Safety and Security of Programmable Network Infrastructures

D. Scott Alexander William A. Arbaugh Angelos D. Keromytis
Jonathan M. Smith [*]
University of Pennsylvania

September 7, 1998

Abstract

Safety and security are two reliability properties of a system. A *"Safe"* system provides protection against errors of trusted users, while a *"Secure"* system protects against errors introduced by untrusted users. There is considerable overlap between mechanisms to support each property.

Requirements for rapid service creation have stimulated the development of programmable network infrastructures, where end users or service providers can customize the properties of a network infrastructure while it continues to operate. A central concern of potential users of such systems is their reliability, and most specifically their safety and security. In this paper, we explain the impact the network service model and architecture have on safety and security, and provide a model with which policies can be translated into restrictions of a general system.

We illustrate these ideas with the Secure Active Network Environment (SANE) architecture, which provides a means of controlling access to the functions provided by any programmable infrastructure.

1 Introduction

A variety of proposals for exposing some control of network infrastructures have been made. These proposals have been driven by a common need, that of accelerating service creation. Sometimes services can be created using features of existing systems, *e.g.,* a set of stocks we are interested in for price updates. The IP Internet has successfully demonstrated a service introduction model where advanced services are overlaid on the underlying IP datagram forwarding function.

Our interest in this article is the somewhat aggressive proposal of a *programmable* network infrastructure. What this means is that a user or operator of such a network has facilities for

[*] `http://www.cis.upenn.edu/~{salex,waa,angelos,jms}`. This work was supported by DARPA under Contract #N66001-96-C-852, with additional support from Intel Corporation.

directly modifying the operational semantics of the network itself. Thus, the role of network and endpoint become far more malleable for the construction of new applications. This is in contrast to the service overlay model (e.g., the TCP reliable bytestream protocol) where service introduction at the "edge" of the virtual infrastructure is very easy, but changes in the infrastructure itself have proven very difficult (*e.g.*, RSVP and Mbone).

1.1 Why worry?

Since network infrastructures are shared, they must be very reliable. For example, in traditional telephony, an independent power system ensured operation in the face of external electrical failure. The connection-oriented service model dictated extremely reliable switches, as the failure of any one switch induced connection (and hence service) failure. The IP Internet infrastructure is presumed to be unreliable, but in fact employs dynamic routing so that the packet delivery function turns out to be very reliable in a reasonably well-connected network topology (TCP ensures end-to-end reliability.)

Exposing this shared infrastructure to users must preserve (some) expectations of reliability, while allowing the infrastructure to be multiplexed to derive the economic advantages of sharing. Since we understand how to build reliable components and links, the central issue is the service model and its implications for delivering service with a multiplexed infrastructure. With the extended service model of a programmable infrastructure, these questions become much more complex than in simple time-division multiplexing of a link. For example, if a new version of IP is desired by one user, it is not clear whether other users must use that version. If the new version is designed not to affect other users, what steps must a user take to be allowed to use this new version?

1.2 Safety and Security of a network infrastructure

In a network with a minimal service model, such as the Internet, most of the security and safety concerns are delegated to the network edges, where services such as name resolution are implemented. Services internal to the network, such as routing and route updates, introduced security vulnerabilities; for example, if a router purposely or mistakenly advertises low-cost routes, huge volumes of traffic will be routed toward it, rendering that part of the network practically inoperable. These vulnerabilities can be addressed to some extend by using the IP security protocols, but a number of problems persist.

1.3 Outline of the rest of the paper

In the next section we provide some historical background on the predecessors of today's programmable networks. In Section 3, we introduce and explain the threat model for programmable network infrastructures. Section 4 explains how an infrastructure can be protected by restricting the actions which can be performed on it. Section 5 points out the important difference between an infrastructure with safe and secure nodes, and a network which is safe and secure as a system.

Section 6 illustrates many of the concepts explained in the paper with highlights from the Secure Active Network Environment (SANE) [AAKS98] and briefly discusses other approaches to active network security, and Section 7 concludes the paper with a short list of open problems.

2 History of Programmable Network Infrastructures

It is a misconception that programmability is a recent advance in communications networks. Telephony, for example, has provided many new services as a result of the programmability of electronic switching systems; the 5ESS is essentially a highly reliable minicomputer [MBD+83]. As well, from the very start, the Internet infrastructure was built on a programmable platform, the Interface Message Processor (IMP), a specialized minicomputer system. The issue is the nature of the programmability. The telephony infrastructure was (and largely remains) the domain of the telephone companies. They sold a *service* and retained control of the means (*i.e.*, the software systems) of providing the service. The service interface did not, and does not, include any means of modifying the service. Likewise, the Internet infrastructure does not allow any means of flexible customization of the service provided by the Internet itself; all service customization (as with telephone answering systems or facsimile systems) is overlaid at the endpoints. The model of programmability we are interested in, then, is one where the infrastructure itself can be customized, or more generally, programmed.

The most important historical basis for programmable infrastructures come from AIN [BCR], computer operating systems and process migration/mobile code. AIN, an attempt by the telephony industry to accelerate service creation, was successful in that it demonstrated acceleration of service introduction using a programmable service composition model within the context of a telephone call; the programmable phone system has reduced service introduction times by about a factor of 50. Extremely restricted AIN access has been sold as a service to selected customers such as airlines. Computer operating systems have illustrated ideas, especially in the domain of safety and security, for providing a carefully-designed set of restrictions on resource access to permit multiplexing of the hardware resources. Mobile code, which follows directly from earlier work in process migration [Smi88], explores issues in distributed control, distributed access to resources, and local support for remotely generated code.

The single most significant system in terms of research results was the Softnet [FZ81] system designed and implemented at the University of Linköping in Sweden. The Softnet nodes were packet radio systems consisting of a pair of microprocessors, one running a link controller kernel which provided real-time services, and the other running a multitasking Forth system which allowed the node behavior to be modified on the fly. An interesting property of this system was that the memory manager released memory (including that containing code) if it hadn't been touched/used recently, essentially implementing a soft-state model for a programmable infrastructure.

3 Threat Model for Programmable Network Infrastructures

Notions of safety and security are unfortunately context-specific. It would be convenient, of course, if security were a simple testable property, like a number's oddness or evenness, but this is not the case. The essential task in developing a threat model is identifying reasonable expectations users might have of a programmable network infrastructure, and once that is done, identifying ways in which those expectations might fail to be met.

Programmable network infrastructures are characterized both by their role as network infrastructures (e.g., they can transport data in some fashion), and the customizability of the infrastructure to meet application needs. Threats shared with traditional network infrastructures include uncontrolled unfairness of service (controlled unfairness of service is often desirable; this is the essence of "Quality of Service"), and unauthorized access to the control plane (*e.g.*, various route update forgeries on the IP Internet).

The difference with a programmable network infrastructure is that a portion of the control plane is *intentionally* exposed, and this can lead to far more complex threats than exist with an inaccessible control plane. For example, an overload-based denial of service attack on control plane services may not fundamentally inhibit packet transport, but may deny functions required for the (proper) operation of many applications. A denial of service attack on the transport plane may also inhibit access to the control plane. Unauthenticated access to the control plane can have severe consequences to the security of the whole network.

4 Protection by Restriction

The basis of safety and security in any setting is controlled access to resources. The resource control is dictated by a *policy*, which defines which operations can be performed on the resources. Concrete representations used to reflect policies in operating systems include access control lists (ACLs) and capabilities. An ACL, for example, is a logical 3-tuple of $<$object, subject, action$>$, or more specifically, $<$resource, user, permissions$>$. The information in these representations is used to provide policy *enforcement*. Thus, an ACL (or a capability) can be viewed as a set of restrictions that govern the access to an object.

4.1 Policy Enforcement

Wherever there is a security policy there are restrictions. The interesting system design question is how these restrictions are enforced, and under what set of assumptions.

Thus, a virtual memory typically restricts access to certain areas of the real address space where specialized data such as system tables and interrupt vectors reside. A virtual machine operating system provides access to all instructions except those which must be restricted, *e.g.*, the ones which update mappings between real and virtual addresses. In this design, a hardware-augmented interpreter is checking restrictions at system run-time (albeit rather efficiently with modern hardware). A second design point is to statically check for the existence of "bad" instructions or addresses in an executable image or program before execution begins. The challenge with this design point is that many computer languages do not provide enough information to determine whether the program is safe prior to execution. This is in a way not surprising; a completely general solution to this problem would solve the halting problem, an impossibility. In any case, the language system must determine correctly whether all restrictions are met, and any loader must verify that the language system has approved of the program before executing it. This becomes rather complex in multiple language environments, which are a requirement of many operating systems.

Figure 1 illustrates a programming language system overlaid on a traditional operating system. Protection checks which are done at the language system level can be subject to various static versus dynamic checking tradeoffs (*e.g.,* array-bounds checking), while most OS decisions must be dynamic, even if logically "cached" in the manner of Synthesis [PMIM88]. An additional issue with such a structure, illustrated with the heavy dashed line, is the cost of operating system boundary-crossings. This suggests a preference, where possible, for performing restrictions at the programming language level, and performing them statically at that level.

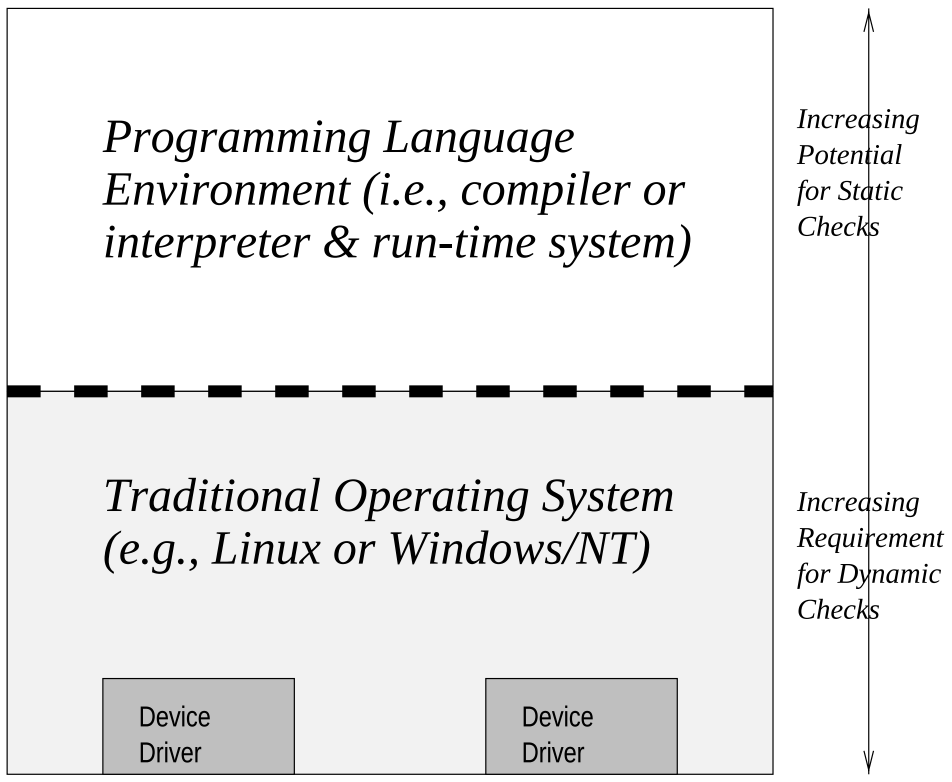
Programming Language Environment (i.e., compiler or interpreter & run-time system)
Increasing Potential for Static Checks
Traditional Operating System (e.g., Linux or Windows/NT)
Increasing Requirement for Dynamic Checks
Device Driver
Device Driver

Figure 1: Some tradeoffs between Programming Languages and OS

4.2 Restricted Languages

The attraction of static checking is that it need be done once, and thereafter restrictions need not be checked, assuming the program does not change. If the programming language used is restricted, perhaps by design, then checking that a program is valid may be both possible and sufficient. An example of this approach has been used by the designers of the Programming Language for Active Networks (PLAN). PLAN is simple, resource-bounded and has the flavor of Scheme and ML; it has been applied to a variety of networking tasks. Since the language is so simple, PLAN programs themselves can do little harm; in this, it is like the UNIX shell. As with the shell, the language itself is limited, and its power comes from the ability to augment the language with extensions, which in the UNIX case are programs executable by the shell.

An alternate approach to simple special-purpose languages is to restrict access to features of a general-purpose programming language. The potential disadvantage of this approach is the need to carefully analyze the safety and security implications of access to features and services. However, this will be necessary whenever extensions are required in the special-purpose language approach as well. A potential advantage of the restricted general-purpose language approach is that an additional extension language is not needed; any and all services can be type-checked etc., in a unified manner.

Hybrid systems may be the most pragmatic approach, where static analysis is able to ensure programs are conformant, and run-time checking is used to perform checks which cannot be performed statically. Most modern programming language implementations are in fact hybrids; a ready example is the popular Java programming language. Many properties of Java programs can be analyzed before the program is executed by the compiler and/or verifier, and runtime restrictions are enforced by the Java Virtual Machine (JVM). The risk of this approach, illustrated with many of the security flaws in early Java implementations, is that the security of the JVM relies upon the static checks of the verifier. If the checks in the verifier are less stringent than those in the compiler, the result is the possibility of programs which can run on the JVM which would not compile because of their violation of restrictions.

5 Node vs Network Safety

Controlled access to resources is the basis of safety and security. Implicit in our discussion of programming environments and virtual machines is that our major concern is the safety and security of a single node. While systems have been designed [Per92] which can cope with faulty nodes, even such designs must rely on having at least a simple majority of nodes which can be trusted. Thus, node safety is the basis of network safety. However, network safety is a more complex problem than node safety, since the network is an aggregate of distributed resources.

Consider the following example: multicast service is an essential function in data networks, and thus a “program” which supports multicast must be possible. The simplest such program is one which sends a packet from one port to two other ports, constructing a simple binary multicast tree. At a given node, this program would be analyzed and validated. Now imagine further that a copy of this program is associated with each input port on each node in the network, a valid configuration using a valid program. When the packets enter the network, two copies will emerge from the node, and so on until the network collapses under the load. Thus, the simple example is *node-safe*, but not *network-safe*. This idea is illustrated by Figure 2, which shows that network-safe

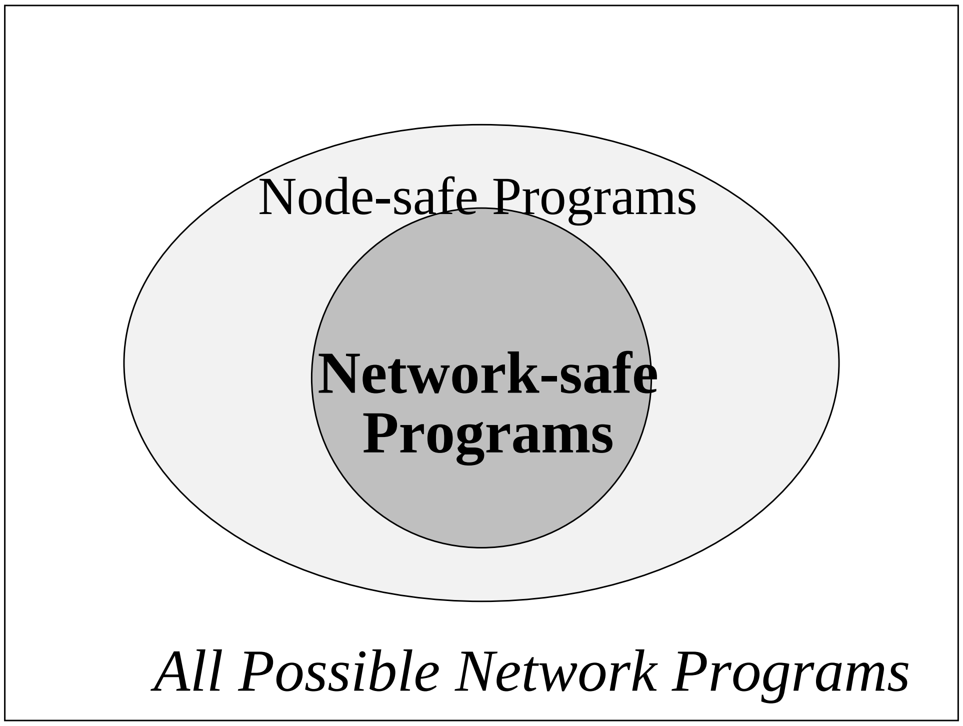

Figure 2: Restriction to node-safe and network-safe

programs are a subset of node safe programs, which in turn are a subset of all programs. In each case, the important engineering issue is to discover techniques which keep systems in the desired subspace.

The most promising approaches to this problem are those which treat the programmable network infrastructure as a distributed system. In this way, local actions can be combined into a global behavior; this is the basic principle used by TCP/IP congestion control, where local responses to inferred congestion protect the network (at least from load caused by TCP).

Given the scale of modern network infrastructures, it seems unlikely that system restrictions based on dynamic checking will be appropriate, since the cost of cooperation between nodes will be high. We believe that an infrastructure which separates policy enforcement into two phases, robust static analysis before network entry, coupled with checking that this analysis has been done, is more likely to succeed. This is completely pragmatic, as computation at the edges of the network will be cheap, and verifying that the checking has been done is typically much cheaper than the checking itself. We note that the technological basis for these ideas is being established today with systems such as Proof-Carrying Code [Nec97].

The other part of the picture is specifying behavior in a distributed system. Fortunately, there has also been significant recent progress in this area. The most promising thrust is the concurrent programming languages with robust formal support, such as the pi-Calculus [MPW92]; practical realizations of such ideas exist in such languages as *Pict* [PT98].

6 Existing Active Network Security Architectures

In the context of the DARPA-sponsored Active Networks project, we have designed a system to control resource access in programmable network infrastructures, called the Secure Active Network Environment (SANE). We have described SANE and its realization in the SwitchWare active network architecture elsewhere [AAKS98].

The ANTS [WGT98] project at MIT is using MD5 checksums of the active code both as a means of referencing it and as a primitive access control mechanism. While this method works in an experimental network setup, a more flexible and distributed mechanism for specifying security policies and naming code is needed for large scale networks. Furthermore, special privileges (or restrictions), as well as packet confidentiality and integrity services have not been defined. The SANE architecture can be readily adapted as a security framework for ANTS.

The more recent Cherubim proposal, from University of Illinois, is using the Java security managers to control the behaviour of active programs, and resembles in may ways the SANE model of active code control.

A number of ideas can be adapted from the Ensemble project at Cornell, which uses multicast communications to establish and maintain state in a distributed system.

Finally, a working group within the Active Networks project has been defining a common security meta-architecture, but it has not become concrete enough for implementation.

6.1 The SwitchWare project

The SwitchWare project is a joint effort of the University of Pennsylvania and Bellcore. The overall project goal is to accelerate network evolution by turning store-and-forward networks into store-*COMPUTE*-and-forward networks, an approach we originated in the Protocol Boosters [FMS$^+$98] project. The goal is to build a network infrastructure which balances flexibility, usability, security and performance; early results are very promising [ASNS97]. The current infrastructure provides a node model where modules can be loaded into the node on-the-fly by the ALIEN active loader [Ale98]. The loader is sufficiently powerful that the extreme case of "capsules" can be supported by treating each packet as a module, although a more typical use of the facility is to add a service used by streams of inactive packets. Among the logical services available at a node is the PLAN system described in Section 4.2.

The ALIEN system is currently implemented in the type-safe Caml language, using the Linux operating system for low-level services such as raw Ethernet sockets. Much of the software is available from the SwitchWare Web site, `http://www.cis.upenn.edu/~switchware`. The ALIEN system restricts access to system services using a namespace restriction scheme called "module thinning", the basic idea being that if the program cannot access certain facilities, it cannot misuse them.

6.2 SANE Details

The Secure Active Network Environment (SANE) [AAKS98] is an architecture providing: a demonstrably minimal set of trust assumptions; the ability to securely bootstrap [AFS97] the remainder of the SANE system when the trust assumptions are met; and authentication and naming services for code that is loaded.

SANE uses the approach of verifying *integrity*[1] of lower layers of a system to provide a guaranteed operating environment for enforcement of restrictions at higher layers of a system. Integrity is a weaker property than correctness, but proofs of correctness can be based on the integrity of verified components. SANE identifies a minimal set of system elements upon which system integrity is dependent and then builds a cryptographically-backed integrity chain, from that set of system elements up to a full running system. This protects the assumptions about behavioral restrictions and correctness of operation that can be used to minimize the cost of per-packet operations.

An illustration of SANE in the context of the overall SwitchWare network element architecture is shown in Figure 3.

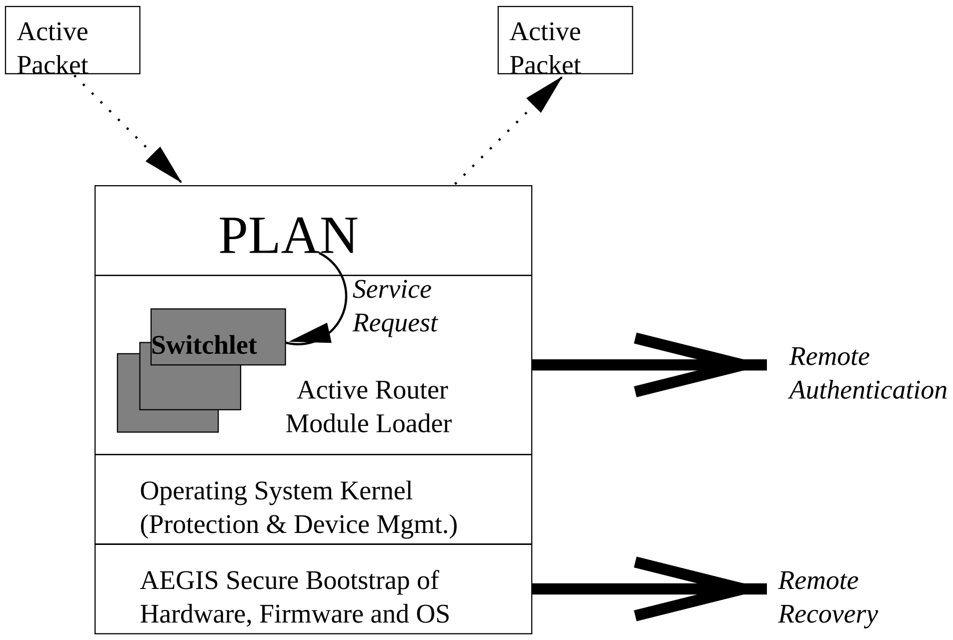
Active Packet
Active Packet
PLAN
Service Request
Switchlet
Active Router Module Loader
Remote Authentication
Operating System Kernel (Protection & Device Mgmt.)
AEGIS Secure Bootstrap of Hardware, Firmware and OS
Remote Recovery

Figure 3: SANE's Relation to SwitchWare

The secure bootstrap system (named AEGIS) assumes that the first 32 KB of the system BIOS is unmodified, that there is a protected key source, and if automated recovery [AKFS98] is desired, that a trusted source exists from which damaged components can be recovered. Under these assumptions, the secure bootstrap process performs, for each system level (*e.g.*, BIOS initialization, Flash ROMS, Boot Block, OS etc.), a cryptographic verification on the level's code before passing control to that level.

[1] We define system *integrity* to mean that the system (hardware and software configuration) is not altered from some known (and presumably correct) state.

If a verification fails, the AEGIS system attempts to recover, and since the secure bootstrap should work on unattended network elements, remote recovery is desirable. If a trusted source is available, it is contacted for a correct copy of the damaged (modified) component. The component is obtained, installed, and the bootstrap process is restarted. The AEGIS system uses a novel technique called "Chained Layered Integrity Checks," (CLIC) and is extensible to any infrastructure which is modifiable.

This latter point should be emphasized, because one of the major changes that hardware manufacturers have made in their systems has been the addition of field-upgradable (and hence modifiable) hardware components. Thus many peripheral cards load "firmware" (which is merely software operating at a lower level of the system architecture than "software" usually runs at!) which defines their behavior. SANE ensures that the expected values of the firmware and other modifiable components of the system are as they should be. We believe that any active network elements based on commercial hardware will take this form; in fact it would be impractical to *not* allow field-upgrades of a network element's components.

SANE provides a remote recovery protocol to cope with integrity check failures. Cryptographic protocols obtain an integrity-checked version of the failed component (*e.g.,* a modified operating system or operating system boot block) from a trusted source. This version replaces or shadows the failed component and the bootstrap process is retried. The remote authentication scheme used in the recovery process is based on the STS Station-to-Station protocol. The same scheme is largely replicated to authenticate packets from remote systems when the active node is in an operational state. SANE uses KeyNote [BFIK98] as a public-key infrastructure and specification mechanism for trust relationships in a distributed system.

6.3 SANE Summary

The SANE elements are combined into a system using the following design principles:

- dynamic checks, performed while the active node is operating, should be as fast as possible, as they are done many times;

- static checks, performed before the active node enters the operating state, can be expensive, as they are done infrequently (typically, once);

- system performance can be improved by tradeoffs which decrease the cost of the dynamic checks at the expense of more costly static checks, or, ideally, by using static checks to eliminate the need for any dynamic checking at all, analogous to "once at compile time" versus "many at run-time" (for example, the computationally expensive digital signatures employed to ensure the integrity of even more heavyweight static checks such as formal verification).

7 Conclusions

Programmable network infrastructures are now appearing, in various forms. The driving force is a need for rapid development of distributed applications requiring customized services from the network. The programmability of a network infrastructure is achieved by exposing some of the control interfaces to the user of the network. The keys to safety and security are the restrictions placed on the programs which use these interfaces. While there is now considerable understanding of how to test whether a program is safe to run on an individual node, the problem of further restricting programs to be network-safe remains open. There is some promising research which attempts to address this issue [JW97].

The Secure Active Network Environment (SANE), used in the SwitchWare active network project, is broadly useful in programmable network environments. It enforces security policy in three ways. First, it integrity-checks all components of an active node, in the steps from power-on to initiation of the module loader, ensuring that the module loader's policy enforcement capabilities have not been subverted. Second, it uses module thinning (a language system imposed restriction) to control access to interfaces. Third, it associates interfaces (*e.g.,* procedure names and object names) with access privileges (through KeyNote certificates), so that authorizations can be extended to remote systems. Thus, SANE can be used to control access to resources throughout the network.

In the context of network-safety, the SANE infrastructure does not check whether a program is network-safe or even node safe. Rather, SANE's role is that of a set of tools and protocols which form a basis for extending trust among groups of programmable systems. It does not solve the harder problem of automating the checking of network safety properties.

The flexibility of programmable network infrastructures is their primary appeal. This flexibility will not be widespread unless the relationship between flexibility and security is clarified, and further, that acceptable tradeoffs between flexibility and security can be discovered and deployed.

References

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

[AFS97] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *IEEE Security and Privacy Conference*, pages 65–71, May 1997.

[AKFS98] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith. Automated recovery in a secure bootstrap process. In *Internet Society 1998 Symposium on Network and Distributed System Security*, pages 155–167, March 1998.

[Ale98] D. Scott Alexander. *ALIEN: A Generalized Computing Model of Active Networks*. PhD thesis, University of Pennsylvania, 1998.

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

[BCR] Inc. Bell Communications Research. *AIN Release 1 Service Logic Program Framework Generic Requirements*.

[BFIK98] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The keynote trust management system. Work in Progress, August 1998.

[FMS$^+$98] D. C. Feldmeier, A. J. McAuley, J. M. Smith, D. Bakin, W. S. Marcus, and T. M. Raleigh. Protocol Boosters. *IEEE JSAC (Special Issue on Protocol Architectures for the 21st Century)*, pages 437–443, April 1998.

[FZ81] R. Forchheimer and J. Zander. Softnet - Packet Radio in Sweden. In *Proceedings, AMRAD Conference*, 1981.

[JW97] A. Jeffrey and I. Wakeman. A Survey of Semantic Techniques for Active Networks, 1997.
`http://www.cogs.susx.ac.uk/users/ianw/papers/an-survey.ps.gz`.

[MBD$^+$83] R. W. Mitze, H. L. Bosco, N. X. DeLessio, R. J. Frank, N. A. Martellotto, W. C. Schwartz, and R. W. Wolfe. 3B20D Processor and DMERT as a Base for Telecommunications Applications. *Bell System Technical Journal*, 62(1):181–190, January 1983.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, pages 1–77, 1992.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM Press, 1997.

[Per92] Radia Perlman. *Interconnections: Bridges and Routers*. Addison-Wesley, 1992.

[PMIM88] Calton Pu, Henry Massalin, John Ioannidis, and Perry Metzger. The Synthesis System. *Computing Systems*, 1(1), 1988.

[PT98] B. C. Pierce and D. N. Turner. *Pict: A Programming Language based on the pi-calculus*. MIT Press, 1998. Technical Report CSCI 476, CS Dept., Indiana University.

[Smi88] Jonathan M. Smith. A Survey of Process Migration Mechanisms. *ACM SIGOPS Operating Systems Review*, pages 28–40, July 1988.

[WGT98] David J. Wetherall, John Guttag, and David L. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols. In *Proceedings, IEEE OpenArch 98*, 1998.