A Secure PLAN

Michael Hicks, Angelos D. Keromytis, and Jonathan M. Smith

***Abstract*—Active networks, being programmable, promise greater flexibility than current networks. Programmability, however, may introduce safety and security risks.**

This correspondence describes the design and implementation of a security architecture for the active network PLANet [1]. Security is obtained with a two-level architecture that combines a functionally restricted packet language, PLAN [2], with an environment of general-purpose service routines governed by trust management [3]. In particular, a technique is used which expands or contracts a packet's service environment based on its level of privilege, termed *namespace-based security*.

The design and implementation of an active-network firewall and virtual private network is used as an application of the security architecture. Measurements of the system show that the addition of the firewall imposes an approximately 34% latency overhead and as little as a 6.7% space overhead to incoming packets.

***Index Terms*—Active firewall, active networks, active packets, PLAN, programming languages, security.**

I. Introduction

Active networks [4] offer the ability to program the network on a per-router, per-user, or even per-packet basis. Unfortunately, this added programmability threatens the security of the system by allowing a wider range of possible 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 [1]. 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 the packet language for active networks (PLAN), [2]. 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 utilities. 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 [5]. This general architecture is shared by many so-called active packet systems, including ANTS [6]–[8], SNAP [9], PAN [10], and others.

A central goal of PLANet is to provide Internet-like service as a baseline, augmented by 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, and protocols like HTTP, FTP, and SMTP 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*, i.e., 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 its contents have not been tampered with, 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 should be 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: using 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. 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 our architecture was developed specifically for PLANet, we believe it is more broadly applicable. In particular, it will apply to any active network infrastructure that manages general-purpose, node-resident services in combination with safe (whether active or passive) packets. Our approach to service security is also relevant to extensible systems, like some web servers and operating systems.

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 discussion of PLAN and its relevant characteristics, we present possible methods of security management and the ones we have chosen to implement are *namespace-based security* with *policy-based parameterization*. We describe how we enable authentication, and manage relevant security information, such as which service routines are available to which principals, using query certificate manager (QCM [11]). We then demonstrate how we have used our system to implement two applications: a simple active firewall, and a virtual private network for active packets. Finally, we present some related work and conclude.

II. 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.

Manuscript received July 1, 2002; revised March 24, 2003 and June 28, 2003. A shorter version of this paper was published in the International Working Conference on Active Networks [68], and an extended version of that paper was published in the DARPA Active Networks Conference and Exposition [69]. This work was supported by DARPA under Contract N66001–96-C-852, NSF under Grant ANI 98–13875, with additional support from the Intel Corporation. This paper was recommended by Guest Editors W. Pedrycz and A. Vasilakos.

M. Hicks is with the University of Maryland, College Park 20742, USA (e-mail: mwh@cs.umd.edu).

A. D. Keromytis is with Columbia University, New York, NY 10027, USA (e-mail: angelos@cs.columbia.edu).

J. M. Smith is with the University of Pennsylvania, Philadelphia, PA 19104, USA (e-mail: jms@cis.upenn.edu).

Digital Object Identifier 10.1109/TSMCC.2003.817347

A. 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.

1) **Denial-of-Service**. Because of the greater expressibility of active network programs (compared to traditional passive 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. We address only active node-specific denial-of-service (DoS) considerations; the much harder network DoS problem is better addressed through other means (e.g., [12], [13]).
2) **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.

Passive networks are vulnerable to these same attacks; active networks simply expand the "vocabulary" of an attacker. For example, an attacker can mount a denial-of-service attack over the Internet by attempting to overload a web server with a constant flow of HTTP GET requests. If the attacker has enough resources (such as a coordinated fleet of "drone" machines to make requests), it can overwhelm the ability of the server to perform useful work. An active network can make such attacks easier (particularly when, like PLANet, it provides active packets) because it increases the maximum amount of resources required to process a single packet, and thus the attacker needs fewer resources to overwhelm its target. The goal of PLANet's security architecture is to reduce the effect and increase the difficulty of mounting attacks, particularly denial-of-service attacks, while still preserving the utility of the network's active capabilities. At this point, auditing techniques can be used to discover the source of an attacker, such as IP traceback [14], [15] and pushback [12]. Moreover, such techniques are more easily implemented and deployed in an active setting.

B. Architecture

As already described in Section I, 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 justify this claim in Section III.

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 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 its 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 Fig. 1.

[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 its public key.

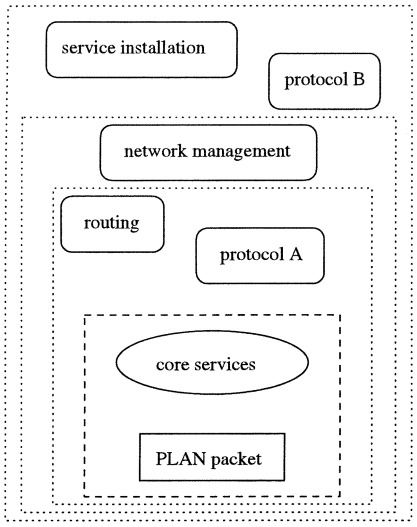

Fig. 1. PLANet's security architecture. The contents of the dashed box are available to all incoming packets, while the dotted boxes encapsulate service packages available only to select users. Services may be further restricted by what parameters they can be called with.

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 Sections III and IV. We describe PLAN's security properties in the Section IV, and then present our service management methodology.

III. Packet Security via PLAN

PLAN [2] is a small functional language resembling ML [16], [17]. It differs most importantly from other functional languages in that it provides language-level support, using the primitive `OnRemote` among others, for evaluating an expression at a remote node. Invoking `OnRemote` results in a newly spawned packet that is sent to and evaluated at the remote location. PLAN was designed as the foundation of PLANet's security, with the intention that all PLAN programs can be considered safe. In this section we introduce PLAN and describe the language's security properties.

A. PLAN: The Packet Language for Active Networks

PLAN supports standard programming features, such as functions and arithmetic, and features common to functional programming, like lists and the list iterator *fold* (intuitively, *fold* executes a given function f for each element of a given list, accumulating a result as it goes). Two notable restrictions are that functions may not be recursive and iteration must be bounded. PLAN programs call service routines present on the executing node using normal function call syntax. These services are implemented in a full programming language such as C, Java [18], Cyclone [19], ML [16], or any other language.

PLAN's `OnRemote` primitive is used to direct a computation to take place on a different node, and has the effect of creating a new packet that is sent to that node to initiate the computation. The computation is specified as a function call to perform remotely, along with 0 or more arguments that are evaluated locally. The following example executes

the function f at the node *host* with the argument 4; the arithmetic $1+3$ is performed by the invoking node.

$$\texttt{OnRemote}(|f|(1+3), \texttt{host}\ldots).$$

`OnRemote` takes two additional arguments (the ... in the above example).

1) A *resource bound* count, which is greater than 0. Each packet has associated with it a resource bound that is decremented on each hop, as with the IP "Time To Live" (TTL) counter. When a new packet is created with `OnRemote`, its resource bound is initialized by subtracting the specified amount from the parent packet program.
2) A *routing function*. This is the name of the service that is to provide hop-by-hop lookups to route the packet to its final destination. A variant of `OnRemote`, called OnNeighbor, does not require a routing function, but restricts spawned packets to execute only on immediately adjacent nodes. These packets are therefore responsible for their own routing.

Remote evaluation with `OnRemote` is best-effort and asynchronous: the `OnRemote` call returns immediately and does not wait for any result from the spawned packet.

PLAN provides the ability to manipulate programs as data, via a construct known as a *chunk* (short for "code hunk"). A chunk may be thought of as a function that is waiting to be executed. In PLAN, chunks are first-class—they can be passed as arguments to functions and stored in variables—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. The syntax $|\mathrm{f}|(4)$ in the above example is used to define a chunk literal; when this chunk is evaluated, the function f will be executed with the argument 4. A chunk can also be evaluated manually by passing it to the `eval` service, which resolves the function name with the current environment, performs the application, and returns the result. The code $\mathrm{eval}\ (|\mathrm{f}|(4))$ is thus equivalent to simply invoking $\mathrm{f}(4)$. Chunks play an important role in service security, as we discuss in Section V.

Fig. 2 shows how to program *ping* in PLAN. This program is executed by packaging it into a packet and sending it to our ping target, indicating it should evaluate the function ping upon arrival. This results in the chunk $|\texttt{reply}|(\texttt{payload})$ being created and sent back to the source of the original packet, as determined via the `getSource` service routine. The call to `getRB` returns all of the current packet's resource bound, which is here donated to the new packet. The new packet is routed using the `defaultRoute` routing service. When the return packet evaluates at the source, it prints the message "success."

B. 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.

1) **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 by exploiting unchecked array accesses to force buffer overflows. PLAN prevents such protection attacks by enforcing strong typing, as is done in languages like Java, ML, and Modula-3.
2) **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 nonfixed-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).
3) **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. Therefore, the number of hops that a PLAN program and any of its progeny may take is limited by the initial value of this counter, thus preventing denial-of-service attacks on the network infrastructure.

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

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

Fig. 2. Ping in PLAN. Service invocations are in italics.

The first mechanism is widely understood in both the active networks community and the extensible operating systems and mobile code communities [6], [9], [20]–[24]. It has the nice benefit that capability-style protection can be enforced by the language, dramatically reducing protection costs. The latter two mechanisms have come into common usage in packet-based active network schemes [6], [9], and [25], but the first technique of the two is less appreciated. Most active network systems of which we are aware assume that a general-purpose, type-safe language combined with resource counters is sufficient; misbehaving threads are simply killed when they exceed their resource limits.

However, abrupt termination is both potentially unsound, and quite costly [26]. 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. Systems using language-based protection typically restrict sharing, at some performance cost, to support safe termination, [22], [27]. Operating systems have traditionally segregated processes into distinct address spaces, at a significant performance penalty to interprocess communication, so that they can be killed abruptly without worry of shared resources. In PLANet we require neither mechanism because we are guaranteed that packet programs will terminate on their own.

C. Resource Bounding

While guaranteed termination is an important property, to adequately defend against denial-of-service attacks we must strengthen it to bound the resources consumed prior to termination. The 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.*

Such a property is useful for active networks because it directly relates a router's resource usage to the number and size of the packets it

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

processes. For example, it can know the maximum amount of memory needed to execute the packet, based on its size. If a router is experiencing overload, it can decide to drop packets based solely on their maximum possible resource usage (based on their size), without having to partially evaluate them or examine their contents.

Of course, even a linear relationship is unhelpful if the constant of proportionality is large. As we discussed earlier, the constant of proportionality for routing IP packets is very small, which requires an attacker to amass substantial resources to mount a denial-of-service attack by flooding. We would prefer at the least to retain this state of the affairs for an active network.

To satisfy a linearity property in PLAN, we must rule out programs like the following one, which executes in time exponential in its length.

$$\begin{aligned}\texttt{fun f1}() &= () \\ \texttt{fun f2}() &= (\texttt{f1}();\ \texttt{f1}()) \\ \texttt{fun f3}() &= (\texttt{f2}();\ \texttt{f2}()) \\ \texttt{fun f4}() &= (\texttt{f3}();\ \texttt{f3}()) \\ \texttt{fun exponential}() &= (\texttt{f4}();\ \texttt{f4}()).\end{aligned}$$

This program defines five functions (that do nothing), but requires a total of 31 function calls to completely evaluate `exponential` (or $2^n - 1$ calls, where n is the number of function definitions). We prevent such programs by requiring that for any PLAN function f, which calls some number of other PLAN functions $g_1 \ldots g_n$, the sum total of PLAN functions called by $g_1 \ldots g_n$ is at most 1. Moreover, we place a constant bound c on the length of lists to be iterated over with fold ; each multiple of c decrements one resource bound from the packet.

More recently, a follow-on to PLAN called SNAP [9] has been proposed, which is an assembly-like language for packet programming. SNAP programs meet the linear resource usage property with a small constant of proportionality. For example, SNAP instructions can allocate at most three words per instruction. We have developed a compiler to compile PLAN programs into SNAP programs, which essentially imposes the restrictions we have described above [28]. 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, more work is needed to better understand the tradeoff between resource security and flexibility in unauthorized packets.

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 more easily launched. For this reason, it is critical that a service management system be in place. We discuss our approach, among others, of using trust management to manage namespaces in Section IV.

IV. 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 II-A. 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.

A. 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. We can expand on this observation to arrive at the following requirements for our setting.

1) 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 its users and services.
2) Enforcement mechanisms
   - To minimize the size of our *trusted computing base*, enforcement mechanisms should be simple to understand and employ [29]. 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* [3]. In a trust management system, each user, or *principal*, is assigned some level of trust (or privilege). 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 the 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 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 protected service routine, 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 Section V.

B. 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 store state on routers temporarily might apportion different amounts of space to different principals. We call such per-principal differences in service usage *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 VI. 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 VII-A. 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 ids 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 QCM, and the way that we formulate our policies in Section VI.

Beyond this service-based policy, we might like to specify more general resource usage constraints, such as bounding CPU and memory use. While we do not consider such constraints in this paper, they have been considered in work we have done elsewhere. In particular, we have found that resource-based policies can be achieved with assistance from lower-level system software, as in the SQoSH [30] and RCANE [31] systems, which share a software base used to implement many PLAN services. SQoSH uses trust management techniques to control a virtual-clock based bandwidth allocation system, and RCANE uses 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 [32], to provide resource guarantees.

C. 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 a PLAN program, 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, the action fails and an exception is raised. In effect, this is a 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 this approach. First, only those packets that use privileged (noncore) 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.

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 [33] built on top of ALIEN [34]. Here we used Objective Caml [16] as our service language, and extended its support for namespace-based security (referred to as *module thinning* by Rouaix [35]) to support our policies.

In Sections V and VI, we describe the mechanisms used by PLAN programs for authentication and authorization.

V. Implementing Authentication

Before a PLAN program may invoke a trusted service, its associated principal must be determined; this is the process of authentication. Authentication in open networks 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. Before we describe chunk authentication, we give an overview of the basic principles behind public key cryptography and digital signatures.

A. Public Key Cryptography and Authentication

A cryptographic algorithm is "symmetric" if the same key is used to encrypt and decrypt (e.g., DES [36]). Public key systems use two different keys: a private key, $K_{private}$, and a public key, K_{public}, where $D_{K_{private}}(E_{K_{public}}(M)) = M$. That is, a message is encrypted by principal's public key and then decrypted by its private key. Examples of public key cryptographic systems are RSA [37] and DSA [38].

Public key cryptographic systems have a significant advantage over symmetric systems in that two principals can exchange a message, or verify the validity of a digital object, provided they have acquired the peer's public key in some trusted manner, without need to engage in some real-time exchange. In contrast to symmetric key systems, where the principals must exchange keys in a confidential manner, public keys do not need to be confidential.

Digital signatures use a public key system to bind an object to a public/private key pair. To sign the object, the signer computes a function of the object and the private key.[3] The result of this function is verifiable by anyone knowing the corresponding public key. A valid signature assures the verifier that, modulo bad key management practices on the part of the signer or some break-through in forgery, the signed object was indeed signed by the signer's private key and that it has not been modified since that time.

Public key certificates (e.g., X.509 [39]) are statements made by a principal (as identified by a public key) about another principal (also identified by a public key). Public key certificates are cryptographically signed, such that anyone can verify their integrity, i.e., the fact that they have not been modified since the signature was created. They are typically used to bind a public key to some form of identity, such as an IP address, a DNS name, an email address, etc.

B. Authenticating Chunks

As described in Section III-A, a chunk encapsulates some PLAN code, and can be executed remotely using `OnRemote` or locally using `eval` . We have added a service called `authEval` which takes as arguments a chunk, a digital signature, and a public key. Here, the signature is the result of signing the binary representation of the provided chunk using the private key that pairs with the provided public key. `AuthEval` verifies the signature, and 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 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. Second, only those programs that require authorization must pay 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. We address these problems by using an additional authentication protocol developed as part of work on secure active network environment (SANE) [31], [40]. We briefly describe SANE next, and describe how its protocol is implemented in PLANet.

C. SANE

A key goal of SANE is to enable a sphere of trust among various nodes and/or applications across a distributed, potentially untrusted infrastructure. To achieve this, SANE defines a novel cryptographic authentication protocol, which allows a principal and a node to authenticate each other and generate a *shared secret* and an identifier for that secret, named "*SPI*." Once the protocol is completed, parties can use the shared secret to authenticate via HMAC-SHA1 [41] digital signatures, in a way similar to that used in the IPsec [42] 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, e.g., an integer keeps track of the largest sequence number received, and a 64-bit mask shows 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.

The SANE protocol is essentially a variation of the station-to-station protocol (StS) [43], which builds on top of the Diffie-Hellman key agreement protocol (DH) [44]; these protocols permit two parties to establish a shared secret over an untrusted communication medium. We describe these protocols briefly below, and then describe the SANE authentication protocol and how it is implemented in PLANet.

The DH algorithm is based on the difficulty of calculating discrete logarithms in a finite field. Each participant agrees to two primes, g and p, such that g is primitive $\bmod p$. These values do not need to be protected in order to ensure the strength of the system, and therefore can be public values. Each participant then generates a secret large random integer. Bob generates x as his large random integer and computes $X = g^x \bmod p$. He then sends X to Alice. Alice generates y as her large random integer and computes $Y = g^y \bmod p$. She then sends Y to Bob. Bob and Alice can now each compute a shared secret, k, by computing $k = Y^x \bmod p$ and $k = X^y \bmod p$, respectively. The values X, Y, g, and p can all be made public without loss of security.

Unfortunately, the Diffie-Hellman algorithm is susceptible to a man-in-the-middle attack. The attack can be defeated, however, by combining Diffie-Hellman with a public key algorithm, such as DSA or RSA, as proposed in the station to station protocol.

In its simplest form, shown in Fig. 3, StS consists of a Diffie-Hellman exchange, followed by a public key signature authentication step, typically using the RSA algorithm in conjunction with some public-key certificate scheme such as X.509 [39]. In most implementations, the

[3]Usually a "summary" of the object is signed, computed through a cryptographic one-way hash function.

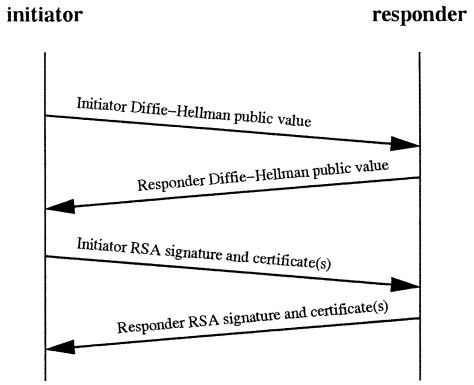

Fig. 3. Four-message station to station key agreement protocol.

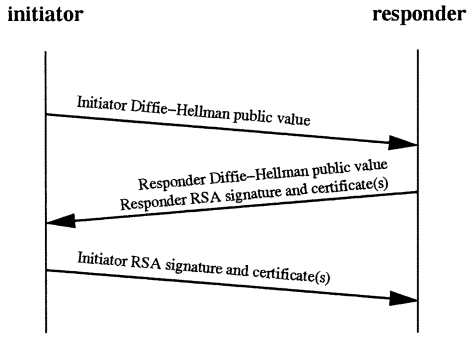

Fig. 4. Three-message station to station key agreement protocol.

second message is used to piggy-back the responder's authentication information, resulting in a three-message protocol, shown in Fig. 4. Other forms of authentication may be used instead of public key signatures (e.g., Kerberos [45] tickets, or preshared secrets), but these are typically applicable in more constrained environments. The short version of the protocol has been proven to be the most efficient [46] in terms of messages and computation.

D. SANE Authentication Protocol in Planet

The SANE authentication protocol is a variation of the StS protocol. Here we describe the protocol in terms of its implementation in PLANet, assuming that an application wishes to mutually authenticate with an active node. Analysis and further details can be found in the SANE papers [31], [40], and the PLAN documentation [47].

1) To start, the user application requests authentication with a remote PLAN node. The application sends a PLAN program to the node with which it wants to authenticate. This program invokes the PLAN service `DHmessageOne` with two arguments: a *certificate*, and a signature of that certificate using the user's public/private key pair. In the current implementation, we use DSA [38] keys for authentication. All certificates used during the exchange are PLAN tuples[4] which begin with the following four fields:
   — signer's public key;
   — random number (a *cookie*);
   — time at which certificate is valid;
   — time at which certificate expires.

   The latter three of the above fields are are essentially used to prevent replay attacks. Note that the duration of time fields implies the level of synchronization between the two nodes' clocks. The remaining fields of the first certificate are:
   — sender's *exchange id*;
   — receiver's address;
   — sender's address.

   The exchange id is generated at the sender, and is used to identify this particular protocol exchange. At the completion of the protocol it will be used to establish the *SPI*, described later. For a node, the address is represented as a PLAN *host*, while an application's address is of type $\texttt{host} \times \texttt{port}$.
2) When the node receives the message, it verifies the signature on the certificate with the certificate signer's key. It then makes sure the certificate is active and has not expired, and that the receiver's address is the current node.

   If the current node wishes to negotiate with the sender, it creates a bit of state to keep track of the exchange. It stores the sender's exchange id, calculates its own exchange id, and additionally stores the sender's address and public key. It also calculates its local portion of the shared secret and the "public" value of this secret. The response certificate includes both exchange id's, the public value, the application's public key, and both addresses.

   Since the response message is being sent to an application, rather than a node, it is packaged as a tuple, labeled by a string "DHmessageTwo," to be delivered to the application. This tuple also contains the certificate and its signature.
3) Application verifies the signature, looks up the exchange id to find the information stored about this exchange, and verifies that it is all correct. It then calculates its secret and corresponding public value, then combines it with the value in the message to produce the shared secret. The SPI identifying the secret is then calculated based on the two exchange id's. This SPI is used to identify the secret in later messages which have been signed using the secret. The application's public value is included in a message back to the node which is essentially a mirror of the message just received. As described earlier, this third message is actually a PLAN program that invokes the PLAN service `DHmessageThree` with two arguments: the certificate containing the described information, and a signature of that certificate using the user's public/private key pair.
4) The node receives the final message and repeats the actions taken by the application for the previous message. No response is sent; the protocol is complete.

Each principal in the exchange now has a secret known only to the other principal to be used for signing future communications. In our implementation, the secret is stored in two tables; one table indexed by the peer's address (which includes other information about the protocol), and another indexed by SPI. The former table may be used by the application when it wants to send a message to the peer, the latter table is used to look up the SPI found in a signed chunk so that the signature can be verified.

Note that this authentication exchange is not limited to an application contacting a node—nodes may contact other nodes and applications may contact other applications. The latter is in common practice in the Internet today, and the former may be used, for example, to establish trust relations between administrative domains.

VI. Implementing Authorization

As our policy manager, we have chosen to use the QCM [11], 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

[4]A tuple is simply an aggregate data structure, like a struct in C . A tuple that contains something of type int and something of type float would have type $\mathrm{int} \times \mathrm{float}$.

used instead. In particular, we have used the KeyNote [48] trust-management system in related work [33]. We begin by briefly describing QCM, and then explain how we encode our security policies as QCM policies.

A. QCM

According to the QCM website[5]

> A QCM is a server used for the authenticated distribution of sensitive information over an insecure network. A QCM server acts like a secure, distributed database: it queries remote QCM servers to answer local queries about distributed data, and ensures the authenticity of the data by cryptographic means. Moreover, QCM can accept *digitally-signed certificates* issued by remote servers. When such certificates are submitted along with the local query, queries to the remote servers are short-circuited. The management of queries and certificates is completely automatic and transparent to the user. Applications such as directory services, public key distribution, and distributed access control lists are directly programmable in QCM, and QCM has a formal semantics and correctness guarantees.

QCM manages data organized in sets, which can be built up from constants, like strings, integers, and keys, and from other sets, using set union. For example, a QCM server could define the set PKD that associates a user's name with his or her public key as follows:

$$\mathrm{PKD} = \{(\text{“Alice”}, K\,\mathrm{alice}), (\text{“Bob”}, K\,bob)\}$$

Sets can be queried by by using set comprehensions. For example, the following query resolves AliceKeys to the set that contains all of the keys associated with Alice in PKD.

$$\mathrm{AliceKeys} = \{\mathrm{k} |\ (\text{“ Alice”}, \mathrm{k}) \in \mathrm{PKD}\}.$$

Simple set membership can be performed creating a singleton set if and only if membership conditions hold, as in

$$\{\text{“yes”} |\ K\,\mathrm{alice} \in \mathrm{AliceKeys}\}$$

This query will resolve to the singleton set containing “yes” if the variable AliceKeys contains the given key; it will resolve to the empty set otherwise.

QCM set definitions are location-specific. That is, local namespaces are made global by prepending them with location/owner of the namespace: $K\,\$x$ is the global name of the local name x in $K's$ namespace, and is pronounced, “$K's$ x.” Here, K refers to the public key of a principal that holds data at its home server. Global names can be referred to from any location. For example, Alice may define her PKD by additionally incorporating Carl's PKD, which is defined by his remote QCM server:

$$\mathrm{PKD} = \{(\text{“Alice”}\,K\mathrm{alice}), (\text{“Bob”}, K\mathrm{bob})\}\ \text{union}\ K\mathrm{carl}\,\$\ \mathrm{PKD}$$

Part of the novelty of QCM is that querying a set with remotely-defined components can automatically result in a queries being sent to remote sites. For example, if the authorization service on K *alice* makes a membership test on set PKD, QCM will automatically query K *carl* if necessary. However, QCM is optimized to reduce communication overhead by being conservative: some queries can be resolved by using partial, local information only. For example, the membership test for Alice's key, above, would not require a remote query to Carl's site.

QCM also supports the use of *certificates*, which are signed assertions about set relationships, and can be used to avoid remote queries. For example, the following illustrates a certificate that asserts that K\$PKD contains at least the pair of Alice with her public key (and is signed by K).

$$\mathrm{K\ says}(\text{”Alice”}, K\,\mathrm{alice}) \in \mathrm{PKD}.$$

Such a certificate could be provided before making a query, and would potentially prevent messages from being sent to remote sites. Servers may wish to operate in a mode in which no remote queries are sent automatically, but instead relevant certificates must be provided up front. While more onerous for the user/application, this may prevent denial-of-service attacks on the certificate retrieval system.

The version of QCM that we use is implemented as a service in PLAN, and 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. Certificates may be passed as additional arguments to $\mathrm{authEval}$, or may be obtained during node-node authentication.

B. Implementing Service Policies in QCM

We use QCM sets to define both namespace-based security policies and per-service parameterizations. Using QCMs location-specific definitions and certificates should allow such policies to scale as the number of users, services, and nodes in the network grows.

Namespace Control Policies: Following our general policy requirements discussed in Section IV-B, 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; the former defines services that should be added to the service symbol table and the latter defines services that should be subtracted. Once a principal has been authenticated, QCM is queried and the symbol table is modified as directed; the 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:

$$\begin{aligned}
\mathrm{p1} &= \langle \mathrm{p1's\ \texttt{public key}} \rangle; \\
\mathrm{p1_svcs} &= \{\text{“}\texttt{print}\text{”}\}; \\
\mathrm{p2} &= \langle \mathrm{p2's\ \texttt{public hboxkey}} \rangle; \\
\mathrm{p2_svcs} &= \{\text{“}\texttt{thisHost}\text{”}\} \\
\mathrm{acl} &= \{(\mathrm{p1}, \mathrm{p1_svcs}, \{\}), \\
&\qquad (\mathrm{p2},\ \texttt{union}(\mathrm{p2_svcs}, \mathrm{p1_svcs}), \{\})\}.
\end{aligned}$$

In addition to identifying the keys of p_1 and p_2, we define two sets, $\mathrm{p1\_svcs}$ and $\mathrm{p2\_svcs}$, which specify the respective 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 $\mathrm{p1\_svcs}$, while the second says that $p_2's$ environment should be thickened by both $\mathrm{p1\_svcs}$ and $\mathrm{p2\_svcs}$. 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.

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) principal or set of principals (as in the ACL);

[5]http://www.cis.upenn.~edu/ qcm/.

2) 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:

$$\begin{aligned} \texttt{def} &= \langle \texttt{default user's key} \rangle; \\ \texttt{resident} &= \{(\texttt{def}, \langle \texttt{amount} = 100 \rangle), \\ &\quad (\texttt{p1}, \langle \texttt{amount} = 1000 \rangle)\} \end{aligned}$$

This policy indicates that unauthenticated 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,[6] 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.

Scaling Policies: An interesting question is how this infrastructure should be used to deploy services and update policies over a large administrative domain. In our prototype implementation, new service routines can be installed using the service routine `installServices`, which dynamically loads some provided code into the router. This service should obviously be privileged, requiring authorization to use. One benefit of the PLANet architecture is that interesting protocols or mechanisms (such as soft-state, unreliable packet delivery with fragmentation/reassembly, etc.) can be encoded using a few general services, with the majority of the logic being coded as PLAN [1]. As such, we expect that services will be added relatively infrequently, which implies that security policies will not change often (since the number of privileged users for a given domain is likely to be relatively static). Given this, one way to update the QCM policy for each node would be to allow local policies to refer to a single 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. For more fine-grained changes, we can augment local policy with certificates provided by authenticated programs.

VII. Applications

As a proof-of-concept of our security architecture, we have designed and implemented an *active firewall* using PLANet, as well as an *active virtual private network*. In this section, we describe both applications and present some performance measurements.

A. Simple Active Firewall

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 Fig. 5. When packets can be active, this simple approach can be too limiting.

Firewalls typically filter certain types of packets, for example 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, unless we wish to filter out all active packets (which could be the case when under a denial-of-service attack) we need an alternate way of stopping 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 our premise, stated in Section II-B, 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.

[6]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.

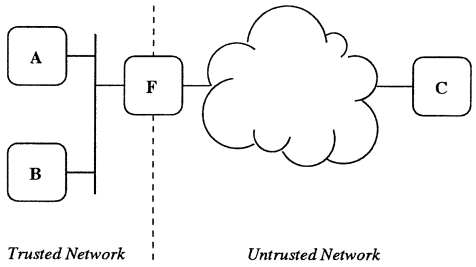

Fig. 5. Trusted network behind a firewall.

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:

$$\texttt{fun wrapper(c, sign)} = (\texttt{zeroRB()}; \texttt{authEval(c, sign)}).$$

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 $\mathrm{authEval}$. 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" [29]. We could extend our framework to allow $\mathrm{authEval}$ to take a public key rather than a signature, accepting the identity of the key if and only if the principal whose key it is has at most a thin set in the node policy (as is the case for the guest). In Section VII-C we present results for the more naive case (which approximates the performance of the VPN we describe next), and can derive the performance for the more optimized case.

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 Fig. 6.

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.

B. Active Virtual Private Network

A virtual private network (VPN) is essentially two or more trusted networks connected by secure "tunnels" across untrusted links. These

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

Fig. 6. Host QCM program.

tunnels are made secure by encryption, such that when a packet leaves one trusted network and enters the tunnel it is encrypted, and then is decrypted upon exiting the tunnel and reentering a trusted network. In the Internet, IPSec [42] may be used to implement VPNs.

This idea is depicted in Fig. 7. Here we have two trusted networks, consisting of nodes A, B, and $P1$ for network I, and C, D, and $P2$ for network II, connected by a secure tunnel across an untrusted network. A packet originating in *network I* destined for *network II* is encrypted at its firewall $P1$, sent across the untrusted network, decrypted at the peer firewall $P2$, and then finally delivered to its ultimate destination in *network II*.

We can implement this idea in PLANet as follows. The VPN would be set up by having nodes $P1$ and $P2$ mutually authenticate, resulting in a shared secret. Say that node A sends a packet to node D. As it exits *network I*, the firewall $P1$ intercepts, encapsulates, signs, and forward the packet to the other network.[7] In particular, $P1$ first extracts the representation of the packet's chunk (call it c) and creates a tunneling chunk $|\texttt{fwd}|(\texttt{c}, \texttt{D}), \texttt{sign}$, where D is the original destination address within the remote network, and fwd refers to the following PLAN code added to the packet:

$$\texttt{fun fwd(c : chunk, dest : host)} =$$
$$\texttt{OnRemote(c, dest, getRB(), defaultRoute)}$$

Next, it signs the tunneling chunk using the secret it shares with $P2$, and replaces the packet's original chunk c with the chunk $|\texttt{authEval}|(|\texttt{fwd}|(\texttt{c}, D), \texttt{sign})$ where sign is the newly created signature. Finally, it alters the destination of the packet to be $P2$ rather the D, and sends it onward.

When the packet arrives at the peer firewall $P2$, it will perform the $\mathrm{authEval}$. This will grant the packet the full privileges of the remote network, with which it sends the original chunk c to the originally intended destination D. Since $P2$ is acting as a firewall described above, had the packet not authenticated in this way it would have had its privileges reduced upon entering the remote network. This illustrates nicely how our authentication and authorization framework can form the common ground of the dual notions of firewall and VPN.

C. Performance Analysis

We analyze the performance of our active firewall by comparing a filtered and nonfiltered ping. In both cases, the initiating host lies in the trusted network and is pinging a node in the untrusted network. We do not present performance measurements for our VPN application because its results are essentially the same as the firewall.

The PLAN code for ping is illustrated in Fig. 2. Our analysis examines the additional cost to elapsed time and packet size.[8] For our

[7]Normally, a VPN would encrypt, rather than sign, outgoing packets. This can be done as well in our framework given an encryption service that requires authorization before it may be used. However, we elide this detail to keep the presentation simple.

[8]The reader may note that the numbers reported here are slightly different from those reported in [1]; this is due to changes made to the PLANet implementation.

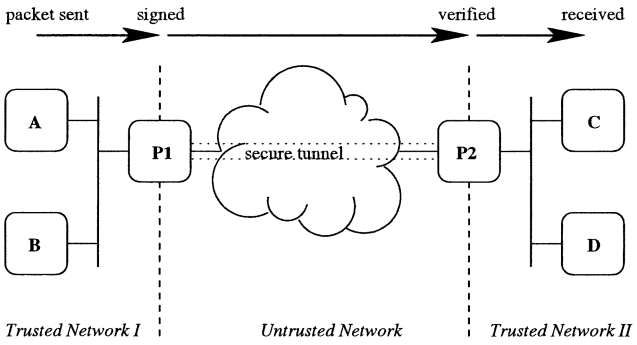

Fig. 7. VPN consisting of two trusted networks connected by a secure tunnel.

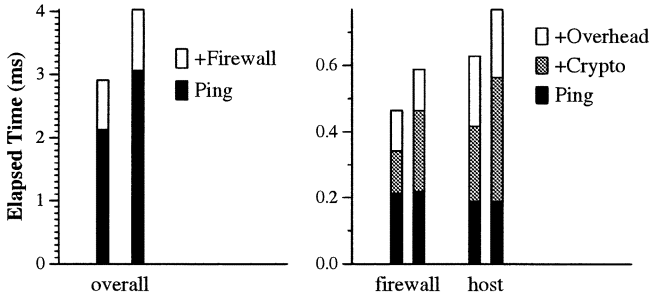

Fig. 8. Ping elapsed time with and without the firewall. The left bar of each pair is with a 0-byte payload, and the right bar is for maximally-sized (1500 bytes) packets.

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 Section VII-A, 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.

Fig. 8 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 additional 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.

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

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

Fig. 9. 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, thus the maximum payload in the $+firewall$ case is slightly smaller.

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%. Note that the cryptographic operations cannot be removed in the case of the VPN.

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 remarshal 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 fairly 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 [9], [28].

Space Overhead: The firewall also imposes a space cost due to the extra code and signature that is attached to the incoming packets Fig. 9 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 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" [49]. 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 [8].

VIII. Related Work

Securing active networks [50] has demanded three major research thrusts.

1) 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.
2) 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 [51].

   The SANE [40] architecture is part of the SwitchWare Project [5] at the University of Pennsylvania. SANE provides the ability to securely bootstrap [52] an active node, and authentication and naming services for code that is loaded. The main differences between this work and SANE are that 1) we can depend on a provably safe language (PLAN) for those packets that do not require special privileges, and 2) we have scalably built a means for controlling service usage via trust management. Furthermore, programming constructs available in PLAN (e.g., chunks) considerably ease the task of implementing security abstractions. SANE was developed in conjunction with the ALIEN architecture [34], which (like us) employs namespace-based security and strong typing. Taken together, these techniques prevent active code from calling functions or accessing data even in a shared address space. Similar approaches have been taken in [20], [53], [54]. Other language-based protection schemes can be found in [20]–[24].

   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 [31], the performance penalty of frequent cryptographic operations can be substantial.
3) 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" [55], examples of which include RCANE [56], JanOS [57], AMP [55], and Scout [58]. These systems manage resources which may be used by a safe programming environment in service invocations, including management of resources used concurrently by multiple programming environments.

   For example, the extensions of SANE described in the paper by Alexander *et al.* [31] to manage soft real-time streams showed that a SANE-based front-end could provide access control for services and resources with associated time bounds. This was

implemented with a low level scheduler in the supporting operating system, and would enable, for example, supporting "quality of service"-like features such as priorities and construction of differentiated-service network architectures. A particularly interesting coupling with the work reported here would be bandwidth control at the active firewall authorized by credentials.

0ur use of $\mathrm{authEval}$ resembles Java stack inspection (JSI) [59], [60]. 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 $\mathrm{enablePrivilege}$. It would be interesting to apply the same approach to nested $\mathrm{authEval}$ calls to provide the same sort of security.

The SPIN [20] Project investigated the construction of extensible operating system kernels, with the idea that type-safe Modula-3 code could be loaded into an operating system for reasons of performance or access to resources. SPIN's dynamic binding infrastructure [61] provides mechanisms with which one could implement our approach to service security. In particular, loaded modules can be linked against a restricted interface, and calls to sensitive functions can be interposed with "guards," which could perform policy-based parameterization. Grimm and Bershad [62] focus on separation of policy and enforcement, and control abstractions crossing protection domains with redirections of procedure or method invocations. SPIN's extensibility is targeted at a workstation environment rather than the network service enhancement environment of PLAN, and is thus less concerned with scalable, distributed policies[9] than Secure PLAN. SPIN and other approaches to language-based security, like the J-Kernel [22] and KaffeOS [27], are quite concerned with resource control of untrusted code. The resource-limited nature of PLAN allows us to avoid the thorny issues that arise here.

Perhaps the most closely related architecture, albeit one instantiated with traditional operating systems mechanisms such as tagged objects (where the tags are associated with permissions) is the "Sub-Operating System" (SubOS) approach of Ioannidis, *et al.* [63]. In that system, there is fine-grained access control of arriving code under control of a nonremovable identifier attached to objects (such as code and data) that arrive over a network. Three key distinctions in our system, including the active firewall, relative to SubOS are 1) the active firewall actively rewrites code to reflect restrictions on it, rather than attaching tags which must be further resolved against a privilege set; 2) the rewrites are performed for any and all trusted hosts, ensuring that an improperly configured element cannot mistakenly execute active code considered dangerous; and 3) privilege can be increased *or* decreased in our system, unlike the SubOS, where privilege is always decreased relative to the executing user, the main goal being the control of locally executed active content.

IX. Conclusions

The Secure PLAN architecture couples limited but safe active packets with general-purpose, but potentially unsafe service routines. The architecture has two major advantages. First, packets that 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. Recent work by Moore [64] characterizes what constitutes a safe service in light of the model discussed in Section III-C. proof-carrying code [65], [66] is one way to certify safety in low-level code, so we would hope its techniques could ensure safety as Moore defines it. A related certification technique uses dependent types to prove that services consume a bounded amount of time and/or space [67].

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 requires per-invocation checks. We have sought to enable scalability by carefully encoding the namespace-based policy, and by using a decentralized trust management system [3].

The active firewall and active VPN are novel applications resulting from our approach. 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 in a traditional firewall architecture. Our experimental implementation has demonstrated such an approach can have acceptable performance.

Our applications also make a novel use of active networking technology. The system exploits the ability of PLAN to manipulate "chunks" to build a far more flexible security gateway for network services. In particular, the combination of trust management policy and namespace security allows extremely fine-grained control of permitted operations for remote users. One might view the active firewall as providing a selectable continuum of access to services rather than merely simple actions such as `pass, drop or log`. It is thus in the spirit of active networking: flexibility and security, with high performance.

[9]In fact, our results include almost all of the future work suggested in the Grimm and Bershad paper, who foresaw the need for policy specification languages, distributed authentication, and high performance for access control operations.

Acknowledgment

The authors would like to thank S. Nettles, J. Moore, and T. Jim for helpful discussions concerning this work, and the anonymous referees for providing useful feedback. We would also like to thank T. Jim for providing the PLAN-based implementation of QCM.

References

[1] M. Hicks, J. T. Moore, D. S. Alexander, C. A. Gunter, and S. Nettles, "PLANet: An active internetwork," in *Proc. 18th IEEE Computer Communication Societ INFOCOM Conf.*, 1999, pp. 1124–1133.

[2] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles, "PLAN: A packet language for active networks," in *Proc. 3rd ACM SIGPLAN Int. Conf. Functional Programming Languages*, 1998, pp. 86–93.

[3] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management," in *Proc. 17th Symp. Security Privacy.*, Los Alamitos, CA, 1996, pp. 164–173.

[4] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. I. Wetherall, and G. J. Minden, "A survey of active network research," *IEEE Commun. Mag.*, pp. 80–86, Jan. 1997.

[5] D. S. Alexander, W. A. Arbaugh, M. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and I. M. Smith, "The switchware active network architecture," *IEEE Network Mag.*, vol. 12, pp. 29–36, 1998.

[6] D. I. Wetherall, J. Guttag, and D. L. Tennenhouse, "ANTS: A toolkit for building and dynamically deploying network protocols," in Proc. IEEE Conf. Open Architectures Network Programming, Los Alamitos, CA, Apr. 1998.

[7] D. Wetherall, "Active network vision and reality: Lessons from a capsule-based system," in *Proc. 17th Symp. Operating Systems Principles*, Kiawah Island, SC, Dec. 1999, pp. 64–79.

[8] M. Hicks, J. T. Moore, D. Wetherall, and S. Nettles, "Experiences with capsule-based active networking," in DARPA Active Networks Conf. Exposition, May 2002.

[9] J. T. Moore, M. Hicks, and S. Nettles, "Practical programmable packets," in *Proc. 20th IEEE Computer Communication Society INFOCOM Conf.*, Apr. 2001, pp. 41–50.

[10] E. L. Nygren, "The Design and Implementation of a High-Perfonnance Active Network Node," M.A. thesis, Mass. Inst. Technol., Cambridge, MA, 1998.

[11] C. A. Gunter and T. Jim, "Policy-directed certificate retrieval," *Softw.-Pract. Exp.*, vol. 30, no. 15, pp. 1609–1640, 2000.

[12] J. Ioannidis and S. M. Bellovin, "Implementing pushback: Router-based defense against DDoS attacks," in Proc. Network Distributed System Security Symp. (NDSS), Feb. 2002.

[13] A. D. Keromytis, V. Misra, and D. Rubenstein, "SOS: Secure overlay services," in *Proc. ACM SIGCOMM Conf.*, August 2002, pp. 61–72.

[14] S. Savage, D. Wetherall, A. Karlia, and T. Anderson, "Practical network support for IP traceback," in *Proc. ACM SIGCOMM Conf.*, Aug. 2000, pp. 295–306.

[15] D. Dean, M. Franklin, and A. Stubblefield, "An algebraic approach to IP ttaeeback," in *Proc. Network Dsitributed System Security Symp. (NDSS)*, Feb. 2001, pp. 3–12.

[16] X. Leroy. (2002) The Objective Caml System, Release 3.05. Instinit Nat. Rec. Informntique Automatique (INRIA). [Online]. Available: http://caml.inria.fr

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

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

[19] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proc. USENIX Annu. Technical Conf.*, Monterey, CA, June 2002, pp. 275–288.

[20] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Pitszcynski, D. lietket, S. Eggers, and C. Chambers, "Extensibility, safety and performance in the spin operating system," in *Proc. 15th Symp. Operating Sytems Principles*, Dec. 1995, pp. 267–284.

[21] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Trans. Comput. Syst.*, vol. 12, no. 4, pp. 271–307, Nov. 1994.

[22] C. Hawblitzel, C. Chang, and G. Czajkowski, "Implementing multiple protection domains in java," in *Proc. 1998 USENIX Annu. Technical Conf.*, June 1998, pp. 259–270.

[23] J. Y. Levy, J. K. Ousterhout, and B. B. Welch, "The Safe-Tcl security model," in *Proc. 1998 USENIX Annu. Technical Conf.*, June 1998, pp. 271–282.

[24] J. Moore. (1998) Mobile Code Security Techniques. Univ. Pennsylvania, Philadelphia. [Online]. Available: http://www.cis.upenn.edu/~jonm/papers/cis700.ps

[25] B. Schwartz, W. Zhou, A. W. Jackson, W. T. Strayer, D. Rockwell, and C. Partridge, "Smart packets for active networks," in *Proc. IEEE Conf. Open Architectures Network Programming*, 1999, pp. 90–97.

[26] M. Hicks. (1998) PLAN System Security. Dept. Comput. Infornm. Sci., Univ. Pennsylvania, Philadelphia. [Online]. Available: http://www.cis.upenn.edu/~switchware/papers/plan _security.ps

[27] G. Back, W. C. Hsieh, and J. Lepreau, "Processes in kaffeOS: Isolation, resource management, and sharing in java," in 4th USENIX Symp. Operating Systems Design Implementation, San Diego, CA, Oct. 2000.

[28] M. Hicks, J. T. Moore, and S. Nettles, "Compiling PLAN to SNAP," in *Proc. 3rd Int. Working Conf. Active Networks*, vol. 2207, I. W. Marshall, S. Nettles, and N. Wakamiya, Eds., Oct. 2001, Lecture notes in Computer Science, pp. 134–151.

[29] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proc. IEEE*, vol. 63, pp. 1278–1308, Sept. 1975.

[30] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, S. Muir, and J. M. Smith, "Secure quality of service handling (SQoSH)," *IEEE Commun.*, vol. 38, pp. 106–112, Apr. 2000.

[31] 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," *J. Commun.*, vol. 3, no. 1, pp. 4–18, Mar. 2001.

[32] 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 J. Selected Areas Commun.*, vol. 14, no. 7, pp. 1280–1297, Sept. 1996.

[33] K. G. Anagnostakis, M. W. Hicks, S. Ioannidis, A. D. Keromytis, and J. M. Smith, "Scalable resource control in active networks," in *Proc. 2nd Int. Working Conference Active Networks*, vol. 1942, H. Yashuda, Ed., Oct. 2000, pp. 343–358.

[34] D. S. Alexander, "ALIEN: A Generalized Computing Model of Active Networks," Ph.D. dissertation, Univ. Pennsylvania, Philadelphia, 1998.

[35] F. Rouaix, "A web navigator with applets in caml," in *Proc. 5th Int. World Wide Web Conf. Compater Networks Telecommunications Networking*, vol. 28, May 1996, pp. 1365–1371.

[36] "Data Encryption Standard," U.S. Dept. Commerce, Tech. Rep. FIPS-46, 1977.

[37] "PKCS #1: RSA Encryption Standard," R. Laboratories, version 1.5 ed., 1993.

[38] "Digital Signature Standard," U.S. Department of Commerce, Tech. Rep. FIPS-186, 1994.

[39] "X.509: The Directory Authentication Framework," Int. Telecommun. Union, Geneva, Switzerland, CCITT, 1989.

[40] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith, "A secure active network environment architecture: Realization in switchware," *IEEE Network Mag.*, vol. 12, pp. 37–45, 1998.

[41] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," IETF, Tech. Rep. RFC 2104, 1997.

[42] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol,", IETF Tech. Rep. RFC 2401., 1998.

[43] W. Diffie, P. van Oorschot, and M. Wiener, "Authentication and authenticated key exchanges," *Designs, Codes Cryptog.*, vol. 2, pp. 107–125, 1992.

[44] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inform. Theory*, vol. IT-22, pp. 644–654, Nov 1976.

[45] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, "Kerberos authentication and authorization system," in Project Athena Technical Plan, Dec. 1987, Section E.2.l.

[46] L. Gong, "Efficient network authentication protocols: Lower bounds and optimal implementations," *Distrib. Comput.*, vol. 9, no. 3, pp. 131–145, 1995.

[47] M. W. Hicks. (2001) PLAN Security Guide. [Online]. Available: http://www.cis.upenn.edu/~switchware/PLAN/ docs-ocaml/security.ps

[48] M. Blaze, J. Feigenbaum, J. loannidis, and A. Keromytis, "The role of trust management in distributed systems security," in *Secure internet Programming*. New York: Springer-Verlag, 1999, vol. 1603.

[49] J. Mogul and S. Deering, "Path MTU Discovery," IETF, Tech. Rep. EEC 1191, 1990.

[50] (1998) Security Architecture for Active Nets. [Online]. Available: http://www.ittc.uksns.edu/ansecure/0079.html

[51] S. Murphy, E. Lewis, R. Watson, and R. Yee, "Strong security for active networks," in *Proc. IEEE Conf. Open Architectures Network Programming.*, Apr. 2001, pp. 63–70.

[52] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith, "Automated recovery in a secure bootstrap process," in *Proc. Network Distributed System Security Symp.*, Mar. 1998, pp. 155–167.

[53] X. Leroy and F. Rouaix, "Security properties of typed applets," in *Secure internet Programming*. New York: Springer-Verlag, 1999, vol. 1603.

[54] T. von Eicken, "J-Kernel a capability bated operating system for java," in *Secure Internet Programming*. New York: Springer-Verlag, 1999, vol. 1603.

[55] L. Peterson, V. Gottlieb, M. Hibler, P. Tullman, J. Lepreau, S. Schwab, H. Dandekar, A. Purtell, and J. Hartman, "An OS interface for active routers," *IEEE J. Selct. Areas Commun.*, vol. 19, pp. 473–487, Mar. 2001.

[56] P. Menage, "RCANE: A resource controlled framework for active network services," in *Proc. 1st Int. Workshop Active Networks*, vol. 1653, S. Covaci, Ed., Springer-Verlag, June 1999.

[57] P. Tullmann, M. Hibler, and J. Lepreau, "Janos: a Java-oriented OS for active network nodes," *IEEE J. Select. Areas Commun.*, vol. 19, no. 3, Mar. 2001.

[58] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hastman, "Scout: A Conmmunications-Oriented Operating System," Depart. Comput. Sci., Univ. Arizona, Tucson, Tech. Rep, 1994.

[59] C. Fournet and A. Gordon, "Stack inspection: Theory and variants," in *Proc. ACM Symp. Principles Programming Languages*, Jan. 2002.

[60] D. S. Wallach and E. W. Felten, "Understanding java stack inspection," in *Proc. IEEE Symp. Security Privacy*, May 1998, pp. 52–63.

[61] P. Pardyak and B. N. Bershad, "Dynamic binding for an extensible system," in *Proc. USENIX Symp. Operating Systems Design Implementation* , 1996, pp. 201–212.

[62] R. Grimm and B. N. Bershad, "Providing policy-neutral and transparent access control in extensible systems," in *Secure Internet Programming*. New York: Springer-Verlag, 1999, vol. 1603, pp. 317–338.

[63] S. Ioannidis, S. M. Bellovin, and J. M. Smith, "Sub-operating systems: A new approach to application security," in *10th ACM SIGOPS Eur. Workshop*, Sept. 2002.

[64] J. T. Moore, "Practical Active Packets," Ph.D. dissertation, Univ. Pennsylvania, Philadelphia, 2002.

[65] G. C. Necula, "Proof-carrying code," in *Proc. 24th Annu. ACM S1GPL4N-SIGACT Symp. Principles Programming Languages.*, New York, Jan. 1997, pp. 106–119.

[66] G. C. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *Proc. USENIX Symp. Operating Systems Design and Implementation*, 1996, USENIX, pp. 229–243.

[67] K. Crary and S. Weirich, "Resource bound certification," in *Symp. Principles Programming Languages*, 2000, pp. 184–198.

[68] M. Hicks and A. D. Keromytis, "A secure plan," in *Proc. 1st Int. Workshop Active Networks*, vol. 1653, S. Covnci, Ed., Springer-Verlag, June 1999, pp. 307–314.

[69] M. Hicks, A. D. Keromytis, and J. M. Smith, "A secure plan (extended version)," in *Proc. DARPA Active Networks Conf. Exposition.*, May 2002, pp. 224–237.

[70] J. Vitek and C. Jensen, *Secure Internet Programming: Security Issues for Mobile And Distributed Objects*. New York: Springer-Verlag, 1999, vol. 1603.