. Farber, David C. Feldmeier, Carl A. Gunter, Scott M. Nettles, William D. Sincoskie, and Jonathan M. Smith. Switchware: Accelerating network evolution with a software switch for active networks.

[17] Carl A. Gunter and Trevor Jim. Design of an application-level security infrastructure. In Catherine Meadows and Hilarie Orman, editors, *DIMACS Workshop on Design and Formal Verification of Security Protocols*, September, 1997.

[18] Carl A. Gunter and Trevor Jim. Policy directed certificate retrieval, July 1998.

[19] John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting. Liquid software: A new paradigm for networked systems. Technical Report TR 96-11, University of Arizona, June 1996. http://www.cs.arizona.edu/liquid/.

[20] C. Hedrick. Routing information protocol. Technical report, RFC 1058, June 1988.

[21] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. Network programming with PLAN. In *Workshop on Internet Programming Languages*. Springer, 1998. To appear.

[22] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of the International Conference on Functional Programming Languages*. ACM, 1998.

[23] Michael Hicks, Jonathan T. Moore, Scott Alexander, Carl A. Gunter, and Scott Nettles. Planet: A active network testbed. 1998.

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

[25] Butler Lampson and Ron Rivest. SDSI—a simple distributed security infrastructure. `http://theory.lcs.mit.edu/~cis/sdsi.html`.

[26] Françoìs Louaix. A web navigator with applets in Caml. In *Fifth WWW Conference*, 1996.

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

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

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

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

[31] Radia Perlman. *Interconnections: Bridges and Routers*. Addison-Wesley, 1992.

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

[33] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[34] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OPENARCH*, April 1998.

[35] Y. Yemini and S. daSilva. Towards programmable networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, October 1996. http://www.cs.columbia.edu/~dasilva/netscript.html.