# A Secure Plan

Michael Hicks[1] and Angelos D. Keromytis[1]

Distributed Systems Lab
CIS Department, University of Pennsylvania
200 S. 33rd Str., Philadelphia, PA 19104, USA
{mwh,angelos}@dsl.cis.upenn.edu

**Abstract.** Active Networks promise greater flexibility than current networks, but threaten safety and security by virtue of their programmability. In this paper, we describe the design and implementation of a security architecture for the active network *PLANet* [HMA+99]. Security is obtained with a two-level architecture that combines a functionally restricted packet language, PLAN [HKM+98], with an environment of general-purpose service routines governed by *trust management* [BFL96]. In particular, we employ a technique which expands or contracts a packet's service environment based on its level of privilege, termed *namespace-based security*. As an application of our security architecture, we present the design and implementation of an active-network firewall. We find that the addition of the firewall imposes an approximately 30% latency overhead and as little as a 6.7% space overhead to incoming packets.

## 1 Introduction

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

At the University of Pennsylvania, we have developed an Active Internetwork called *PLANet* [HMA+99]. PLANet's node architecture consists of two levels: the *PLAN level* and the *service level*. All programs at the PLAN level reside in the messages, or packets, that are sent between the nodes of the system. These programs are written in PLAN, the Packet Language for Active Networks [HKM+98]. PLAN programs are simple by nature, and serve to 'glue' together service level programs. In contrast, service level programs (or *service routines*), reside at each node and are invoked by PLAN programs evaluating there. Service routines are general-purpose and may be dynamically loaded across the network [AAH+98].

The current Internet (IP [Pos81] and its supporting protocols) allows any user with a network connection to have some basic services. In addition to basic packet delivery provided by IP, basic information services like DNS, `finger`, and `whois`, and protocols like HTTP, FTP, SMTP, and so forth are provided. Similarly, a goal of PLANet is to allow any user of the network to have access to basic services; these services should naturally include some 'activeness.' This goal implies that some functionality, like packet delivery in the current Internet, should not mandate authentication; in PLANet, *we allow all 'pure' PLAN programs to run unauthenticated*. A PLAN program is considered 'pure' if it only makes calls to service routines considered safe; for example, determining the name of the current host is a safe operation, while updating the host's router table is not. Which routines are to be exported (*i.e.,* made available for invocation by unauthenticated PLAN programs) is a matter of local policy. For example, a router in the center of the network may expose very few service routines, while an end-host might provide a more liberal execution environment. We expect that in a large Active Network, a number of basic 'core services' would be ubiquitous.

This paper presents the design and implementation of a security architecture in PLANet. We begin by presenting a description of our architecture, after describing the attacks it protects against. We then follow with a description of the implementation of this architecture in PLANet. While some attention is given to security at the PLAN level, the majority of the paper deals with security at the more dangerous service level. Following a brief discussion of PLAN and its relevant characteristics, we present possible methods of security management and the one we have chosen to implement: *namespace-based security*. We describe how we enable authentication, and manage relevant security information, such as which service routines are available to which principals, using QCM (Query Certificate Manager [GJ98]). We then demonstrate how we have used our system to implement a simple firewall that 'filters' active packets. Finally, we present some related work.

## 2 Architecture

In this section we present the general architecture of our security mechanisms. We first describe the threats that an active network must secure against. We then describe the structure of our security architecture for handling these threats.

### 2.1 Threat Model

The two major threats to any active networking system are to the *public resources* of the system: the CPU, memory, and network; and to the *contents* of the system: the packets themselves and the information stored on routers. The more specific forms these threats might take are outlined below.

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

- **Protection**. Programs should be protected from interference by other programs. In particular, one program should not be able to read or write data private to another program without authorization, either while the packet program is in transit or when it is running. This property implies program isolation.
- **Spoofing**. As mentioned, we would like to allot greater privilege to some packets, such as those associated with the node administrator. Therefore, it is important that these packets be properly authenticated, and that no impersonation attacks be possible.

### 2.2 Architecture

As already described in the Section 1, we partition the problem of defending against these attacks into the PLAN level and the service level, using different mechanisms at each level. At the PLAN level, security is obtained via functional restriction: the nature of the PLAN language and the 'core services' made available to all PLAN programs prevent attacks, particularly denial-of-service and protection attacks, from being formulated.

At the service level, we make use of an authorization system to govern access to 'unsafe services.' Such services (*e.g.,* network management) are necessary for the operation of the active node, but should not be made available to general users. Our architecture associates with each principal (user) a set of service routines and policies that are allowed at his level of privilege. The policies are enforced and the routines are made available after the user is successfully authorized. This architecture is illustrated in Figure 1.

We flesh out the details of this architecture in the next two sections. We describe PLAN's security properties in the next section, and then present our service management methodology.

## 3 PLAN

PLAN [HKM$^+$98] is a small language that has elements in common with Haskell [Has], Scheme [Sch], and ML [MTH90, Ler]. It differs most importantly from these in that it includes a primitive `OnRemote` (among others) for evaluating an expression at a remote node. Invoking `OnRemote` will result in a newly spawned packet. PLAN also has some desirable security properties, which we will mention below.

#### PLAN's Security Properties

PLAN is the foundation of PLANet's security for the simple reason that PLAN programs may be run without authentication. By design, the language has properties that prevent some attacks; that way all programs written in PLAN are 'safe.' The language is resource- and expression-limited, thus preventing CPU and memory denial-of-service attacks. For example, all PLAN programs are

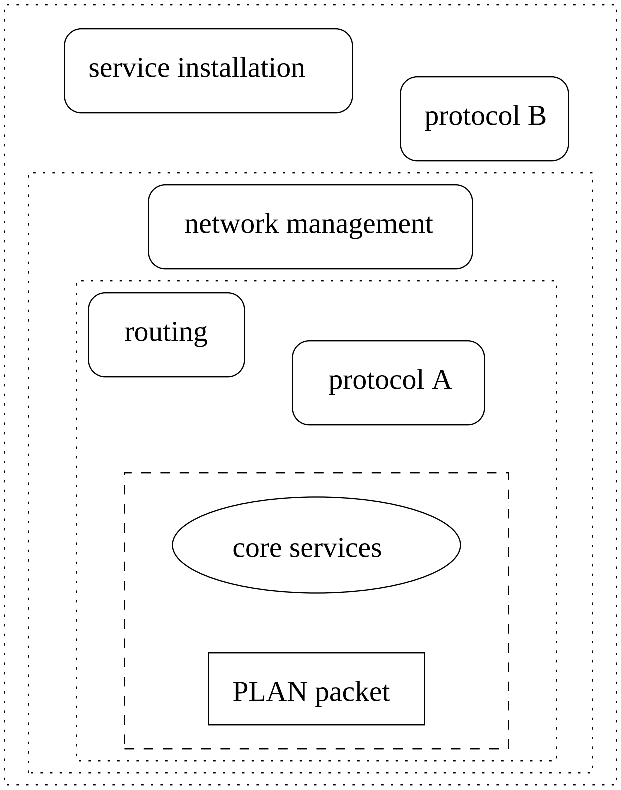


**Fig. 1. PLANet's security architecture.** The contents of the dashed box are available to all incoming packets, while the dotted boxes encapsulate service packages available only to select users.

guaranteed to terminate[1], since PLAN does not provide a means to express non-fixed-length iteration or recursion. Additionally, PLAN programs are isolated from one another since there is no means of direct communication among them, and because the language's strong typing and garbage collection prevent indirect means, such as through pointer swizzling or buffer overflows.

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 new packet is sent. Therefore, the number of hops that a PLAN program or any of its progeny may take are limited by the initial value of this counter. This mimics the functionality of the IP "Time To Live" (TTL) field.

The safety of a PLAN program is predicated on the safety of the services it calls. If a service allows a program to, for example, perform general recursion,

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

then denial-of-service attacks may be launched. For this reason, it is of critical importance that a system for managing the services be in place. We discuss our approach, among others, of using trust management to manage namespaces in the following section.

## 4 Service Security via Trust Management

Because of their general-purpose nature, service routines may perform actions which, if exploited, could be used to mount an attack. A radical approach to this problem would be to prevent *any* service routine from being installed that could potentially harm the node. However, this would preclude the addition of service routines—for example, network management operations—that should be available to *trusted* users. We thus employ security mechanisms which allow authorized programs to access potentially unsafe service routines.

### 4.1 Trust Management

In determining the form of these security mechanisms, we arrived at some basic requirements:

- The mechanisms should be simple to understand and employ, and should not rely on the existence of a widely-available infrastructure.
- Security policies should be modifiable as needed, by the proper administrative entities, while the system is operating.
- Policy mechanisms should be flexible enough to address current as well as future application needs, at least to the extend that those can be predicted.
- Security mechanisms must scale to support increasing numbers of different applications, users, administrative entities, and their trust relations.

To meet these requirements, our service security relies on *trust management* [BFL96, BFS98].

Trust management assigns some level of privilege (or trust) to a user, or *principal*[2], of the system. In particular, if a running PLAN program wishes to invoke a privileged service routine or alter a service parameter, the principal associated with the packet must be authenticated, and then the operation must be authorized. If either step fails, the operation is denied. We consider the question of policy and mechanism for authorization below; details about our particular implementation of authentication and authorization are presented in the next section.

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

### 4.2 Policy and Mechanism

Before applying trust management, we must consider what sorts of policies we would like to express, and what particular mechanisms we shall use to enforce these policies.

As mentioned, we essentially want to be able to encode our policy as an ACL which indicates what services, above the core services, are available to certain users. We also find it convenient to indicate which services should be *subtracted* from the default environment for a particular user; this will be motivated in Section 6. For purposes of simplicity and scalability, we choose to map *sets* of principals to *sets* of services. We would also 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 $p$. In keeping with our requirements, this ACL should scale to include many nodes, principals, services; and should be alterable on-the-fly.

Furthermore, we want to specify not only *whether* a service routine may be invoked, but *how* it may be used. For example, a *resident state* service which allows packets to leave state on the routers might apportion different amounts of space to different users. We should also be able to specify general resource usage parameters, such as CPU and memory use. For more discussion on policy mechanisms, see [BFIK99].

To enforce security policy we require strong principal authentication, and use a *policy manager* on every node; more details are given in the next section. We also must decide when authentication and authorization will take place. Two possible approaches are:

- Perform the check at service-routine invocation time. Each time a service routine is called from PLAN, a check is made to see if the 'current principal' is allowed to access the service. If not, an exception is raised. This is essentially a more elaborate variation of the Unix system-call mechanism.
  A drawback of this approach is that *all* service calls must check the execution policy; we would prefer not to impose this additional overhead for unauthenticated programs. Of course, we could add the code for service checks only to those routines that might be subject to policy restriction. This might be applicable to the set of standard, core services, or to services that do not require policy-based parameterization.
- Perform the check at some predetermined time, such as at packet link time. The advantage of this approach is that only those packets which require authorization will have to pay for it. On the other hand, this makes it difficult to parameterize services based on policy.

We employ the middle ground of these two approaches. Packets must authenticate themselves at some point before accessing privileged services; at this time, the appropriate services are added to (or subtracted from) the packet's current service symbol table. We call this approach *namespace-based security.* Since PLAN is strongly typed and looks up services on an as-needed basis, programs are incapable of invoking code outside of this updated table.

Additionally, we allow those services which may require policy-based parameterization to query the policy manager at the appropriate times during their execution. For example, the resident state service mentioned above would query the local policy to determine how much memory the current principal was allowed to occupy.

We feel there are some compelling advantages to this approach. First, namespace-based policies are simple to formulate and easy to change. Second, because namespace-based security is centrally-administered, individual service routines may be written without concern for security, and policies may change dynamically without worry of inconsistency. Furthermore, unauthenticated programs may access the core services without additional performance penalty. Finally, because namespace-based security is not by itself sufficient, we allow services to formulate their own usage policies.

There is still some work to be done in our current system. Namespace-based security only applies to PLAN service routine calls, not calls between service routines. This is slightly more difficult, but entirely possible, since Caml provides a mechanism which may be used to implement namespace-based security: *module thinning*. The use of module thinning has been explored for active networks in [Ale98] and for mobile agent systems in [LOW98]. Also, while we have experimented with mechanisms for enforcing resource usage, we have yet to arrive at ones that are sufficiently lightweight. Relevant details may be found in [Hic98].

## 5 Implementation

In this section, we describe the mechanisms used by PLAN programs for authentication and authorization.

### 5.1 Authentication

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

A chunk (or **c**ode **hunk**) may be thought of as a function that is waiting to be applied. In PLAN, chunks are first-class—they may be manipulated as data—and consist internally of some PLAN code, a function name, and a list of values to be used as arguments during the application. A chunk is typically used as an argument to `OnRemote` to specify some code to evaluate remotely. A chunk may also be evaluated locally by passing it to the `eval` service, which resolves the function name with the current environment, performs the application, and returns the result.

We have added an additional service called `authEval` which takes as arguments a chunk, a digital signature, and a public key. `authEval` verifies the signature against the binary representation of the chunk. If successful, the chunk is evaluated; otherwise, an exception is raised. The authenticated principal is associated with its chunk during evaluation. This is implemented by mapping the principal to the current thread identifier. Services invoked as a result of chunk evaluation can do their own authorization, by obtaining the principal public key from the authentication service. Because a caller's thread identifier cannot be forged, this provides a safe way to track a principal without worry that some malicious service will change the associated principal after the authentication phase.

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

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

To address these problems, we make use of a protocol in which a user and a node authenticate each other and generate a shared secret for future communications and an identifier for that secret (named Security Parameters Index, or "SPI"). These exchange identifiers are used at the conclusion of the protocol to create SPI's for future use of the shared secret. The protocol is essentially a variation of the Station-to-Station protocol [DvOW92]; more details may be found in [AAKS98, AKS98].

Once the protocol is completed, parties may use the shared secret to authenticate via HMAC-SHA1 [KBC97], in a way similar to that used in the IPsec [Atk95] 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, a sliding-window scheme may additionally be used. We reflect the use of HMAC-SHA1 in PLAN by altering the signature of `authEval` to take a chunk and a tuple consisting of the SPI, the counter, and the HMAC signature over all of the previously mentioned items.

### 5.2 Authorization

As our policy manager, we have chosen to use the Query Certificate Manager (QCM) [GJ98], which provides comprehensive security credential location and retrieval services, employing a distributed ACL. While in this paper we are making use of QCM, our architecture is designed so that other policy managers be used instead. In particular, we are also experimenting with the KeyNote [BFIK98, BFIK99] trust-management system.

QCM is used to specify the services to be added to or subtracted from the default service-environment by associating certain *thicken* and *thin* sets of services with a principal or set of principals. Once a principal has been authenticated,

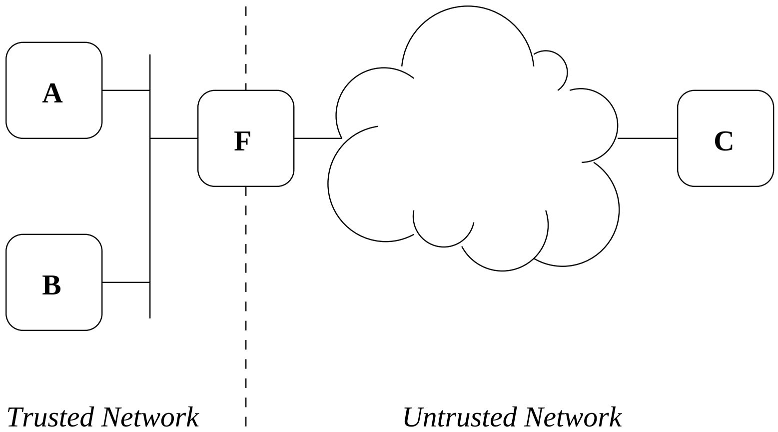


**Fig. 2. A trusted network behind a firewall.**

these sets are used to modify the default environment. The resulting service environment is then used during subsequent chunk evaluation. As an optimization, we can cache this environment for future reference, thus avoiding repeated invocations of QCM and reconstructions of the environment.

A key advantage of using QCM is that it can be used for more than just specifying sets of principals on a per-node basis. In particular, sets described in a distributed manner impose no additional query complexity. For example, a node $A$ may define a set which partially resides at another node $B$:

`l` = $\{\ p_1,\ p_2,\ \ldots\ ,\ p_n\ \}$ `union` $B\$m$

If the authorization service on $A$ makes a membership test on set `l`, QCM will automatically query $B$ if necessary. QCM may also make use of certificates, which are signed assertions about set relationships, to short-circuit remote queries. These may be passed as additional arguments to `authEval`, or may be obtained during node-node authentication. This allows QCM to implement both *push*- and *pull*-based information-retrieval.

## 6 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 Figure 2. In this section, we describe how our security architecture can be used to implement an *active* firewall.

### 6.1 Implementation

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

Our approach is that rather than filter packets at the firewall, we associate with them a *thinned* service environment in which any potentially harmful services are removed. The packets may then be evaluated inside the trusted network using only those services. While this may seem to contradict our premise stated in Section 2.2 that the default environment should consist only of 'safe' services, in the context of a trusted Intranet, we would expect that the default privilege allowed to local packets exceeds that of foreign packets. Furthermore, we would not want to impose the overhead of authentication and authorization on local packets in the general case.

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

```
fun wrapper(c:chunk, sign:blob): unit =
  (zeroRB(); authEval(c,sign))
```

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

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

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

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

The *thin* set is defined by the variable `guest_thinned_services` at principal `firewall`. Notice that the *thicken* set is empty. The firewall would provide certificates during node-node authentication that indicate the contents of its

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

**Fig. 3. Host QCM Program**

```
  fun reply(payload:blob):unit =
    print("Success")

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

**Fig. 4. Ping in PLAN**

`guest_thinned_services` variable. Should the firewall policy be updated after initial authentication, the firewall would push certificates to the end host to reflect this change.

### 6.2 Performance Analysis

We analyze the performance of our active firewall by comparing a filtered and non-filtered ping. In both cases, the initiating host lies in the trusted network and is pinging a node in the untrusted network. The PLAN code for ping is illustrated in Figure 4. We examine the additional cost to elapsed time and packet size.[3] For our experimental setup, we directly-connect three machines with 100 Mbit Ethernet, configuring the center machine as the active firewall. Each machine is a 300 MHz Pentium II with 250 MB of memory running Linux 2.0.30. PLANet runs directly on top of Ethernet.

**Time Overhead** As described in the previous section, the addition of the firewall affects the packet processing time on the router and on the host initiating the "ping." While a router would normally just forward any packet it receives, the firewall has to additionally sign and encapsulate packets destined for the trusted network. On the initiating host, normal interpretation of the "reply" packet is

[3] The reader may note that the numbers reported here are slightly different than those reported in [HMA+99]; this is due to additional changes made to the current PLANet implementation.

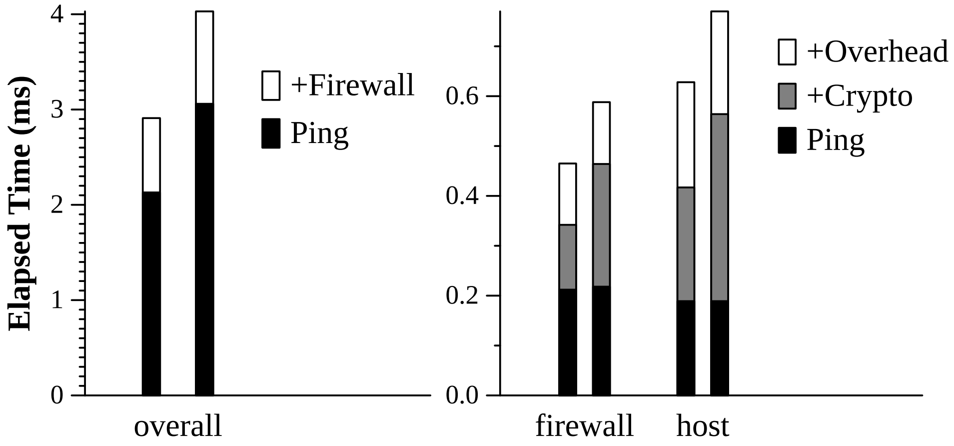


**Fig. 5. Ping elapsed time with and without the firewall**. The left bar of each pair is with a 0-byte payload, and the right bar is for maximally-sized (1500 byte) packets.

further burdened by the need to decapsulate, verify the firewall's signature, and thin the environment.

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

The base ping times for 0-byte and maximal payloads are 2.13 and 3.06 ms, respectively; the firewall adds 27% and 32% of respective overhead to these times (raising them to 2.91 and 4.03 ms). By examining the component costs, we can see that of this overhead, between 1/3 and 1/2 is attributable to signing and verification, based on the packet size. For the firewall, the remaining overhead is due to encapsulation costs (which requires extra marshalling and copying), while for the end-host it is due to decapsulation and the additional interpretation cost of the wrapper code. The time to thin the environment at the end host is negligible because we cache the thinned environment. If we eliminate the cryptographic operations, by the means described earlier, we reduce the end-to-end ping times to 2.58 and 3.38 ms for 0-byte and maximal payload, respectively. This reduces the firewall-induced overhead to 21% and 32%.

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

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

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

There is room for further improvement. The cost of the cryptographic operations (for cases when they are actually needed) could be reduced through parallelism (to improve bandwidth) and special-purpose hardware (to improve both bandwidth and latency). Furthermore, the cost of PLAN interpretation is extremely high; a smarter interpreter would improve both the cost of the basic ping as well as the encapsulated version. We are currently investigating both solutions.

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

The no-payload reply packet is 80 bytes (consisting of code and fixed fields), while the encapsulated version is 181 bytes, for an overhead of 126%. Of the 101 bytes of overhead, 12 bytes are due to the signature. Since the overhead is fixed, its impact is reduced with packet size. Looking at the maximally-sized packet, we see that this 101 bytes only 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). We are currently examining how to best express in PLAN a mechanism similar to "Path MTU Discovery" [MD90]. 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 are currently considering ways in which the participants in a protocol may cache code rather than always transmitting it with the packet. One promising 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.

## 7 Related Work

Research in the area of security for active networks is in its early stages. The SANE [AAKS98] architecture is part of the SwitchWare Project [AAH+98] at the University of Pennsylvania. SANE provides the ability to securely bootstrap [AKFS98] the remainder of the SANE system, and authentication and naming services for code that is loaded. The main difference between this work and SANE lies in that we can depend on a provably safe language (PLAN) for those packets that do not require special privileges. Furthermore, programming constructs available in PLAN (*e.g.,* chunks), considerably ease the task of implementing security abstractions.

SANE is currently used in conjunction with the ALIEN architecture [Ale98]. Security is achieved in ALIEN through a combination of module thinning and type safety. 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 [LR99, BSP+95, vE99]. Other language-based protection schemes can be found in [BSP+95, CLFL94, HCC98, LOW98, Moo98].

Finally, a working group within the Active Networks project has been defining a common security meta-architecture [Mur98]. However, this architecture has not become concrete enough for implementation.

Trust management was first introduced with the PolicyMaker system [BFL96]. It addressed the authorization problem directly, rather than handling the problem indirectly via name-based authentication and access control lists, and it provided an application-independent definition of "proof of compliance" for matching up requests, credentials, and policies. A full description of the system can be found in [BFL96, BFS98], and experience using it in several applications is reported in [BFRS97, LSM97, LMB]. Other trust-management systems are described in [BFIK98, CFL+97]. For an overview of trust management, see [BFIK99].

The Secure PLAN architecture is a hybrid which couples highly-scrutinized active extensions with unauthenticated active packets supported by these extensions. This has two major advantages. First, packets which do not require the computational cost of authentication do not pay it–we believe that such packets will comprise the majority of active network usage. The second advantage is that security analysis, perhaps including validation and verification [NL96, Nec97], can be focused on a small set of routines rather than all possible active programs.

We have demonstrated that our architecture addresses possible threats while still preserving the flexibility and usability of the system. This architecture is based on language safety, authentication, and trust management. We demonstrated the practicality and acceptable performance of our approach experimentally, in the context of an active firewall.

## Acknowledgements

We would like to thank Scott Nettles, Jon Moore, and Jonathan Smith for the various discussions and their comments on earlier versions of this paper.

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

## References

[AAH+98] D. S. Alexander, W. A. Arbaugh, M. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare Active Network Architecture. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 12(3):29–36, 1998.

[AAKS98] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. A Secure Active Network Environment Architecture: Realization in SwitchWare. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 12(3):37–45, 1998.

[AKFS98] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith. Automated Recovery in a Secure Bootstrap Process. In *Proceedings of Network and Distributed System Security Symposium*, pages 155–167. Internet Society, March 1998.

[AKS98] W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. DHCP++: Applying an Efficient Implementation Method for Fail-stop Cryptographic Protocols. In *Proceedings of Global Internet (GlobeCom) '98*, pages 59–65, November 1998.

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

[Atk95] R. Atkinson. Security Architecture for the Internet Protocol. RFC 1825, August 1995.

[BFIK98] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust-Management System, June 1998. Work in Progress: `http://www.cis.upenn.edu/~angelos/keynote.html`.

[BFIK99] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The Role of Trust Management in Distributed Systems Security. In *Secure Internet Programming* [VJ99].

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, Los Alamitos, 1996.

[BFRS97] M. Blaze, J. Feigenbaum, P. Resnick, and M. Strauss. Managing Trust in an Information Labeling System. In *European Transactions on Telecommunications, 8*, pages 491–501, 1997.

[BFS98] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance Checking in the PolicyMaker Trust-Management System. In *Proceedings of the Financial Cryptography '98, Lecture Notes in Computer Science, vol. 1465*, pages 254–274. Springer, Berlin, 1998.

[BSP+95] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of 15th Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[CFL+97] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust Management for Web Applications. In *World Wide Web Journal, 2*, pages 127–139, 1997.

[CLFL94] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. In *ACM Transactions on Computer systems*, November 1994.

[DvOW92] W. Diffie, P.C. van Oorschot, and M.J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[GJ98] Carl A. Gunter and Trevor Jim. Policy-directed certificate retrieval. `http://www.cis.upenn.edu/~qcm`, 1998.

[Has] Haskell: A purely functional language. `http://www.haskell.org`.

[HCC98] C. Hawblitzel, C. Chang, and G. Czajkowski. Implementing Multiple Protection Domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 259–270, June 1998.

[Hic98] Michael Hicks. PLAN System Security. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, April 1998.

[HKM+98] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.

[HMA+99] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles. PLANet: An active internetwork. In *Proceedings of the Eighteenth IEEE Computer and Communication Society INFOCOM Conference*, pages 1124–1133. IEEE, 1999.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. Technical report, IETF RFC 2104, February 1997.

[Ler] Xavier Leroy. The Caml Special Light System (Release 1.10). `http://pauillac.inria.fr/ocaml`.

[LMB] R. Levien, L. McCarthy, and M. Blaze. Transparent Internet E-mail Security. `http://www.cs.umass.edu/~lmccarth/crypto/papers/email.ps`.

[LOW98] J. Y. Levy, J. K. Ousterhout, and B. B. Welch. The Safe-Tcl Security Model. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 271–282, June 1998.

[LR99] X. Leroy and F. Rouaix. Security properties of typed applets. In *Secure Internet Programming* [VJ99].

[LSM97] J. Lacy, J. Snyder, and D. Maher. Music on the Internet and the Intellectual Property Protection Problem. In *Proceedings of the International Symposium on Industrial Electronics*, pages SS77–83. IEEE Press, 1997.

[MD90] J. Mogul and S. Deering. Path MTU Discovery. Internet RFC 1191, November 1990.

[Moo98] J. Moore. Mobile Code Security Techniques. Technical Report MS-CIS-98-28, University of Pennsylvania, May 1998.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Mur98] Security Architecture for Active Nets, June 1998. Draft available at `http://www.ittc.ukans.edu/~ansecure/0079.html`.

[Nec97] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, New York, January 1997.

[NL96] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Second Symposium on Operating System Design and Implementation*, pages 229–243. Usenix, Seattle, 1996.

[Pos81] Jon Postel. Internet Protocol. Internet RFC 791, 1981.

[Sch] Scheme home page. `www-swiss.ai.mit.edu/scheme-home.html`.

[vE99] T. von Eicken. J-Kernel a capability based operating system for Java. In *Secure Internet Programming* [VJ99].

[VJ99] Jan Vitek and Christian Jensen. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. Lecture Notes in Computer Science. Springer-Verlag Inc., New York, NY, USA, 1999.