A Secure Active Network Environment Architecture

Realization in SwitchWare

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

"Доверяы, но Проверяы"
"Trust, but Verify"

Abstract—
Active Networks is a network infrastructure which is programmable on a per-user or even per-packet basis. Increasing the flexibility of such network infrastructures invites new security risks. Coping with these security risks represents the most fundamental contribution of Active Network research. The security concerns can be divided into those which affect the network as a whole and those which affect individual elements. It is clear that the element problems must be solved first, as the integrity of network-level solutions will be based on trust of the network elements.

In this paper, we describe the architecture and implementation of a Secure Active Network Environment (SANE), which we believe provides a basis for implementing secure network-level solutions. We guarantee that a node begins operation in a trusted state with the AEGIS secure bootstrap architecture. We guarantee that the system remains in a trusted state by applying dynamic integrity checks in the network element's run time system, using a novel naming system, and applying node-to-node authentication when needed.

I. Introduction

A variety of proposals for programmable network infrastructures are currently extant, such as open signaling [1] and Active Networks [2]. These proposals share the goal of improving network flexibility and functionality through introduction of an accessible programming abstraction, which may be available on a per-user or even a per-packet basis. In the SwitchWare project [3], University of Pennsylvania and Bellcore are collaborating on research into the architecture of Active Network elements.

The goal of programmable network architectures is to provide an acceleration of network service creation. Protocols provide a set of rules by which compliant systems can participate in communications. To build a global virtual infrastructure such as the IP [4] Internet, a "minimal" interoperability requirement was set, namely a packet format and a common addressing scheme. Service enhancements, such as the TCP reliable stream protocol [5], occur at the endpoints of the virtual infrastructure. Since all IP-compliant network infrastructures must support the IP protocol, change of the infrastructure itself is slow and highly constrained. As the Internet has become commercialized, the standardization process has slowed considerably; yet at the same time there is increasing demand for enhanced services.

Active Networks follows the approach first proposed in the "Protocol Boosters" project [6], of enabling on-the-fly modification of network functionality, for example to adapt to changes in link conditions. Protocol Boosting is a design methodology, but Active Networks provides an infrastructure general enough to support any network reprogramming. This is done by raising the level of abstraction of the interoperability layer from a packet format to a programming environment accessible to programmers. Not surprisingly, there are applications for a programmable network infrastructure:

- Provision of value-added services such as non-co-routed paths (to enhance throughput via striping [7] or reliability in the face of link failure).
- Distributed, intelligent, low-latency decision-making and economic algorithms (which can be very scalable) [8], [9], [10] can be employed to solve the network congestion problem.
- Provide loadable diagnostic functionality for network management [11] and distributed monitoring [12].

More applications of Active Networks can be found in [13], [14], [15], [16], [17].

A. Threats

Threats to network infrastructure are intimately tied to the model used for sharing the infrastructure. For example, with the unreliable best-effort model provided by the Internet and the lack of per-hop security properties, security policies are enforced end-to-end.

IP packets are anonymous to the routers, and they, at least before extensions such as multicasting (*e.g.,* MBONE [18]) and RSVP [19], are allocated service on a FIFO basis. IPSEC [20] provides authentication services, but it remains unclear how support for Quality of Service (such as RSVP) will be integrated with authentication services. As it stands, the Internet infrastructure is vulnerable to a variety of denial of service attacks as a consequence of minimal resource accountability, as well as a variety of other attacks such as traffic analysis. We note that since the resource model in the routers is so simple, sophisticated threats are posed by attacks on services implemented at the endpoints, *e.g.,* the notorious "Syn-Ack" (also known as "Syn-flooding") attack [21] on TCP/IP and the "Ping of Death" [22].

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

- Identify the sending network element,
- Identify the sending user,
- Authorize access to appropriate resources based on these identifications,

Scott Alexander, William Arbaugh, and Angelos Keromytis are each working toward a Ph.D. in Computer and Information Science at the University of Pennsylvania.

Jonathan M. Smith is an Associate Professor at the University of Pennsylvania.

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

Old Russian saying.

- Allow execution based on the authorizations and security policy.

The principals involved in the authorization and policy decisions in the security model are users, programmers and administrators and network elements. The network elements are presumed to be under physical control of an administrator. Programmers may not have physical access to the network element, but may possess considerable access rights to resources present in the network elements. Users may have access to basic services (*e.g.,* transport), but only resources that the network elements are willing to export to all users, at an appropriate level of abstraction. Users may also be allowed to introduce their own services, or load those written by others.

In networking terminology, the first three steps comprise a form of admission control, while the final step is a form of policing. A second separation 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.

B. A high-level view of a SANE architecture

Systems are organized as layers to limit complexity. A common layering principle is the use of levels of abstraction to mark layer boundaries. A computer system is organized in a series of abstraction levels, each of which defines a "virtual machine", upon which the higher levels of abstraction are constructed. Each of the virtual machines presupposes that it is operating in an environment where the abstractions of underlying layers can be treated as axiomatic. When these suppositions are true, the system is said to possess *integrity*. Without integrity, no system can be made secure.

Thus, any layered system is only as secure as the foundation upon which it is built. For example, a number of attempts were made in the 1960s and 1970s to produce secure computing systems using a secure operating system environment as a basis [23]. An essential presumption of the security arguments for these designs was that the system layers underpinning the operating system, whether hardware, firmware, or both, were trusted. We find it surprising, given the great attention paid to operating system security [24], [25] that so little attention has been paid to the underpinnings required for secure operation, *e.g.,* a secure bootstrapping phase for these operating systems.

Under the presumption that the hardware comprising the machine (the lowest layer) is valid, the integrity of a layer can be guaranteed *if and only if:* (1) the integrity of the lower layers is checked, and (2) transitions to higher layers occur only after integrity checks on them are complete. The resulting integrity "chain" inductively guarantees system integrity. We call this the Chaining Layered Integrity Checks (CLIC) model. Once the system is operational however, integrity violations may occur through failure of the higher-layer security mechanisms or through malfunctioning software components or other reasons. Preventing or restricting such violations necessitates the presence of higher-layer security mechanisms.

The overall approach to security taken in the SwitchWare project is to provide carefully circumscribed functionality to network programmers, by means of a programming language which allows us to limit functionality and run in a controlled environment. We have implemented a prototype of such a network element, and applied it to the problem of constructing an extended LAN (bridging).

II. SANE Issues and Architecture

In this section we discuss the issues which arise from the threat model we presume. After a discussion of these issues, we further discuss integrity and trust relationships at various levels in the system. Finally, we outline SANE, which addresses the division of integrity checking and enforcement into static and dynamic portions.

A. Separation of Concerns

We make a somewhat artificial, albeit useful, division of our concerns into *static* and *dynamic*. Static concerns are those which can be checked once, or infrequently, as in the case of an Active Network bootstrapping from a cold start into an operational state. Dynamic concerns are those which must be continuously addressed to maintain the operational state of the system.

There are several major advantages to this division that can be used in a system design. First, as static checks are done once, or very few times, they can be very expensive if this pays off in a significant increase in security. Second, dynamic checks can be made faster if it is known that the static checks have been performed in advance. Finally, these divisions usually closely follow the division of a system into layers of abstraction. If the proper trust and integrity relationships are preserved, the operation of the entire system can be trusted.

B. Integrity and Trust

Integrity is a way of saying that a system is what we expect it is; that is, it is unmodified. Trust is a more complex relationship, as something can be unmodified, but not trusted, while if a system is trusted, it must remain unmodified for the trust relationship to hold.

Integrity and Trust relationships in an Active Network setting are of several types. In a layered architecture, each layer in a system trusts the layer below it.

For an Active Network node, a trusted node architecture can be constructed by making the lowest layers of the system trusted, and then ensuring that higher layers depend on the integrity of these lower layers.

There is, however, a significant difference when the actions of the node are programmable and the programs come from hosts or other Active Network elements. In this case, we must construct a web of trust between participating elements. Further, trust is not enough: a downloaded program from a trusted node may be flawed and may damage the receiving node. *Dynamic integrity* checks ensure that the node remains a participating element of the Active Network in spite of such threats.

It is clear then that any architecture for system security in an Active Network must use a combination of static checks and dynamic checks to remain secure.

C. Architecture

The basic layered structure of SANE is shown in Figure 1. Here, we will explain the overall organization of the architecture and its principal goals. The remainder of this article expands on the components of the architecture.

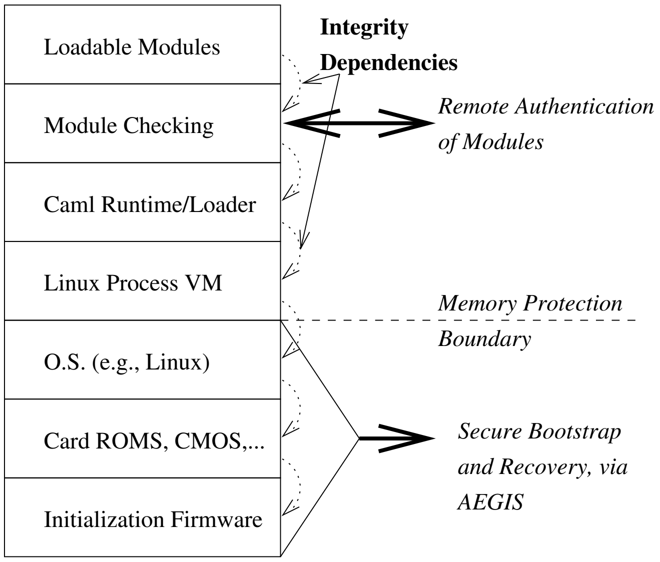

Fig. 1. **SANE Architecture**

The lower layers of the architecture ensure that the system *starts* in an expected state. The design utilizes a secure bootstrap architecture, called AEGIS, to reach the stage where dynamic integrity checks can be applied on a per-user or per-packet basis. AEGIS assumes the integrity of the initialization firmware, and little else (for recovery, a network-accessible trusted source is also required). It then repeatedly, until the Active Network element is operating, checks the integrity of the succeeding layer in the bootstrap before passing control to it. Integrity is checked with a digital signature. This process results in the *expected* operational system starting execution; it makes no guarantees that that system operates correctly. Eventually, we hope to address at least a fraction of operational correctness issues with the application of formal methods.

When the Active Network element is operational, it maintains security in several ways. First, it performs remote authentication when required for node-to-node authentication. Second, it provides a restricted execution environment for the evaluation of switchlets (the programs received from the network). Finally, it uses a novel naming scheme we have developed to partition the node's services name space between users. The authentication and integrity checks performed before a language system begins operating on it, such as checking a digital signature, are static. This is in contrast to dynamic checks performed (*e.g.,* by trying to type-check the packet's code or constrain its execution). These latter checks are performed frequently and thus must be performed efficiently; they guarantee that the network element remains secure, and remains operating.

The issue of where to place restrictions on behavior, the basis of security, can be addressed using a programming language run time system. Memory protection is a common solution in cross-language (and therefore multiple execution environment) systems. Memory protection hardware coupled with privileged instructions can be used to provide a "virtual machine" which operates with a subset of the instructions and addressing capabilities available to programs such as operating systems. The hardware support allows these restrictions to be checked dynamically, *e.g.,* on every memory reference. Use of this approach is necessary where no restrictions are placed on the programming environment, but rather only on the executable machine code form of a program.

A second approach, and the one which we pursue, is to restrict the programmer in the choice of language, and within the context of this language environment, to restrict programs to those which are secure. The potential technical advantage of this approach is that many security properties (*e.g.,* access to regions of memory) can be analyzed at compile time, and thus checked once when compilation takes place rather than dynamically at run time. Thus, this design approach can provide security-based restrictions on program actions, while preserving good performance.

D. *Public Key Infrastructure*

A very important element of our proposed architecture is the public key infrastructure. It is assumed that every user (or group of users) and every Active element owns a public/private key pair, and that these keys (and certificates) are used to authenticate and authorize actions of those entities. For the remainder of this paper, key owners will be referred to as principals.

It is also desirable that the infrastructure allows selective authorization delegation, so that flexible access and resource control policies can be built. Finally, depending on the underlying network fabric, our preferred method to revoke a certificate is by expiration; this minimizes network traffic when authorization checks are performed. In our implementation we intend to use SPKI [26] and PolicyMaker [27].

III. 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 bootstrapping includes [28], [29], [30], [31]. A more extensive review of AEGIS and its differences with the above systems can be found in [32].

A. AEGIS Layered Boot and Recovery Process

We have divided the boot process into several levels to simplify and organize the AEGIS BIOS modifications, as shown in Figure 2. Each increasing level adds functionality to the system, providing correspondingly higher levels of abstraction. The lowest level is Level 0. Level 0 contains the small section of *trusted* software, digital signatures, public key certificates, and recovery code. The integrity of this level is assumed to be valid. We do, however, perform an initial checksum test to identify PROM failures. The first level contains the remainder of the usual BIOS code and the CMOS. The second level contains all of the expansion cards and their associated ROMs, if any. The third level contains the operating system boot block(s). These are resident on the bootable device and are responsible for loading the operating system kernel. The fourth level contains the operating system, and the fifth and final level contains user level programs and any network hosts.

The transition between levels in a traditional boot process is accomplished with a jump or a call instruction without any attempt at verifying the integrity of the next level. AEGIS, on the other hand, uses public key cryptography and cryptographic hashes to protect the transition from each lower level to the next higher one, and its recovery process through a trusted repository ensures the integrity of the next level in the event of failures [33].

The trusted repository can either be an expansion ROM board that contains verified copies of the required software, or it can be another Active node. If the repository is a ROM board, then simple memory copies can repair or shadow failures. In the case of a network host, the detection of an integrity failure causes the system to boot into a recovery kernel contained on the network card ROM. The recovery kernel contacts a "trusted" host through the secure protocol described in Section III-B to recover a signed copy of the failed component. The failed component is then shadowed or repaired, and the system is restarted (warm boot).

B. Recovery Protocol

The protocol we use throughout this paper and in our architecture is based on the Station to Station protocol [34]. The basis of the protocol is the Diffie-Hellman exchange [35] for key establishment, and public key signatures for authentication (to avoid man-in-the-middle attacks). In our architecture we use DSA [36] (a NIST-approved digital signature algorithm), but other (*e.g.,* RSA [37] 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 [33].

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).

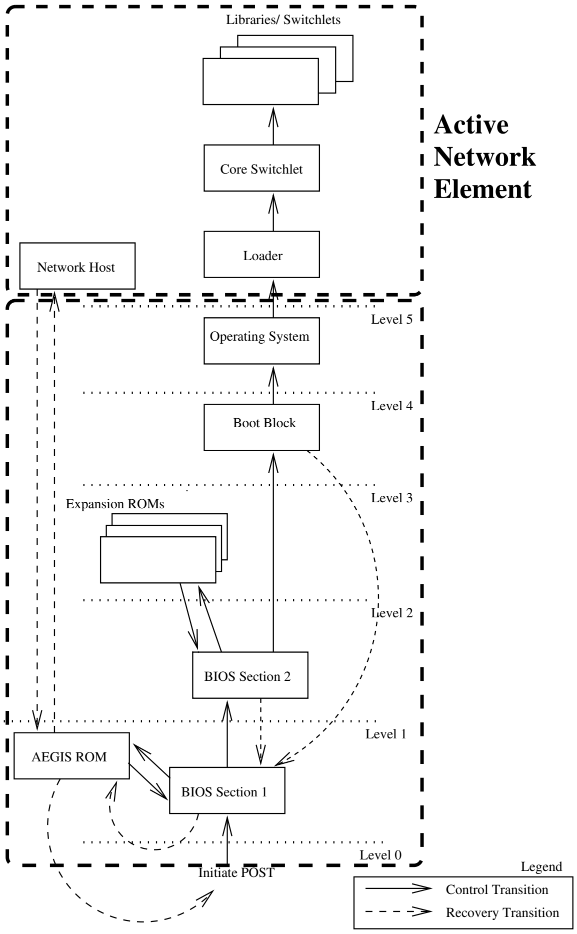

Fig. 2. **AEGIS boot control flow**

IV. BOOTSTRAPPING A SANE NETWORK

Once the node has been brought up in a secure manner, it attempts to establish trust relations with its direct peers. The same protocol that was described in section III-B is used to exchange certificates and establish a shared secret key with each of the peer Active nodes. The certificates exchanged at this stage are used to verify the neighbors, establish administrative domains (and their boundaries) and the trust relations inside and between those domains. The secret key and the trust relations will then be used to:

- Minimize path setup costs, as we will describe in the section VI-A.
- Allow mobile-agent [38], [39] types of applications, where per-hop authentication (and possibly encryption) may be necessary. An API will be defined that lets a programmer make use of these services.
- Secure message exchange between peer Active nodes, such as for routing messages or network management.
- Establish authenticated packet forwarding channels.
- Deter link traffic analysis; the Active node administrator will

then be able to allocate a percentage of the available bandwidth as an encrypted, always-busy, channel. An eavesdropper on the link will then be unable to determine which messages were forwarded to the peer node. Again, an API will be defined that programmers can take advantage of.

V. The Active Network Infrastructure

With the operating system verified and booted, the next step is to make the node part of the Active Network. This is accomplished by loading two final layers. Given our definition of Active Networks, not surprisingly, the lower layer of our network infrastructure is a loader which can load our Active programs. On top of the loader is a Core Switchlet which provides essential services. These two layers are permitted to execute a number of operations that are critical for the operation of the system, and are thus considered privileged. Higher layers can affect the underlying system only through the interfaces presented by the loader and the Core Switchlet. Finally, a non-privileged layer consisting of a set of library routines which provide common services is added. This layering, together with the applications or switchlets, is illustrated in Figure 2.

The lower two layers provide the basis of the dynamic security model in the network infrastructure. They do this by using a strongly-typed language which supports garbage collection and module thinning. Using these techniques, we move from static to dynamic enforcement of our security mechanisms.

A. Why Does the Language Matter?

The programming language defines what operations the programmer can perform. By careful choice of language, we can limit some of the undesirable actions that a programmer might unintentionally or maliciously perform. Thus, through the choice of language, we can prevent certain classes of security violations.

The first property that we desire from the language is strong typing. In a strongly typed language, the only way to convert data from one type to another is through a well-defined conversion routine. Thus, one can typically transform an integer into a floating point value, but cannot perform conversions to or from a pointer type. In a weakly typed language like C, it is this ability to freely convert types which leads to the need for heavier security mechanisms including separation of address spaces between processes.

The second property that we desire is garbage collection. If the programmer is able to manage storage directly, two problems can result. The first is failure to free storage which can lead to loss of performance throughout the system. The second, more dangerous problem, occurs when storage is returned to the allocator and then referenced later. If the storage has been reassigned to another user, it is possible to discover another user's information. Worse yet, if the address is no longer valid, a fault results which must be handled to avoid crashing the entire system. Garbage collection thus allows us to prevent memory violations in a common address-space environment.

The third property that we desire is module thinning. By modules, we mean a set of functions and values which have been combined into a package by the programmer. Module thinning is a technique which allows us to pick and choose which functions and values from a module are available to a switchlet which we load. Module thinning would be equivalent to being able to change which methods are public or private based on the requesting class, in the object oriented world. For example, in the Thread module that we use, there is a function which allows one to kill any process on the system by specifying its process ID. This is inappropriate for switchlets, so we do not make this available except to the loader and the Core Switchlet.

The final property which we require is the ability to dynamically load programs. Clearly, if we intend to run programs that arrive over the net, we must have a way to link those programs into the running system and evaluate them. Dynamic loading gives us this ability. Other systems may use other approaches; for example, an Active node that uses memory protection to isolate processes would probably start a new process and execute the received code in that context.

The Caml programming language [40] provides these features. Caml additionally provides us with a threads interface and static type checking. The former allows a natural programming style and precludes the need to implement a scheduler. The latter pushes many of the costs associated with the type system to compile time. Thus, checks that other systems perform repeatedly at runtime, we perform once at compile time.

We are often asked "Why not Java instead of Caml?" Java has security properties which are appropriate for applets but are not suitable for our purposes. Specifically, we cannot access Ethernet frames from Java without modifying the runtime system. It is also difficult to provide different security models to different users. Since Caml is designed as a general purpose programming language, it allows us to build our own security mechanisms. Despite this, there is nothing inherent in our architecture which limits us to Caml. With appropriate modifications to the Java runtime, Java could support the SANE architecture.

B. 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, 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 also responsible for providing the mechanism by which modules are loaded. Currently, the mechanisms provided are loading from disk or loading byte-code received over the network. The Core Switchlet governs the policy by which this mechanism may be used and may provide interfaces to the mechanism.

C. 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 divided into five modules.

The first module is `Safestd`. This module 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. The rest of the access to Unix functions has been thinned away.

In order to allow the user to supply error or status messages, we have a `Log` module. The user supplies a string which will be saved to a system log. For convenience while debugging, we currently write the messages to a disk file, but for security purposes, we intend to extend this module to limit the amount and frequency of messages produced by any given thread.

Access to the network is provided by the `Unixnet` module. This allows switchlets to access network interfaces for either sending or receiving frames. Currently, only one switchlet is allowed to have access to a given interface. In the near future, we intend to modify this module to receive and demultiplex the data. Access to the data will then be available to any switchlet, assuming said switchlet can prove that it has the authority to access the data as described in section VI-A.

The last of the five modules is `Safethread`. As mentioned, this provides 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 runtime system to check the resources currently consumed and allow or deny the request for additional usage.

D. 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 and implementations of IP and UDP [41].

E. The Active Bridge: An Active Networking Application

To demonstrate the utility of this infrastructure, we have implemented an Active Bridge [42]. This bridge is built from several switchlets which build up layers of functionality. In particular, by loading just the lowest layer, we can demonstrate a buffered repeater. The next switchlet adds a learning algorithm. Finally, the highest layer of the bridge adds spanning tree functionality to give us a nearly IEEE-compliant [43] bridge.

VI. Dynamic Security Checks

Once the Active node is operating, we rely upon dynamic security checks and measures to ensure that the access and resource use policies defined by the administrator are followed. Furthermore, the node needs to provide certain guarantees in regards to service access (essentially, user isolation); some of these guarantees are provided by the underlying operating system and programming language. However, some of our guarantees must be built through additional mechanisms provided by our system.

A. Access Control

One of the basic goals of Active Networks is allowing users to install their own protocols on network elements, in the form of dynamically loaded modules. Since these modules may have access to critical resources, it is imperative that this access be controlled. Furthermore, in some cases it is necessary to authenticate packets belonging to some particular packet sequence, if they need to be handled in some "privileged" manner (*e.g.,* going through a firewall or delivery to some service.) In the next two sections, we extend the mechanism described in section III-B to provide authentication and authorization mechanisms.

These trust relations will be established along a path of Active nodes in most cases. Two possible methods of path establishment are:

- Via direct negotiation with each node, possibly in parallel. The implication here is that the initiator can both identify and communicate directly with these nodes, instead of having to discover the path.
- In a "telescopic" manner, in which a scout packet would identify the next node at each step and initiate the negotiation. In this model, each negotiation has to finish before the next one begins (in order to establish a communications path from the current node back to the initiator).

A.1 Principal Authentication/Authorization

On an Active node, when a principal requests an action (such as use a resource) that is privileged according to local policy, he has to provide credentials that authorize him to perform said action. The protocol that would implement the negotiation is the modified version of the STS protocol, as described in section III-B.

Once the node and the principal have established a security association, they can use it to authenticate (and possibly encrypt) all or some of the messages between them. The node retains all the credentials associated with this exchange, so it can determine whether future attempted actions of the principal are acceptable.

Figure 3 shows the packet format once the security association is established. The authenticator will be included in the packet along with an SPI (the Security Parameters Index is a value used along with the principal and/or node identifiers to indicate the particular security association) and a replay detection counter, similar to the IPsec Authentication Header [44].

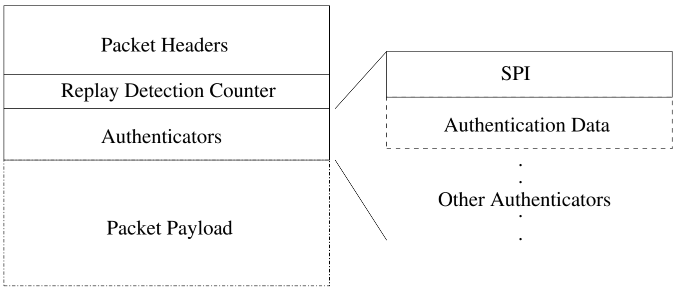

Fig. 3. **Authenticator Header**

However, doing this negotiation with every node along a path to a remote end node is bound to prove costly in two ways:

1. time (both real and CPU cycles spent on the cryptographic operations)
2. more importantly, packet overhead; for every node in the path, there would have to be a different authenticator (since the shared key is different between the principal and each node)

The impact of these problems and their solutions depends on the environment in which the Active node is operating. Based on the types of attacks which must be protected against, we describe a series of measures which may be taken.

The first step is a simple optimization; once the described negotiation has taken place, the principal can then use the shared key to distribute another secret key to all the nodes in the path. By using this common key it is possible to have only one authenticator in the packet, which would be verifiable by all the nodes in the path.

There are two potential problems with this approach. There is still significant computational and path establishment overhead. If connections tend to be reasonably long running, this cost will be amortized.

A worse problem occurs because a (malicious) node in the path can perform actions as if it were the user on some other node, since the key is shared between all the nodes. There are a few workarounds to this problem. In some environments, it may be adequate to accept the problem and to establish paths only through trusted nodes. This is likely to be impossible in other environments.

A second workaround is to distinguish between packet authentication and privileged operations. Authentication can be done using the common key, while privileged operations have to make use of the key known only to the particular node and the principal. This means that control operations will be safe, but that "data" can be forged or modified by a malicious or malfunctioning node. Finally, to avoid delivery of "bad" data to the remote endpoint, the packet would then have a second authenticator in it, which would be only verifiable by the two endpoints (and hence be unforgeable by intermediate nodes). If it is important not to deliver corrupted packets to modules running in intermediate nodes, there is a certain probabilistic scheme that can be used to detect tampering, described in Appendix VI-B.

A last optimization is possible, by taking into consideration the results of section IV. If Active nodes in the same administrative domain have a common set of policies regarding access control and resource utilization, it may be sufficient to go through the negotiation protocol once for each such domain (when entering it), and then having the credentials forwarded as necessary, as shown in Figure 4. This reduces the computational effort and the packet overhead necessary to authenticate/authorize the principal and subsequent packets.

The above optimizations can be applied when the policy specified by the owner of the packet flow and the individual node policies allow them. Access policy to a module (once it has been successfully loaded) is specified by the module itself. The Active node will then enforce this policy.

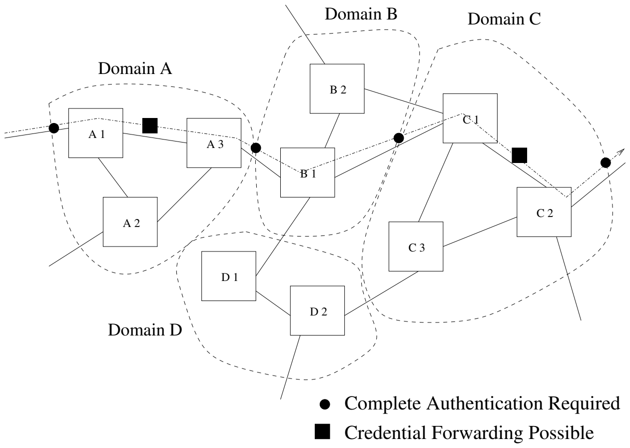

Fig. 4. **Administrative Clouds and Path Setup**

A.2 Single Packet Authentication

For certain classes of applications, the initiating principal may not know exactly which nodes an Active packet will visit (*e.g.,* mobile agent style applications). This means that security association negotiation, as described in the previous section, may not be feasible. However, these programs may need to perform privileged operations on the Active nodes, which means that some form of security guarantees have to be provided. There are a few approaches that can be taken:

- If the administrative domains through which the switchlet will travel are known *a priori*, the initiating principal may establish security associations with nodes in those domains. The established trust relations described in section IV can then be used to forward the credentials inside those domains.
- If the switchlet does not need to perform any privileged operations but requires some security guarantees of its own, it can make use of the existing peer to peer trust relations to do per-hop authentication and/or encryption. For example, if a switchlet requires that each node in its path belong to a list of nodes that it trusts, and since it must trust its creating node, at any hop, it is on a trusted node and can request that that node use its security relations to forward it to another node that is on its list of trusted nodes.
- The switchlet can carry all necessary authorizations the initiating principal believes it may need. These authorizations would be in the form of public key certificates, and the agent needs to be authenticated through a digital signature. While this approach is quite simple, it has two primary drawbacks. It wastes packet space, since all the certificates need to be carried even if they are not used. Further, it is hard to avoid switchlet-replay, unless we assume either network wide (roughly) synchronized clocks or persistent state on the Active nodes (the nodes can then keep track of nonces or agent signatures that have been processed, for as long as the authorizations are valid). Providing these allows a series of potential denial of service attacks.
- When the switchlet needs to perform some privileged operation and needs credentials, it can notify the initiating principal who can then initiate a negotiation to establish a security association. Credentials can then be carried along while inside the same administrative domain. The assumption here is that the

switchlet is able to send the notification message back to the principal, which depends on the both the underlying network infrastructure and the node policies.

B. Detection of Malicious Nodes

In this appendix, we turn to the question of how to ensure that corrupt packets are not delivered to intermediate nodes. We take an approach which allows the originator to make a tradeoff between additional computation and packet space allocated for security headers on one hand and security level on the other. Moreover, we assume that the portion of the packet which might be corrupted should not be changed by any intermediate node. Thus, if any node along the path detects any modification to the immutable part of the packet, it can determine that the packet has been corrupted and can initiate the appropriate recovery procedure. Modifications by outsiders will continue to be detected by the common authenticator shared by all nodes as described in Section VI-A.

Modifications from nodes along the path (who know the secret key) are not so easily detectable. At one extreme (when using only the common authenticator), those modifications are simply undetectable. At the other extreme, including in the packet an authenticator for each node may be too wasteful of resources.

A first approach would be to include a number K of authenticators, where $K < N$ and N is the number of nodes along the path. After checking the common authenticator, a node would check the list of additional authenticators for one addressed to it. If it finds such an authenticator, it verifies that one as well, using the key known only to itself and the initiator. This extra verification step is relatively inexpensive, since it just verifies the common authenticator (as opposed to re-verifying the whole packet). The principal would include authenticators to a randomly chosen set of nodes for every packet.

This approach has two weaknesses. First, a malicious node can simply remove all additional authenticators. Second, such a malicious insider knows which nodes will do the checks, and therefore can modify a packet when the next node in the path is not among those, thus allowing delivery of the corrupted packet to at least one node.

The first problem can be solved by having the principal notify every node, after the path establishment, how many authenticators each packet will include. The second problem is solved by making the authenticators anonymous: after the path is established, the initiator tells each node what its "pseudonym" will be. This pseudonym (a value stored in the SPI field) will be used to associate authenticators to nodes; this way, no node will know which other nodes will do the additional verification on a packet. The initiator also announces to all the nodes the list of valid pseudonyms (but not their bindings) so that a malicious insider cannot substitute an invalid pseudonym for a valid one without detection.

A malicious insider now does not know whether the next node will do the additional verification or not, since it does not know the binding between pseudonyms and nodes. It cannot remove any of the additional authenticators, since every node expects a fixed number of them on each packet. It cannot replace pseudonyms with invalid ones (since every node knows which are the valid pseudonyms) or other valid ones (since the verification would fail).

If the packet includes K additional authenticators and the path has N nodes, the probability of the next node being one that will do the additional verification is K/N in general, or $(K-1)/N$ if one of those authenticators was addressed to this node, assuming a uniform distribution of authenticators among the nodes in the path. The initiator can decide on the value of K by balancing the level of security desired against the acceptable overheads for computation and packet size.

VII. 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.

We present a decentralized way of naming dynamic resources that does not allow name collisions, accidental or malicious. We are making one assumption: in order to load a module on the Active element, the principal must pass some type of authorization check. Furthermore, this authorization is fine grained; each principal is distinguishable for our purposes (although principals can be groups). We believe that this assumption is reasonable, since we expect that an Active element owner will probably want to limit the resources that any principal will potentially consume. (Moreover, we expect that the owner will want to give different access and resource rights to different principals.)

There are then different ways of naming a dynamic resource, each with different semantics:

- The name could be the one-way hash of the code. Assuming certain properties of the hash function, this uniquely identifies the module. The two potential drawbacks to this approach are that different versions of related services have unrelated names and that users have to discover the hash value (either through access to the code or by finding a trusted source that will give the user the hash value).
- The name could be the public key (or its one-way hash) of the module programmer, along with some other identifier assigned by the programmer (such as an ASCII string). The assumption here is that the code may be signed by the programmer (who may be different from the principal who loaded it on the Active element). Version control is possible (subject to the structure of the programmer-assigned identifier). The signature would have to be verified by the Active node before this name becomes "available".
- The name could be the public key (or its one-way hash) of the

principal who loaded the code onto the Active element, along with some other identifier assigned by the principal. Since the principal had to pass an authentication/authorization check before he was allowed to load the code, there is no additional overhead imposed by this scheme.

Different programs may access the same resource through different names, depending on the trust policies of their respective owners. The actual service-access mechanism depends on the node architecture and implementation; we plan to use a portmapper-like approach, but other approaches (*e.g.,* language constructs) are also possible.

Examples of this naming scheme include:

1. $\{P,$ "IPv4/version1"$\}$ — the IPv4 module (version 1) loaded by public key the user with public key P,
2. $\{Q,$ "IPv4/version1"$\}$ — the IPv4 module (version 1) written by the programmer whose public key is Q,
3. $\{H\}$ — the IPv4 module known to the user by its hash value H, or
4. $\{Q,$ "IPv4/version2"|"IPv4/version1"$\}$ — the IPv4 module (version 2) if available, otherwise the previous version of the same module by the same programmer.

This naming scheme allows for any possible name-discovery mechanism. For example "word of mouth," directory-based [45] or DNS-like [46] approaches could all be used, separately or coexisting in the Active Network. The issue is subject to further research.

VIII. SANE Implementation Status and Future Work

The SANE architecture is piece-wise implemented and the integration of the components is now underway. The AEGIS secure bootstrap architecture is currently implemented using a commercial BIOS, and has been tested up to the O.S. kernel level using the FreeBSD UNIX implementation for Intel x86 architecture machines. The AEGIS recovery algorithms are under development, but will draw on an available implementation of the IPSEC protocols for OpenBSD.

The dynamic integrity checking and availability-preservation features of the SwitchWare kernel have been implemented and tested in the prototype Active Bridge. In particular, the Active Bridge demonstrated that the use of functional languages (which are advantageous from a verification perspective) need not impose a severe performance penalty; while full details can be found in Alexander, *et al.* [42], an unoptimized prototype Active Bridge demonstrated Ethernet frame forwarding performance of ca. 1800 frames/second and a bottleneck throughput (tested with `ttcp` between two Alphas running Linux) of about 64 Mbps on 100 Mbps Ethernet connections.

Our current effort is integrating these components in a SANE prototype. The methodology is to use the Utah OS Kit to build a specialized kernel, and then verify this using AEGIS. This project is well underway, and will provide a direct integrity chain between the low-level integrity assumptions and the running dynamic integrity checks.

Extending the work done in SANE, we feel that there are two areas that require further study in our goal for a secure Active Network.

The first area to address is the issue of resource management. While initial Active Network prototyping will focus on best-effort services as a way to obtain operational infrastructure, resource management is essential to many network services such as transport of continuous media traffic. Providing explicit access to the computational and storage capabilities of a node means that there are some very difficult resource co-scheduling problems. An Active Network element must become a multiple resource multiplexer. This opens a variety of new attacks on the network infrastructure including denial of service and new covert channels. We believe that the successful approach will take the form of modern operating systems which control multiplexing at a single system layer, such as the University of Cambridge Nemesis operating system [47] or the University of Arizona Scout/Escort system [48]. These systems allow explicit resource allocation, as well as mechanism for policy enforcement. Putting services in multiplexing-controlled "containers" prevents most overload-based denial-of-service attacks.

The second issue is one of distributed programming. Our threat model has focused on building secure nodes, and providing the infrastructure upon which secure network services can be built. It is a great challenge to build systems which can examine programs, even greatly restricted programs, and decide whether or not they are safe to load. While the halting problem springs to mind, we have a much less difficult problem at the node. Even if we use a language such as the Programming Language for Active Networks (PLAN) [13], some programs must resort to "services" which allow an authorized programmer to perform actions outside of the scope of PLAN itself. It is easy to imagine a well-meaning programmer writing a simple service to read a packet from an input port and write it to two output ports; with such a program a multicast facility might be constructed. If this service was indiscriminately deployed however, packets could be replicated without bound and the network could collapse of overload. This points out the need for systematic global checking and cooperation between nodes, for which we have provided some infrastructure in SANE. The distinction this illustrates is the difference between "node safe" programs and "network safe" programs. We believe that techniques such as the Pi-calculus [49] provide a valuable avenue for exploration.

References

[1] 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.

[2] 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*, pp. 80–86, January 1997.

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

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

[5] Jon Postel, "Transmission control protocol," Internet RFC 793, 1981.

[6] D. C. Feldmeier, A. J McAuley, J. M. Smith, D. S. Bakin, W. S. Marcus, and T. M. Raleigh, "Protocol boosters," *IEEE Journal on Selected Areas in Communications (Special Issue on Protocol Architectures for 21st Century Applications*, vol. 16, no. 3, pp. 437–444, April 1998.

[7] C. Brendan S. Traw and J. M. Smith, "Striping within the network subsystem," *IEEE Network*, pp. 22–32, July/August 1995.

[8] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1280–1297, September 1996.

[9] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta, "Spawn: A distributed computational economy," *IEEE Transactions on Software Engineering*, , no. 2, pp. 103–117, February 1992.
[10] K. Kuwabara, T. Ishida, Y. Nishibe, and T. Suda, "An equilibratory market-based approach for distributed resource allocation and its application to communication network control," *World Scientific*, pp. 53–73, 1996.
[11] C. Partridge and A. Jackson, "Smart packets," Tech. Rep., BBN, 1996, http://www.net-tech.bbn.com-/smtpkts/smtpkts-index.html.
[12] D. J. Farber and J. R. Pickens, "The Overseer: A Powerful Communications Attribute for Debugging and Security in Thin-Wire Connected Control Structures," Tech. Rep. 75, University of California at Urvine, 1975.
[13] "Plan web page," http://www.cis.upenn.edu/~switchware/PLAN/.
[14] 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.
[15] Theodore Faber, "Using active networking to enhance feedback congestion control mechanisms," *IEEE Network Magazine, special issue on Active and Programmable Networks*, 1998.
[16] D. Wetherall, U. Legedza, and J. Guttag, "Introducing new internet services: Why and how," *IEEE Network Magazine, special issue on Active and Programmable Networks*, 1998.
[17] M. Calderon, M. Sedano, A. Azcorra, and C. Alonso, "The support of active networks for fuzzy-tolerant multicast applications," *IEEE Network Magazine, special issue on Active and Programmable Networks*, 1998.
[18] S. E. Deering, "Host extensions for IP multicasting," Internet RFC 1112, 1989.
[19] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation protocol (RSVP) – version 1 functional sepcification," Internet RFC 2208, 1997.
[20] R. Atkinson, "Security architecture for the internet protocol," RFC 1825, August 1995.
[21] L.T. Heberlein and M. Bishop, "Attack Class: Address Spoofing," in *Proceedings of the 19th National Information Systems Security Conference*, October 1996, pp. 371–377.
[22] "Cert advisory ca-96.26: Denial-of-service attack via ping," ftp://info.cert.org/pub/cert_advisories/CA-96.26.ping, October 1996.
[23] M.D. Schroeder, "Engineering a security kernel for MULTICS," in *Fifth Symposium on Operating Systems Principles*, November 1975, pp. 125–132.
[24] F. Mayer M. Branstad, H. Tajalli and D. Dalva, "Access mediation in a message-passing kernel," in *IEEE Conference on Security and Privacy*, 1989, pp. 66–71.
[25] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole, "The operating system kernel as a secure programmable machine," in *Proceedings of the Sixth SIGOPS European Workshop*, September 1994, pp. 62–67.
[26] Carl M. Ellison, Bill Frantz, Ron Rivest, and Brian M. Thomas, "Simple Public Key Certificate," Work in Progress, April 1997.
[27] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management," in *Proc. of the 17th Symposium on Security and Privacy*. 1996, pp. 164–173, IEEE Computer Society Press.
[28] 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.
[29] Bennet Yee, *Using Secure Coprocessors*, Ph.D. thesis, Carnegie Mellon University, 1994.
[30] Paul Christopher Clark, *BITS: A Smartcard Protected Operating System*, Ph.D. thesis, George Washington University, 1994.
[31] Butler Lampson, Martin Abadi, and Michael Burrows, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems*, vol. v10, pp. 265–310, November 1992.
[32] 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*, May 1997, pp. 65–71.
[33] William A. Arbaugh, Angelos D. Keromytis, David J. Farber, and Jonathan M. Smith, "Automated Recovery in a Secure Bootstrap Process," in *To appear in Network and Distributed System Security Symposium*. Internet Society, March 1998, pp. 155–167.
[34] W. Diffie, P.C. van Oorschot, and M.J. Wiener, "Authentication and Authenticated Key Exchanges," *Designs, Codes and Cryptography*, vol. 2, pp. 107–125, 1992.
[35] W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. IT–22, no. 6, pp. 644–654, Nov 1976.
[36] "Digital Signature Standard," Tech. Rep. FIPS-186, U.S. Department of Commerce, May 1994.
[37] RSA Laboratories, *PKCS #1: RSA Encryption Standard*, version 1.5 edition, 1993, November.
[38] Frederick Colville Knabe, *Language Support for Mobile Agents*, Ph.D. thesis, CMU, December 1995.
[39] Günter Kajoth, Danny B. Lange, and Mitsuru Oshima, "A security model for aglets," *IEEE Internet Computing*, vol. 1, no. 4, July - August 1997.
[40] Xavier Leroy, *The Caml Special Light System (Release 1.10)*, INRIA, France, November 1995.
[41] Jon Postel, "User datagram protocol," Internet RFC 768, 1980.
[42] D. S. Alexander, M. Shaw, S. M. Nettles, and J. M Smith, "Active bridging," in *Proc. 1997 ACM SIGCOMM Conference*, 1997.
[43] IEEE, "Media access control (mac) bridges," Tech. Rep. ISO/IEC 10038, ISO/IEC, 1993.
[44] R. Atkinson, "IP authentication header," RFC 1826, August 1995.
[45] C. Huitema, "The X.500 directory services," *cnis*, vol. 16, no. 1-2, pp. 161–166, Sept. 1988.
[46] J. Postel, "Domain name system structure and delegation," Request for Comments (Informational) 1591, Internet Engineering Task Force, Mar. 1994.
[47] 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.
[48] 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," Tech. Rep., Department of Computer Science, University of Arizona, June 1994.
[49] Robin Milner, Joachim Parrow, and David Walker, "A calculus of mobile processes, Parts I and II," *Journal of Information and Computation*, vol. 100, pp. 1–77, Sept. 1992.