A Secure PLAN (Extended Version)

Michael Hicks
Computer Science Department
Cornell University
mhicks@cs.cornell.edu

Angelos D. Keromytis
Computer Science Department
Columbia University in the City of New York
angelos@cs.columbia.edu

Jonathan M. Smith
Computer and Information Science Department
The University of Pennsylvania
jms@central.cis.upenn.edu

Abstract

Active Networks promise greater flexibility than current networks, but threaten safety and security by virtue of their programmability. In this paper, we describe the design and implementation of a security architecture for the active network PLANet [22]. Security is obtained with a two-level architecture that combines a functionally restricted packet language, PLAN [20], with an environment of general-purpose service routines governed by trust management [11]. In particular, we employ a technique which expands or contracts a packet's service environment based on its level of privilege, termed namespace-based security. *As an application of our security architecture, we present the design and implementation of an active-network firewall. We find that the addition of the firewall imposes an approximately 34% latency overhead and as little as a 6.7% space overhead to incoming packets.*

1 Introduction

Active Networks [43] offer the ability to program the network on a per-router, per-user, or even per-packet basis. Unfortunately, this added programmability compromises the security of the system by allowing a wider range of potential attacks. Any feasible Active Network architecture therefore requires strong security guarantees. We would like these guarantees to come at the lowest possible price to the flexibility, performance, and usability of the system.

At the University of Pennsylvania, we have developed an Active Internetwork called PLANet [22]. PLANet's node architecture consists of two levels: the *packet level* and the *service level*. All programs at the packet level reside in the messages, or packets, that are sent between the nodes of the system. These programs are written in PLAN, the Packet Language for Active Networks [20]. Packet programs are simple by nature, and serve to 'glue' together service level programs, just as a shell-script glues together calls to more complicated programs. In contrast, service level programs (or *service routines*), reside at each node and are invoked by PLAN programs evaluating there. Service routines are general-purpose and may be dynamically loaded across the network [2]. This general architecture is shared by many so-called active packet systems, including ANTS [47, 46] (where its 'API' is analogous to PLAN service routines, see the companion paper in this volume for a direct comparison [24]), SNAP [36], PAN [41], *etc.*

A central goal of PLANet is to provide Internet-like service as a baseline, augmented by its active capabilities. The Internet allows any user with a network connection to have some basic services. In addition to basic packet delivery provided by IP, basic information services like DNS, `finger`, and `whois`, and protocols like HTTP, FTP, SMTP, and so forth are provided. Similarly, a goal of PLANet is to allow any user of the network to have access to basic services; these services should naturally include some 'activeness.' This goal implies that some functionality, like packet delivery in the current Internet, should not mandate authorization. There is a pragmatic reason to make the same choice: the converse assumption, in which *all packets* require proper authorization before they can be executed, can be extremely costly. This is because authorization requires *authentication*: each packet must be associated with a principal that is relevant to the authorization policy. Packet-level authentication uses cryptography to ensure that a packet's identity is not spoofed, and cryptographic operations, particularly public-key operations, can be quite expensive relative to normal packet processing. For example,

adding a 30% overhead to packet processing (based on measurements of software-based cryptography that we report at the end of the paper) on each node would severely degrade the performance of the network.

PLANet was designed so that the programs at the packet level are the lowest common denominator with respect to security. That is, all packet programs by themselves (without calls to service routines) are safe by definition thanks to the formal properties of our packet language, PLAN. This is the same model as in the IP Internet—all IP packets are acceptable by default and need not be authorized inside the network. Security, therefore, boils down to the services: in particular, a packet remains safe as long as it only makes calls to service routines that are themselves safe; therefore, we must ask the question "which services can be considered safe?" While for some services the answer is clear (for example, determining the address of the current node is safe), service safety is ultimately a matter of local policy. For example, a router in the center of the network may allow very few service routines, while an end-host might provide a more liberal execution environment. Moreover, a service's safety in general is likely not absolute: calling it might be acceptable for some packets but not for others. For example, a properly authorized network management packet should be allowed to update a node's routing table, while an untrusted packet should not.

This paper presents the design and implementation of the security architecture in PLANet. In particular, we focus on the task of building a secure service infrastructure based on the foundation of a safe packet language, in this case PLAN. While here we focus on our experience with PLANet, we expect that our approach will apply to any active network infrastructure that uses a safe packet language combined with more general-purpose services.

We begin by presenting a description of our architecture, after describing the attacks it protects against. We then follow up with a description of the implementation of this architecture in PLANet. After a brief discussion of PLAN and its relevant characteristics, we present possible methods of security management and the one we have chosen to implement: *namespace-based security*. We describe how we enable authentication, and manage relevant security information, such as which service routines are available to which principals, using QCM (Query Certificate Manager [17]). We then demonstrate how we have used our system to implement a simple firewall that 'filters' active packets. Finally, we present some related work and conclude.

2 Overview of Secure PLAN

To evaluate the effectiveness of any security system, we must consider the threats it defends against. Therefore, we begin by describing the behaviors that threaten an active network, and then describe our two-level security architecture designed to secure against them.

2.1 Threat Model

The two major threats to any active networking system are to the *public resources* of the system: the CPU, memory, and network; and to the *contents* of the system: the packets themselves and the information stored on routers. These threats imply two forms of attack:

- **Denial-of-Service**. Because of the greater expressibility of active network programs (compared to protocol packet headers), there is greater potential for the misuse of the system's public resources, thus denying service to other programs. For general programs, the public resources should be fairly apportioned, while those with more privilege could gain additional latitude.

- **Protection**. Programs should be protected from interference by other programs. In particular, one program should not be able to read or write data private to another program without authorization, either while the packet program is in transit or when it is running (*i.e.*, no packet or program snooping). This property implies program isolation.

In responding to these attacks with a security system, there may be attacks on the security system itself. As mentioned, we would like to allot greater privilege to some packets, such as those associated with a node's administrator. Therefore, it is important that these packets be properly authenticated, and that no impersonation or *spoofing* attacks be possible. Similarly, the authentication and authorization mechanisms should also be robust against *replay* attacks, in which valid, but old messages are replayed in an attempt to gain illegal access.

2.2 Architecture

As already described in Section 1, we partition the problem of defending against these attacks into the packet level and the service level, using different mechanisms at each level. At the packet level, security is obtained via *functional restriction*: the limited nature of the PLAN language prevents attacks from being formulated, particularly denial-of-service and protection attacks. We elaborate on the reasons for this claim in the next section.

At the service level, we make use of an authorization system to govern access to services. While some services may be considered usable by all (we call these the 'core' services), many services that are necessary for the operation of the active node should not be made available to all

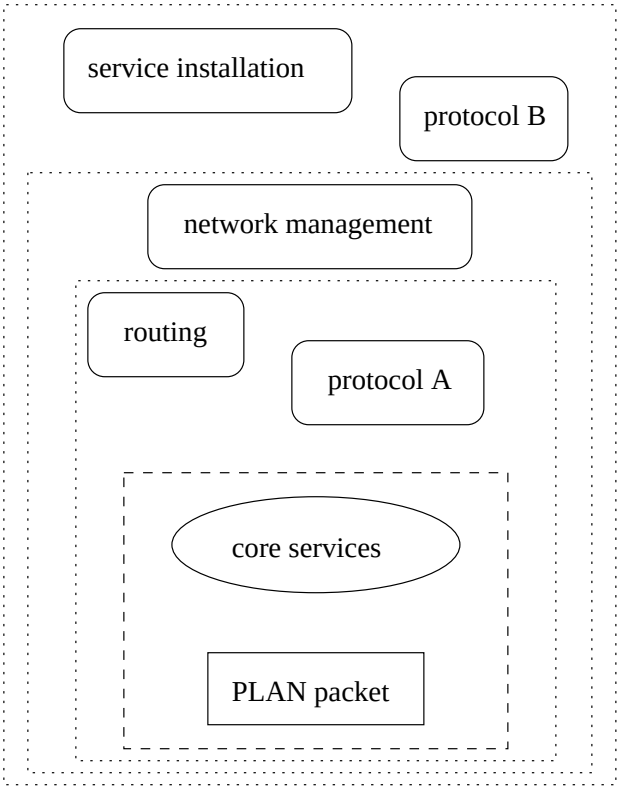

Figure 1. PLANet's security architecture. The contents of the dashed box are available to all incoming packets, while the dotted boxes encapsulate service packages available only to select users. Services may be further restricted by what parameters they can be called with.

packets; an example would be network management functions. Our architecture associates with each principal[1] a set of service routines and policies that are allowed at his level of privilege. The policies are enforced and the routines are made available after the user has been successfully authorized. This architecture is illustrated in Figure 1.

This scheme provides access control for system services. However, once access to these resources is obtained, finer-grained management may be required. For example, more than just say that a packet may or may not have access to a service, we might say that a service is accessible but only when called with certain parameters. We flesh out the details of this architecture in the next two sections. We describe PLAN's security properties in the next section, and then present our service management methodology.

[1] A principal may be a network node or a user. Each principal holds a public/private key pair, and is identified (at least for security purposes) by their public key.

3 Packet Security via PLAN

PLAN [20] is a small functional language resembling ML [28, 32]. It differs most importantly from other functional languages in that it includes a primitive `OnRemote` (among others) for evaluating an expression at a remote node. Invoking `OnRemote` will result in a newly spawned packet. PLAN was designed as the foundation of PLANet's security, with the intention that all PLAN programs can be considered safe. PLAN's security properties are described below.

PLAN's Security Properties

PLAN was designed so that all PLAN programs by their nature are impervious to the attacks we described above. That is, PLAN programs (which do not call service routines, or only call 'safe' ones) should not be able to mount denial-of-service attacks nor should they be able to interfere with other packets or node-resident code and/or data. This is achieved in three ways:

- **Limited expressibility**. PLAN is not a general-purpose language, but is resource- and expression-limited in order to prevent CPU and memory denial-of-service attacks. In particular, all PLAN programs are guaranteed to terminate[2], since PLAN does not provide a means to express non-fixed-length iteration or recursion. In addition, PLAN does not provide means for its programs to directly communicate, meaning that one program cannot directly access or affect another (communication is possible indirectly through services).

- **Strong Typing**. In weakly-typed languages, like C, security restrictions can be overcome by, for example, using unsafe casts to change integers into pointers, or exploiting unchecked array accesses to force buffer overflows. PLAN prevents such attacks by enforcing strong typing, as is done in languages like Java [16]. This idea has become common in recent years (*cf.* [9, 12, 18, 30, 35]).

- **Packet Counting**. While PLAN's language restrictions can bound CPU and memory resource usage on a single node, they are not sufficient in restricting use of *network* resources. For this purpose, PLAN packets have a *resource bound* counter which is decremented each time a packet is sent (*e.g.*, the same packet forwarding itself, or a new packet being spawned). Therefore, the number of hops that a PLAN program and any of its progeny may take is limited by the initial value

[2] PLAN programs terminate as long as the services called also terminate.

of this counter. This mimics the functionality of the IP "Time To Live" (TTL) field.

Of these three mechanisms, the latter two have come into common usage in packet-based active network schemes, and mobile code in general, while the first technique is less appreciated. Most active network systems we are aware of assume that the use of a general-purpose, type-safe language combined with resource counters is sufficient; misbehaving threads are simply killed when they exceed their resource limits. However, recent studies have shown that such an approach is both potentially unsound [8, 18], and quite costly [19]. In particular, without careful engineering, abruptly terminating a packet may leave the system in an inconsistent state, since packets may be manipulating shared resources when they are killed. This problem led Sun to deprecate the `Thread.kill` routine present in early versions of Java.

That said, all resource bounding strategies have their limitations. For example, it has been shown that guaranteed termination is not really a strong enough property [20, 36, 23], and that a tighter per-packet bound is needed. The following following property applies to IP packets, and could well be considered for active packets:

> *The amounts of bandwidth, memory, and CPU cycles that a single packet can cause to be consumed should be linearly related to the initial size of the packet and to some resource bound(s) initially present in the packet.*

As it turns out, PLAN meets this property as well, with some suitable restrictions on function calls and iterators (see [20] for details). More recently, a follow-on to PLAN called SNAP [36] has been proposed, whose programs meet this property outright. Indeed, the security architecture that we propose here will work just as well with SNAP or with any other packet language that prevents the attacks that we have described above. However, while we feel that language-based support for achieving resource bounds is a promising approach, there is still much work to be done.

As we have described it, the safety of a packet program is predicated on the safety of the services it calls. If a service allows a program to, for example, perform unbounded iteration, then denial-of-service attacks can be launched. For this reason, it is of critical importance that a system for managing the services be in place. We discuss our approach, among others, of using trust management to manage namespaces in the following section.

4 Service Security via Trust Management

Because of their general-purpose nature, service routines may perform actions which, if exploited, could be used to mount an attack. A radical solution to this problem would be to prevent *any* service routine from being installed that could potentially harm the node in the ways described in Section 2.1. However, this solution would rule out many useful service routines. Instead, we wish to allow the inclusion of potentially harmful service routines—for example, network management operations—that should only be made available to certain, *trusted* users.

4.1 Trust Management

Given our loose goal of allowing only trusted programs to use potentially unsafe services, it follows we must define a policy that relates trusted programs to unsafe service routines and a means to enforce this policy. Moreover, we can expand on this observation to arrive at the following requirements for our setting:

- Security policies:
  - Policies should be *modifiable* as needed, by the proper administrative entities, while the system is operating. This is particularly important for active networks, as both new users and new services that should be governed by the security policy will appear over time.
  - Policy abstractions should be *flexible* so as to address current as well as future application needs. Again, this requirement derives from the inherent dynamicism of an active network, both in terms of users and services.
- Enforcement mechanisms:
  - To minimize the size of our *trusted computing base*, enforcement mechanisms should be simple to understand and employ. That is, in general, trustworthiness decreases with complexity, since the likelihood of both implementation and user error is higher.
  - It should be possible to implement enforcement mechanisms without relying on the existence of a widely-available infrastructure. That is, each node should be able to make decisions locally, based on its own policy and/or credentials that a user program might present.
  - Security mechanisms must *scale* to support increasing numbers of different applications, users, administrative entities, and their trust relations. Note that the previous requirement for decentralization should improve scalability.

In general, many of these requirements can be met by employing a *trust management system* [11]. In a trust management system, each user, or *principal*, is assigned some level

of privilege (or trust). Based on this trust level, the principal is permitted to perform certain actions, and may potentially delegate those actions to other principals. The novelty of the approach is that trust relationships are managed independently of the particular actions that an application might perform. Instead, the relationships between principals and the actions they may perform are specified in a separate policy, expressed in a special policy language. On each action that requires authorization, the program can invoke the trust management system to determine if the action is authorized for the principal in question. If so, the program can invoke corresponding action, perhaps with some additional parameters provided by the trust management system in response to the query.

Typical trust management systems provide means for updating local policies, for distributing policies across the network, and for using cryptographically-sealed credentials to assert trust relationships. In particular, cryptography is used to authenticate the principal associated with a message before the local policy is checked for that principal.

Applying a trust management system to PLANet is reasonably straightforward. Each PLANet node uses a policy manager from the trust management system to manage its local policy. When a running PLAN program wishes to invoke a privileged service routine or alter the node's state, the principal associated with the packet is authenticated, and the operation is checked against the appropriate policy by the policy manager. If either step fails, the operation is denied. The interesting questions are how to choose policies that admit useful services to the widest number of principals, and how to ensure scalability and good performance through the choice of enforcement mechanisms. We consider the question of policy and mechanism for authorization below; details about our particular implementation of authentication and authorization are presented in the next section.

4.2 Policy

To start, we must consider what kind of policies we would like to express. As mentioned, we essentially want to encode our policy as a mapping between principals and services. Conceptually, each principal has associated with it a list of services that it can access, *i.e.*, a per-principal access control list (ACL). Furthermore, we want to refine this mapping to specify not only *whether* a service routine may be invoked, but *how* it may be used. For example, a *soft state* service which allows packets to leave temporary state on the routers might apportion different amounts of space to different principals. We call such per-principal differences in service evaluation *policy-based parameterization*. In general, because different services will have different usage policies, we permit services to define service-specific policies based on generic service parameters; we present more detail on policy-based parameterization in Section 5.2. Finally, we would like to manage delegation policies with regard to these mappings. For example, we might specify that the services in set s may be accessed not only by principal p, but also by those principals authorized by p.

Encoded naively, a per-principal ACL would not scale as the number of services and principals grows large. To improve scalability, we change our specification of the ACL in two ways. First, we assume a set of core services on the node. The ACL then indicates what services, above the core services, are available to certain principals. We also find it convenient to indicate which services should be *subtracted* from the default environment for a particular principal; this will be motivated in Section 6. Second, rather than map individual principals to lists of services, we define sets of principals and sets of services, and indicate mappings between them. This idea is similar to the use of group permissions in the Unix filesystem: rather than store a list of user id's with each i-node, a single group id is stored instead, which indirectly refers to a set of user id's.

By using a suitably expressive trust management infrastructure, we should be able to encode this set-based policy, and then rely on the trust management infrastructure to provide delegation, admit the possibility of updating the policy, and to administer it in a distributed, decentralized manner. We describe the trust management system we use in our implementation, the Query Certificate Manager (QCM), and the way that we formulate our policies in Section 5.2.

Beyond this simple policy, we would like to be able to specify general resource usage parameters, such as CPU and memory use. While we do not consider such parameters in this paper, they have been considered in work we have done elsewhere. In particular, we have found that such resource-based policies can be achieved with assistance from lower-level system software, as in the SQoSH [3] and RCANE [5] systems, which share a software base used to implement many PLAN services. SQoSH used trust management techniques to control a virtual-clock based bandwidth allocation system, and RCANE used trust management techniques to control a more general resource multiplexing scheme. The scheme was implemented both by changes to language runtimes (unnecessary with appropriate use of our scheme) and by use of a node operating system, Nemesis [29], to provide resource guarantees.

4.3 Mechanism

While the policy manager will handle the issues relating to policy and trust management, we must still decide how to use it most effectively. In particular, we must decide when authentication and authorization will take place, so as to maximize flexibility and performance.

There is a space of possible decisions, bounded roughly by the following two approaches:

1. *Perform policy checks at each service-routine invocation.* Each time a service routine is called from PLAN, a check is made to see if the 'current principal' is allowed to access the service. If this is the first such check, then the principal must be authenticated. If either the authentication or authorization check fails, an exception is raised. In effect, we are proposing a more elaborate variation of the Unix system-call mechanism.

   The benefit of this approach is its flexibility. In particular, policies can take advantage of dynamic information, such as the values of arguments to the service functions. The drawback is that *all* service calls are subject to a runtime check *at each invocation*. This is because the set of services subject to policy, and the policies themselves, might change over time. Therefore, service routines in general need a 'hook' for checking the most recent policy. We can mitigate some of this cost by limiting the routines that might be subject to policy. This might be applicable to the set of standard, core services, or to services that do not require policy-based parameterization.

2. *Perform all checks once-and-for-all, before the packet executes.* That is, all service calls in the packet are authorized before the packet is allowed to execute. The advantage of this approach is that once authorized, the packet can run without dynamic checks. On the other hand, there are two drawbacks. First, policies based on information that is not known at the time of the early check are precluded, reducing flexibility. Second, the static check must consider all possible execution paths, even ones that may not be executed. As a result, one static check could be more costly than a series of dynamic ones.

We employ the middle ground of these two approaches, using two mechanisms. First, before it wishes to access a privileged service, a packet authenticates itself with the node. At this time, the policy is checked, and those services that the packet is authorized (unauthorized) to invoke are added to (subtracted from) the packet's current service symbol table (which at the outset of execution contains just the core services). From then on, if a packet attempts to invoke a service for which it is not authorized, that service will not be in the symbol table and thus access will be denied. Since PLAN is strongly typed and its interpreter looks up services on an as-needed basis, programs are incapable of invoking code outside of this updated table. We call this approach *namespace-based security*.

Second, we allow those services which may require policy-based parameterization to query the policy manager as necessary during their execution. For example, the soft state service mentioned above would query the local policy on each attempt to store new soft state, thereby determining whether the current principal was allowed to allocate additional storage.

There are a number of advantages to our approach. First, only those packets that use privileged (non-core) services must pay for authentication and authorization; unauthenticated programs may run without any performance penalty. This mimics the model of the Internet, which allows normal packets to flow without authentication, while specialized packets, like router control protocol messages and network management messages, need to be authenticated. Second, privileged services that only appear in the policy as access/deny (*i.e.*, they are not subject to policy-based parameterization), do not require a per-invocation check. Finally, services whose usage depends on dynamic information (*i.e.*, the arguments of the invocation, or some other system state) can specify their own policies and invoke the policy manager as needed.

There is more that could be done in our current system. As we have described them, policies only apply to PLAN service routine calls, not calls between service routines. However, this functionality can be added, as we demonstrated in work on a related system [6]. Here we used the service language's support for implementing namespace-based security, called *module thinning*, to support our policies. The use of module thinning has been explored for active networks in ALIEN [1] and for mobile agent systems in Safe-Tcl [30].

5 Implementation

In this section, we describe the mechanisms used by PLAN programs for authentication and authorization. A thorough description of our implementation is found in the PLAN documentation [25].

5.1 Authentication

Before a PLAN program may invoke a trusted service, its associated principal must be determined; this is the process of authentication. Authentication is typically done in a public-key setting by verifying a digital signature in the context of some communication (*e.g.*, a packet). In PLAN, one obvious link between communication and authentication is the *chunk*.

A chunk (or **c**ode **hunk**) may be thought of as a function that is waiting to be applied. In PLAN, chunks are first-class—they may be manipulated as data—and consist internally of some PLAN code, a function name, and a list

of values to be used as arguments during the application. A chunk is typically used as an argument to `OnRemote` to specify some code to evaluate remotely. A chunk may also be evaluated locally by passing it to the `eval` service, which resolves the function name with the current environment, performs the application, and returns the result.

We have added an additional service called `authEval` which takes as arguments a chunk, a digital signature, and a public key. `authEval` verifies the signature against the binary representation of the chunk. If successful, the chunk is evaluated; otherwise, an exception is raised. The authenticated principal is associated with its chunk during evaluation. Because our PLAN interpreter evaluates each packet in its own thread, this can be done by associating the principal with that thread's identifier. Services can determine the 'current principal,' perhaps to query a service-specific policy, by checking this mapping. Because a caller's thread identifier cannot be forged, and because the authentication service is itself a separate service, this provides a safe way to track a principal without worry that some malicious service will change the associated principal after the authentication phase.

There are two key advantages to this approach. One is that a principal signs exactly the piece of code it wants to execute, and may only have extra privilege while executing that piece of code. Secondly, only those programs which require authorization will have the extra time and space overheads.

But the approach has three problems. The first is that the authentication performed here is *one-way authentication*. While the node authenticates the principal, the principal never authenticates the node. This could be a problem if a program is diverted from its intended destination and invoked on a different node. The second problem is that there is nothing guarding against replay attacks. Finally, public key operations are notoriously slow.

To address these problems, we make use of the protocol we defined in SANE (Secure Active Network Environment) [4, 5]. This protocol allows a principal and a node to authenticate each other and generate a shared secret and an identifier for that secret (named the Security Parameters Index, or "SPI"). The protocol is essentially a variation of the Station-to-Station protocol [14]; the reader is referred to [4, 7] for more details. Our PLAN implementation of this protocol is described in more detail in the PLAN documentation [25].

Once the protocol is completed, parties can use the shared secret to authenticate via HMAC-SHA1 [27], in a way similar to that used in the IPsec [26] protocols. To prevent replay, each principal associates a monotonically increasing counter with the shared secret, also included in every transmitted message. To deal with out-of-order delivery, we use a sliding-window scheme, again similar to the scheme used in IPsec. The additional state required is minimal: an integer keeping track of the largest sequence number received, and a 64-bit mask showing which of the previous 64 packets have been received (the window size is configurable; our choice of 64 as the default value was based on IPsec). We reflect the use of HMAC-SHA1 in PLAN by altering the signature of `authEval` to take a chunk and a tuple consisting of the SPI, the counter, and the HMAC signature over all of the previously mentioned items.

5.2 Authorization

As our policy manager, we have chosen to use the Query Certificate Manager (QCM) [17], which provides comprehensive security credential location and retrieval services for set-based policies. While in this paper we are making use of QCM, our architecture is designed so that other policy managers can be used instead. In particular, we have used the KeyNote [10] trust-management system in related work [6].

5.2.1 Namespace control policies

Following our general policy requirements discussed in Section 4.2, our QCM namespace control policy specifies an ACL in terms of the services to be added to or subtracted from the default service-environment (*i.e.*, the core services) by associating certain *thicken* and *thin* sets of services with a principal or set of principals. Once a principal has been authenticated, QCM is queried to discover the *thicken* and *thin* sets, which are then used to add or subtract services from the service symbol table maintained by the PLAN interpreter; this modified symbol table is used for the duration of the authenticated chunk's evaluation. As an optimization, we cache the modified table for future reference, thus avoiding repeated invocations of QCM and reconstructions of the table as long as the policy has not changed.

The following is an example QCM ACL that considers two principals, p_1 and p_2:

```
p1 = <p1's public key>;
p1_thicken = {"print"};

p2 = <p2's public key>;
p2_thicken = {"thisHost"};

acl = {
 ( p1, p1_svcs, {} ),
 ( p2, union ( p2_svcs, p1_svcs ), {})
};
```

In addition to identifying the keys of p_1 and p_2, we define two sets, `p1_thicken` and `p2_thicken`, which are used

to specify the thicken sets of those principals in the ACL. The ACL itself is defined by the variable `acl`, which is a set of three-tuples. The first tuple indicates p_1's environment should be thickened following authentication by `p1_svcs`, while the second says that p_2's environment should be thickened by both `p1_svcs` and `p2_svcs`. In both cases, the thin sets are empty, specified by $\{\}$. Note that in this case, the first element of the three-tuple is an individual principal; more generally, it can be a set of principals.

5.2.2 Policy-based Parameterization

In addition to specifying namespace-based policies, we can specify per-service policies to be used by the services themselves, allowing policy-based service parameterization. Such policies are specified as a set identified by the service's name, whose elements are two-tuples that contain:

1. a principal or set of principals (as in the ACL)
2. a labeled record of length 1, with the label corresponding to a service-dependent parameter name (where multiple parameters per service are reflected as multiple records).

As an example, consider the PLAN `resident` state package which provides user-defined soft state. The resident state policy specifies how much state particular principals are allowed to keep. For example:

```
def = <default user's key>;
resident = { ( def, <amount=100> ),
             ( p1, <amount=1000> ) };
```

This policy indicates that default users (which are automatically given the `def` key) are allowed to have at most 100 words of information stored on the node at any given time,[3] while principal p_1 may store up to 1000 words of information. This policy is enforced in the resident state implementation itself by calling QCM on each store attempt.

5.2.3 Distributed Policies

Though we have not shown it so far, a key advantage of using QCM is that it provides linguistic support for specifying distributed policies. Moreover, sets described in a distributed manner impose no additional query complexity. For example, a node A may define a set `l` in terms of a set `m` which resides at another node B:

`l = {` p_1`,` p_2`, ... ,` p_n `} union` B`$m;`

[3]Note that because all unauthenticated principals share the `def` key, this means that those principals can do little damage to the node, but can deny service to other unauthenticated principals.

If the authorization service on A makes a membership test on set `l`, QCM will automatically query B if necessary. The version of QCM that we use in PLAN actually makes use of PLAN packets to perform its communications. These packets query the QCM service on remote nodes on behalf of the QCM service of the querying node. Interestingly, the QCM service can itself be privileged (and thus subject to policy) as long as there are no cycles in the policy specification of the thicken and thin sets. If this were not the case, QCM would fall into into a distributed, infinite loop.

One way to short-circuit remote queries in QCM is to use *certificates*, which are signed assertions about set relationships. Certificates may be passed as additional arguments to `authEval`, or may be obtained during node-node authentication. This allows QCM to implement both *push*- and *pull*-based information-retrieval.

An avenue of future work is to determine how to best update the QCM policy for each node as the policy changes. For example, we could augment local policy when certificates are provided by authenticating programs. We could also allow local policies to refer to a global policy that resides on another node in the local administrative domain. Thus, when this node's policy changes, those changes are reflected in all of the policies that refer to it.

6 A Simple Active Firewall

As a proof-of-concept of our security architecture, we have designed and implemented an *active firewall* using PLANet. In today's Internet, firewalls are used to prevent the entry of potentially harmful packets arriving from an outside, untrusted network. This is visualized in Figure 2. When packets can be active, this simple approach is too limiting. In this section, we describe how we adapt the traditional notion of a firewall to work in an active setting.

6.1 Implementation

Firewalls typically filter certain types of packets, such as all TCP connection requests on certain port numbers. Usually such packets are easily identified by their protocol headers. In PLANet, and indeed in any active-packet system, there is no quick way to determine a packet's functionality without delving into its contents, which would be a significant performance bottleneck. Therefore, we need an alternate way of filtering out those packets which may be potentially harmful.

Our approach is that rather than filter packets at the firewall, we associate with them a *thinned* service environment in which any potentially harmful services are removed. The packets may then be evaluated inside the trusted network using only those services. While this may seem to contradict

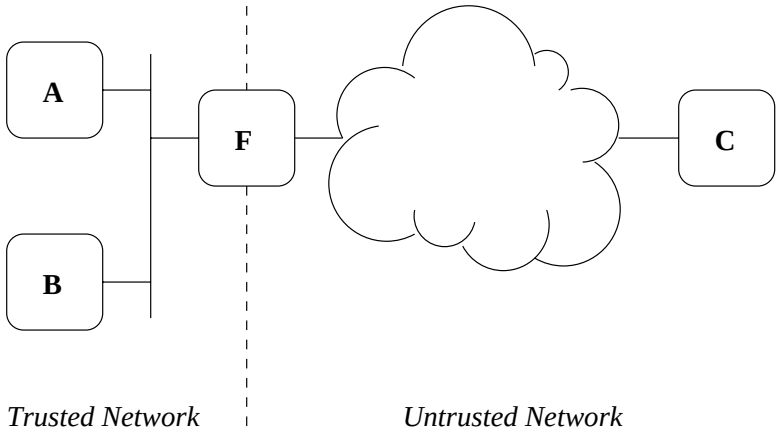

Figure 2. A trusted network behind a firewall.

our premise, stated in Section 2.2, that the default environment should consist only of 'safe' services, in the context of a trusted Intranet we would expect that the default privilege allowed to local packets exceeds that of foreign packets. Furthermore, we would not want to impose the overhead of authentication and authorization on local packets in the general case.

To thin the environment of foreign packets, our firewall associates them with a *guest* identity that has the appropriate policy. To do this, the firewall F wraps the packet's chunk `c` as follows:

```
fun wrapper(c, sign) =
  (zeroRB(); authEval(c,sign))
```

This wrapper first exhausts the packet's resource bound by calling the service `zeroRB`, thus preventing it from sending any additional packets. It then evaluates the packet's chunk c using the guest identity, as indicated by the signature, for the duration of the evaluation. This means that if c attempts to call any services that have been thinned, the call will fail.

This scheme implies that the firewall signs each packet, using the guest's identity, and provides the signature to `authEval`. In order to make this process as fast as possible, the firewall would authenticate with hosts A and B ahead of time using the guest key.

However, because the guest environment will provide less privilege than the default environment, we should be able to avoid the cryptographic cost: any authenticating principal whose environment is thinned and not thickened can be 'taken at its word.' We could extend our framework to allow `authEval` to take a public key rather than a signature, accepting the identity of the key *iff* the principal whose key it is has *at most* a thin set in the node policy (as is the case for the guest). We present results for the more naive case, and can derive the performance for this more optimized one.

```
firewall = <firewall's key>
guest = <guest's key>
acl = {
  ...
  ( { guest }, {},
    firewall$guest_thinned_services )
  ...
}
```

Figure 3. Host QCM Program

How we choose to specify the guest's thinned environment may be accomplished in a number of ways. The simplest way would be specify the thinned environment statically, at each host A and B. However, a more uniform and manageable approach would be that the guest identity is known locally, but its environment is defined at the firewall. The salient part of our host QCM program is shown in Figure 3.

The *thin* set is defined by the variable *guest_thinned_services* at principal *firewall.* Notice that the *thicken* set is empty. To short-circuit remote queries, the firewall provides certificates during node-node authentication that indicate the contents of its *guest_thinned_services* variable. Should the firewall policy be updated after initial authentication, the firewall would push certificates to the end host to reflect this change.

6.2 Performance Analysis

We analyze the performance of our active firewall by comparing a filtered and non-filtered ping. In both cases,

```
fun reply(payload) =
  print("Success")

fun ping(payload) =
  OnRemote(|reply|(payload),
           getSource(), getRB(),
           defaultRoute)
```

Figure 4. Ping in PLAN. Service invocations are in italics.

the initiating host lies in the trusted network and is pinging a node in the untrusted network. The PLAN code for ping is illustrated in Figure 4. Our analysis examines the additional cost to elapsed time and packet size.[4] For our experimental setup, we daisy-chain connect three machines with 100 Mbit Ethernet, configuring the middle machine as the active firewall. Each machine is a 300 MHz Pentium II with 250 MB of memory running Linux 2.0.30. PLANet runs directly on top of Ethernet.

Time Overhead

As described in the previous section, the addition of the firewall affects the packet processing time on the router and on the host initiating the "ping." While a router would normally just forward any packet it receives, the firewall has to additionally sign and encapsulate packets destined for the trusted network. On the initiating host, normal interpretation of the "reply" packet is further burdened by the need to decapsulate, verify the firewall's signature, and thin the environment.

Figure 5 illustrates the elapsed time of ping with and without the firewall. The left figure is the end-to-end time, in which the black bar is the unmodified ping and the white bar is the overhead imposed by the firewall. The right figure similarly illustrates salient component costs for the end host and the firewall with the addtional overhead. For the end host, the time consists of evaluating ping's "reply" packet, while for the firewall, this is the cost of forwarding the packet. The portion of the overhead which may be attributed to signing (at the firewall) and verifying (at the end host) is singled out. In both figures, times are given for 0-byte payloads and maximally-sized payloads. Notice that the overhead added to the component costs, which are the white and gray bars in the figure on the right, add up to the difference in elapsed time for the overall cost, which are the white bars in the figure on the left.

[4]The reader may note that the numbers reported here are slightly different than those reported in [22]; this is due to changes made to the PLANet implementation.

The base ping times for 0-byte and maximal payloads are 2.13 and 3.06 ms, respectively; the firewall adds 37% and 32% of respective overhead to these times (raising them to 2.91 and 4.03 ms). By examining the component costs, we can see that of this overhead, between $1/3$ and $1/2$ is attributable to signing and verification, based on the packet size. For the firewall, the remaining overhead is due to encapsulation costs (which requires extra marshalling and copying), while for the end-host it is due to decapsulation and the additional interpretation cost of the wrapper code. The time to thin the environment at the end host is negligible because we cache the thinned environment. If we eliminate the cryptographic operations, by the means described earlier, we reduce the end-to-end ping times to 2.58 and 3.41 ms for 0-byte and maximal payload, respectively. This reduces the firewall-induced overhead to 20% and 11%.

Notice that the graph depicts verification (which in the figure is the cryptographic component cost for the host) as twice as expensive as signing (which is the cryptographic cost for the firewall). This is due to two related points: we unmarshal PLAN programs *eagerly*, and in order to verify a PLAN value (that is, the original packet's chunk) using `authEval`, that value must first be marshalled into a binary format. These two points combine to mean that we unmarshal the encapsulated chunk when the packet arrives, only to re-marshal it when performing the signature verification. A smarter implementation would unmarshal chunks *lazily*, thus avoiding this extra re-marshalling cost and thereby equalizing signing and verification time.

There is room for further improvement. The cost of the cryptographic operations (for cases when they are actually needed) could be reduced through parallelism (to improve throughput) and special-purpose hardware (to improve both throughput and latency). Furthermore, the cost of PLAN interpretation is extremely high; a smarter interpreter would improve both the cost of the basic ping as well as the encapsulated version. In fact, we have recently been developing a compiler from PLAN to the low-level packet language SNAP, resulting in significantly improved performance [23, 36].

Space Overhead

The firewall also imposes a space-cost due to the extra code and signature that is attached to the incoming packets. Table 6 illustrates the basic space overheads, with and without the firewall.

The no-payload reply packet is 80 bytes (consisting of code and fixed fields), while the encapsulated version is 181 bytes, for an overhead of 126%. Of the 101 bytes of overhead, 12 bytes are due to the signature. Since the overhead is fixed, its impact is reduced with packet size. Looking at the maximally-sized packet, we see that this 101 bytes only

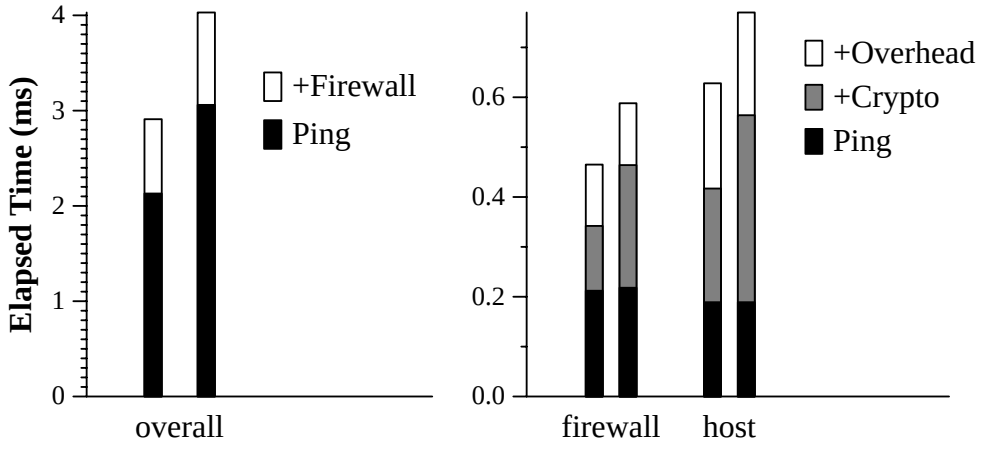

Figure 5. Ping elapsed time with and without the firewall. The left bar of each pair is with a 0-byte payload, and the right bar is for maximally-sized (1500 byte) packets.

	No payload		Maximal payload	
	packet size	rel. overhead	payload size	rel. overhead
ping reply	80 B	*n/a*	1420 B	*n/a*
+firewall	181 B	126%	1319 B	6.8%

Figure 6. Ping reply packet overhead with and without the firewall. Illustrates the additional cost of encapsulation and signing of foreign packets. Note that the signature itself is 12 bytes long.

adds 6.8% of overhead above the 5.3% already imposed by the ping program.

A particular concern is that by adding code to the packet as it passes through the firewall we might exceed the link layer MTU and be forced to fragment the packet. In the pathological (though probably not uncommon) case, each packet received by the firewall will be just smaller than the MTU and thus have to be fragmented after addition of the wrapper code. This problem also appears in the IPsec context, where it remains open to further research. One advantage that we have over IP is that in PLANet we may easily send PLAN programs to customize the host processing (*i.e.*, as a more expressive ICMP). It would be worth examining how to best express in PLAN a mechanism similar to "Path MTU Discovery" [33]. Another possible approach would be to compress the incoming packet, adding a wrapper to perform the decompression upon arrival at the end-host.

A concern about the approach of PLANet in general is the space cost of carrying the code in the packet. To mitigate this overhead, we have considered ways in which the participants in a protocol may cache code rather than always transmitting it with the packet. One approach is to add language-level *remote-references* which may be thought of as pointers to remote objects. Since all PLAN values (including chunks) are *immutable*, the contents of a remote reference may be safely cached without the need for a coherence protocol. In the case of our firewall, the wrapper function code could reside at the firewall, while being cached at the various hosts in the trusted network, thus reducing the in-packet space costs. The issue of code caching is discussed in more detail in [24].

7 Related Work

Securing active networks [37] has demanded three major research thrusts:

- First is the use of programming environments to offer safety and security guarantees, for example the careful design of PLAN and SNAP for safety, the use of module-thinning in ALIEN, and the capability-like namespace isolation scheme ANTS achieves with its MD5 hashes of active packets.

- Second is the extension of the local guarantees achievable within a programming environment to the collection of nodes comprising a network. While PLAN or SNAP, as examples of domain-specific languages,

provide such guarantees irrespective of location, they cannot make such guarantees when remote services are invoked. Cryptographic techniques can extend local safety properties by providing capability-like authorizations for services, as was done in extending ALIEN's protection to remote systems in SANE, and similarly in SANTS [38]. SANTS, which uses an authorization scheme similar to ours, further considers how to handle changes made to the contents of cryptographically signed packets as they traverse the network. However, as Alexander showed in his Caml-based architecture [1], the performance penalty of frequent cryptographic operations can be substantial.

- Third is support for multithreaded operation of active networking systems in ways that provide resource protection. This work has been centered around the lowest levels of the DARPA active network architecture, the so-called "Node Operating System" [42], examples of which include RCANE [31], JanOS [44], AMP [42] and Scout [34]. These systems manage resources which may be used by safe programming environment in service invocations, including management of resources used concurrently by multiple programming environments.

0ur use of `authEval` resembles Java stack inspection (JSI) [15, 45]. In our case, code is afforded the privilege of the principal that signs it for the duration that it runs. JSI refines this idea by examining the call stack and giving the code the privilege of the least privileged principal found on the stack, except when more trusted code explicitly widens the privilege of its callers by invoking `enablePrivilege`. It would be interesting to apply the same approach to nested `authEval` calls to ensure the same sort of security guarantees.

8 Conclusions

The Secure PLAN architecture is a hybrid which couples highly-scrutinized active extensions with unauthenticated active packets supported by these extensions. This has two major advantages. First, packets which do not require the computational cost of authentication and authorization do not pay it. This is because all potentially unsafe computation is relegated to the service level, which can be governed by trust-management techniques. Our experience is that the majority of active packet programs, from diagnostics such as *ping* to best-effort data delivery, require no potentially unsafe services, and therefore should not require authentication. The second advantage, which follows from the first, is that security analysis, perhaps including validation and verification, can be focused on a small set of service routines rather than all possible active programs. That said, it is an important avenue of future work to find ways to automatically certify services as safe, so that they do not need to be protected by a trust-based policy. Proof-Carrying Code [39, 40] is one way to certify safety in low-level code, but so far only simple safety properties have been explored. A related approach uses dependent types to ensure that services consume a bounded amount of time and/or space [13].

While our system uses both programming environment-based safety and cryptography-based techniques to support use of services in networks (and is compatible with any NodeOS approach), the novel architectural contribution is the combination of enforcement mechanisms to allow policy-writers to balance flexibility with performance. In particular, we support both namespace-based security to add to or subtract from a packet's default service namespace, and policy-based parameterization to allow services to formulate their own per-principal usage policies. Namespace-based security can be enforced cheaply at authentication-time, while policy-based parameterization may require per invocation checks. We have sought to enable scalability by carefully encoding the namespace-based policy, and by using a decentralized trust management system [11].

The active firewall is a novel application of namespace-based security. The firewall uses PLAN packets' activeness to protect a trusted environment from untrusted computations. We have demonstrated that our architecture addresses possible threats while still preserving the flexibility and usability of the system, by *actively* modifying the packet behavior, under control of a trust management policy, rather than simply making a permit/deny decision as would be made by a traditional firewall architecture. This architecture is based on language safety, authentication, and trust management. We demonstrated the practicality and acceptable performance of our approach experimentally, using an implementation of the active firewall.

Acknowledgements

We would like to thank Scott Nettles, Jonathan Moore, and Trevor Jim for helpful discussions concerning this work, and the anonymous referees for providing useful feedback. We would also like to thank Trevor Jim for providing the PLAN-based implementation of QCM.

This work was done while all authors were at the University of Pennsylvania, supported by DARPA under Contract #N66001-96-C-852, NSF under grant #ANI 98-13875, with additional support from the Intel Corporation. A shorter version of this paper was published in the International Working Conference on Active Networks [21].

References

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

[2] 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*, 12(3):29–36, 1998.

[3] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, S. Muir, and J. M. Smith. Secure quality of service handling (SQoSH). *IEEE Communications*, 38(4):106–112, April 2000.

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

[5] D. S. Alexander, P. B. Menage, A. D. Keromytis, W. A. Arbaugh, K. G. Anagnostakis, and J. M. Smith. The price of safety in an active network. *Journal of Communications (JCN), special issue on programmable switches and routers*, 3(1):4–18, March 2001.

[6] K. G. Anagnostakis, M. W. Hicks, S. Ioannidis, A. D. Keromytis, and J. M. Smith. Scalable resource control in active networks. In H. Yashuda, editor, *Proceedings of the Second International Working Conference on Active Networks*, volume 1942 of *Lecture Notes in Computer Science*, pages 343–358. Springer-Verlag, October 2000.

[7] W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. DHCP++: Applying an efficient implementation method for fail-stop cryptographic protocols. In *Proceedings of Global Internet (GlobeCom) '98*, pages 59–65, November 1998.

[8] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000. USENIX.

[9] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of 15th Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[10] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, Lecture Notes in Computer Science. Springer-Verlag Inc., New York, NY, USA, 1999.

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

[12] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. In *ACM Transactions on Computer systems*, November 1994.

[13] K. Crary and S. Weirich. Resource bound certification. In *Symposium on Principles of Programming Languages*, pages 184–198, 2000.

[14] W. Diffie, P. van Oorschot, and M. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[15] C. Fournet and A. Gordon. Stack inspection: Theory and variants. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 2002.

[16] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, 1996.

[17] C. A. Gunter and T. Jim. Policy-directed certificate retrieval. *Software - Practice and Experience*, 30(15):1609–1640, 2000.

[18] C. Hawblitzel, C. Chang, and G. Czajkowski. Implementing Multiple Protection Domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 259–270, June 1998.

[19] M. Hicks. PLAN system security. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, April 1998.

[20] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.

[21] M. Hicks and A. D. Keromytis. A secure PLAN. In S. Covaci, editor, *Proceedings of the First International Workshop on Active Networks*, volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, June 1999.

[22] M. Hicks, J. T. Moore, D. S. Alexander, C. A. Gunter, and S. Nettles. PLANet: An active internetwork. In *Proceedings of the Eighteenth IEEE Computer and Communication Society INFOCOM Conference*, pages 1124–1133. IEEE, 1999.

[23] M. Hicks, J. T. Moore, and S. Nettles. Compiling PLAN to SNAP. In I. W. Marshall, S. Nettles, and N. Wakamiya, editors, *Proceedings of the Third International Working Conference on Active Networks*, volume 2207 of *Lecture Notes in Computer Science*, pages 134–151. Springer-Verlag, October 2001.

[24] M. Hicks, J. T. Moore, D. Wetherall, and S. Nettles. Experiences with capsule-based active networking. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE)*. IEEE, May 2002. This volume.

[25] M. W. Hicks. PLAN security guide, 2001. Part of PLAN 3.2 documentation. Available at `http://www.cis.upenn.edu/~switchware/PLAN/docs-ocaml/security.ps`.

[26] S. Kent and R. Atkinson. Security architecture for the internet protocol. Technical Report RFC 2401, IETF, Nov. 1998.

[27] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. Technical report, IETF RFC 2104, February 1997.

[28] X. Leroy. *The Objective Caml System, Release 3.00*. Institut National de Recherche en Informatique et Automatique (INRIA), 2000. Available at `http://caml.inria.fr`.

[29] 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 (JSAC)*, 14(7):1280–1297, September 1996.

[30] J. Y. Levy, J. K. Ousterhout, and B. B. Welch. The Safe-Tcl security model. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 271–282, June 1998.

[31] P. Menage. RCANE: A resource controlled framework for active network services. In S. Covaci, editor, *Proceedings of the First International Workshop on Active Networks*, volume 1653 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1999.

[32] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[33] J. Mogul and S. Deering. Path MTU Discovery. Internet RFC 1191, November 1990.

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

[35] J. Moore. Mobile Code Security Techniques. Technical Report MS-CIS-98-28, University of Pennsylvania, May 1998.

[36] J. T. Moore, M. Hicks, and S. Nettles. Practical programmable packets. In *Proceedings of the Twentieth IEEE Computer and Communication Society INFOCOM Conference*, pages 41–50. IEEE, April 2001.

[37] Security architecture for active nets, June 1998. Draft available at `http://www.ittc.ukans.edu/~ansecure/0079.html`.

[38] S. Murphy, E. Lewis, R. Watson, and R. Yee. Strong security for active networks. In *Proceedings of the IEEE Conference on Open Architectures and Network Programming*. IEEE, April 2001.

[39] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, New York, January 1997.

[40] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating System Design and Implementation*, pages 229–243. USENIX, 1996.

[41] E. L. Nygren. The design and implementation of a high-performance active network node. Master's thesis, Massachusetts Institute of Technology, February 1998.

[42] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullman, J. Lepreau, S. Schwab, H. Dandekar, A. Purtell, and J. Hartman. An OS interface for active routers. *IEEE Journal on Selected Areas in Communications (JSAC)*, 19(3):473–487, March 2001.

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

[44] P. Tullmann, M. Hibler, and J. Lepreau. Janos: a Java-oriented OS for active network nodes. *IEEE Journal on Selected Areas in Communications (JSAC)*, 19(3), March 2001.

[45] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–63, May 1998.

[46] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *17th Symp. on Operating Systems Principles (SOSP'99)*, pages 64–79, Kiawah Island, SC, December 1999. ACM.

[47] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OpenArch Proceedings*. IEEE Computer Society Press, Los Alamitos, April 1998.