Performance Implications of Securing Active Networks

D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis and Jonathan M. Smith[*][†][‡]

Abstract

Security is an obvious risk to active networking, as increased flexibility creates numerous opportunities for mischief. The point at which this flexibility is exposed, *e.g.,* through the loading of code into network elements, must therefore be carefully crafted to ensure security.

The Secure Active Network Environment (SANE) architecture provides a secure bootstrap process resulting in a module loader / packet execution environment. As a set of nodes bootstrap, they exchange certificates to permit secure module exchange.

This paper demonstrates that SANE, while exhibiting performance degradation relative to unsecured operation, is able to perform acceptably. We include measurements comparing the loading of an active ping on a secure versus an insecure infrastructure.

1 Introduction

Active Networks is a proposal for packet-switched networks which are programmable, perhaps on a per-user or even a per-packet basis. The more aggressive proposals share the property that "programs" are loaded into network elements on-the-fly, providing rapid dynamic reconfiguration of the network infrastructure.

Operational infrastructures have been produced by several initial efforts such as Active Bridging [ASNS97], ANTS [WGT98] and PLAN [HKM+98]. These efforts are points in a design space which has many dimensions, the most important of which are flexibility, security, usability and performance. Programmable network elements provide flexibility and usability via the choice of programming language and execution environment. For example, the portability and distributed programming support of the Java programming language have made it a popular basis for active networking prototypes. For such systems, improving the performance of Java [HPB+97] is one approach to improving active network performance.

1.1 Security for Active Networks

Security of active networking is a major challenge, as well as a widespread and legitimate cause for concern. One view of Information security can be characterized as getting the right information to the right person at the right place and time. This is the positive statement of a security policy; other security policies might assert what *cannot* occur. The flexibility of an active networking infrastructure, since it might be exploited for mischief, has the effect of hugely expanding the threat model for attacks on the network infrastructure. For example, "denial-of-service" attacks can now be made against a variety of resources, such as CPU cycles, output link bandwidth and storage, since these are exposed either wholly or in part to loaded programs.

Typical reasons for deferring consideration of security, aside from simple difficulty, are the negative consequences making a system more secure has for each of flexibility, usability and performance. Since the programming language based approaches to active networking offer advantages in terms of flexibility and usability, and performance optimizations for these environments are ongoing, providing security to such an environment would offer an attractive design point among the various tradeoffs.

1.2 Paper Overview

In this paper, we begin with a description of the Secure Active Network Environment (SANE). The SANE infrastructure provides security guarantees to the network elements and overlaid services. Additional security services can be built on top of SANE, using the existing primitives. These primitives include secure bootstrapping using the AEGIS architecture [AKFS98]; key exchange; authentication and identification of network entities; packet confidentiality and integrity; resource and access control; and name space protection.

SANE's approach to providing high performance is based on the observation that security-based restrictions on programming environments are enforced by use of checking, *e.g.,* arguments, addresses or operations. Checking can either be done statically, for example to force the system into an acceptable initial state, or dynamically, to ensure that the system remains in an acceptable state. Since dynamic checks are more common, they are good candidates for optimization, *e.g.,* address checks are performed in a processor's virtual address translation hardware. Our approach to providing security does so with good performance because we perform static checks which allow later dynamic checks to be faster, or even eliminated.

Section 2 briefly presents the SwitchWare architecture. Section 3 discusses old and new threats in such an environment. Section 4 describes the SANE architecture, and Section 5 presents the

[*] Scott Alexander, William Arbaugh, and Angelos Keromytis are each working toward a Ph.D. in computer and information science at the University of Pennsylvania.

[†] Jonathan M. Smith is an associate professor at the University of Pennsylvania.

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

current status of the implementation along with a list of experiments and performance results. Section 6 briefly reviews related projects. Finally, Section 7 discusses future extensions and directions.

2 Overview of SwitchWare

While the Secure Active Network Environment (SANE) architecture is portable across many active networking environments, our experimental prototype is constructed in the context of the SwitchWare active network architecture. SwitchWare is based on the approach of using restricted semantics to contain the behavior of potentially mischievous programs. This has the benefit that enforcing restrictions can be performed once at compile or link time, resulting in a lower cost than an OS approach such as memory protection which requires repeated checks at runtime. These semantic restrictions depend on the integrity of other system components such as the operating system, shared libraries, etc. The semantic restrictions are enforced with a strongly-typed language which supports garbage collection and module thinning.

2.1 Why Does the Language Matter?

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

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

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

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

The final property which we require is the ability to dynamically load programs. Clearly, if we intend to run programs that arrive over the net, we must have a way to link those programs into the running system and evaluate them. Dynamic loading gives us this ability.

The Caml programming language [Ler95] provides these features. Caml additionally provides us with a threads interface and static type checking. The former allows a natural programming style and precludes the need to implement a scheduler. (We have, however, discovered that the scheduler imposes an unexpectedly high overhead. See section 5 for details.) The latter pushes many of the costs associated with the type system to compile time. Thus, checks that other systems perform repeatedly at runtime, we perform once at compile time.

2.2 The Loader

The loader forms the basis of the dynamic security for our network infrastructure. Once it has been securely started by the AEGIS bootstrap, the loader provides a minimal set of services necessary to find the Core Switchlet and start it running. It also provides policy and mechanism for making changes to the Core Switchlet, if that is desirable.

The loader is responsible for providing the mechanism by which modules are loaded. Currently, the mechanisms provided are loading from disk or loading from the network. The Core Switchlet governs the policy by which this mechanism may be used and may provide interfaces to the mechanism.

2.3 The Core Switchlet

The Core Switchlet is the privileged portion of the system visible to the user. Through the use of module thinning, it determines which functions and values are visible to which users. The services that it provides are broken into several modules.

The first module is `Safestd`. This module provides the functions that one would expect to find in any programming language including addition and multiplication as well as more complex abstractions like lists, arrays, and queues. Many functions including the I/O functions have been thinned from this module to make it safe.

The next module is `Safeunix`. This module has been very heavily thinned; it gives access to Unix error information, some time related functions, and some types that are needed for the networking interface that we provide. Access to the rest of the Unix functions has been thinned away.

In order to allow the user to supply error or status messages, we have a `Log` module. The user supplies a string which will be saved to a system log. What and where this system log is, is not defined. For convenience while debugging, we currently write the messages to a disk file, but for security purposes, we intend to extend this module to limit the number and frequency of messages produced by any given thread.

Access to the network is provided by the `Unixnet` and the `Safeudp` modules. `Unixnet` provides access to raw Ethernet frames; `Safeudp` provides access to the Linux implementation of UDP [Pos80]. This allows switchlets to access network interfaces for either sending or receiving frames or packets. Currently, only one switchlet is allowed to have access to a given interface or UDP port. In the near future, we intend to modify this module to receive and demultiplex the data. Access to the data will then be available to any switchlet, assuming said switchlet can prove that it has the authority to access the data. For the work described in this paper, we used the `Safeudp` interface.

Thread support in SwitchWare is provided by a set of three modules: `Safethread`, `Mutex`, and `Condition`. These provide a threads package which helps in the structuring of the system. Each switchlet runs in a thread and is capable of creating additional threads. When a switchlet is first started, it is given an identifier inside of an opaque type. (An opaque type is one which has no conversion functions to or from any other type. Thus, the identifier cannot be forged.) In order to use additional resources including creating additional threads, the switchlet must provide its identifier which allows the system to check the resources currently consumed and allow or deny the request for additional usage.

Finally, we have a set of modules to support loading of switchlets. The `Aegis` module allows access to the AEGIS public keys.

The An_marshal module gives interfaces to allow quick transformations between strings and the standard format in which we access our active packets. The Func module allows files or strings to be loaded and executed based on the system access policy. Finally, Route is a very simplistic static routing scheme which allows us to impose an arbitrary active network topology on top of our physical network without the need to crawl under desks to move cables.

2.4 The Library

The library is a set of functions which provide useful routines which do not require privilege to run. The proper set of functions for the library is a continuing area of research. Some of the things that are in the library for the experiments that we have performed include utility functions for sending and receiving active packets.

3 Threats

An active network infrastructure is very different from the current Internet. In the latter, the only resource consumed by a packet at a router is the memory needed to temporarily store it and the CPU cycles necessary to find the correct route. Even if IP [Pos81] option processing is needed, the CPU overhead is still quite small compared to the cost of executing an active packet. In such an environment, strict resource control in the intermediate routers was considered non-critical. Thus, security policies [Atk95c] are enforced end-to-end. While this approach has worked well in the past, there are several problems. First, denial of service attacks are relatively easy to mount, due to this simple resource model. Attacks to the infrastructure itself are possible, and result in major network connectivity loss. Finally, it is very hard to provide enforceable quality of service guarantees. [BZB$^+$97]

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

- Identify the sending network element
- Identify the sending user
- Grant access to appropriate resources based on these identifications
- Allow execution based on the authorizations and security policy

In networking terminology, the first three steps comprise a form of admission control, while the final step is a form of policing. A second view is that of static versus dynamic checking. Security violations occur when a policy is violated, *e.g.,* reading a private packet, or exceeding some specified resource usage.

4 Overview of SANE

The following subsections present the components of SANE and explain how they fit together. Figure 1 shows the various components of SANE and their dependencies. SANE provides security from the moment that power is applied to an active network node. This is done by using a secure bootstrap process that provides integrity guarantees for nodes firmware and operating system components. Once the operating system and active network environment, *e.g.* Caml runtime have been verified, the static integrity guarantees of the system have been assured and we transition to our dynamic integrity mechanisms.

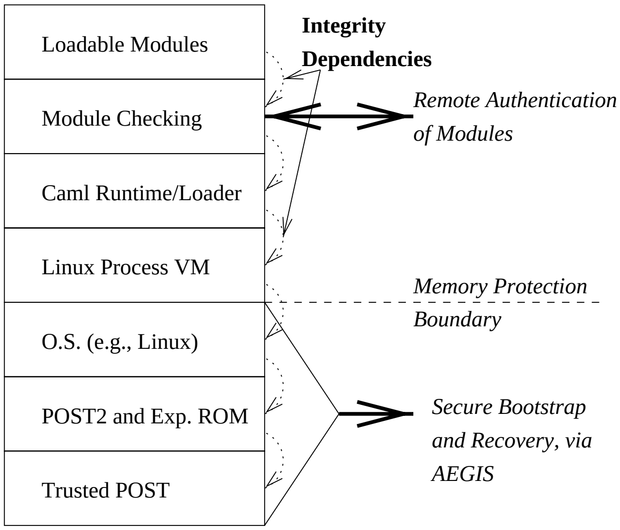

Figure 1: **SANE Architecture**

4.1 AEGIS

All secure systems assume the integrity of the underlying firmware, but typically cannot identify when this assumption becomes invalid. This inability to detect changes in the integrity state of the hardware and firmware results in a significant security problem. The AEGIS Secure Bootstrap architecture reduces the severity of this problem by providing static integrity guarantees of the bootstrap process. We define the static integrity property to mean that an object has not been altered while in storage or transit. We further define dynamic integrity as a property that an object has not been altered while in use. For instance, self-modifying code violates the dynamic integrity property.

AEGIS provides static integrity guarantees by using a combination of two techniques [AFS97] [AKFS98]. The first technique reduces the size of the firmware assumed as having the static integrity property down to the small section that tests the proper operation of memory and the motherboard. The second technique uses induction, digital signatures and modifications to the control transitions from major modules, *e.g.* CALL and JUMP instructions, to ensure the static integrity of the next module. We call the combination of these techniques Chaining Layered Integrity Checks (CLIC).

4.1.1 Chaining Layered Integrity Checks

Complex systems typically use structured decomposition to provide different levels of abstraction with increased functionality at succeeding levels. These decompositions induce a **depends upon for correctness** dependency relation between the levels [Par74]. In secure systems, this form of decomposition serves to reduce the size and number of the objects that require verification, *e.g.* assurance of the object's proper operation. This is important because the verification process is typically difficult and expensive. Once a system of this type has been decomposed and deemed trustworthy by a verification process, ensuring the system remains trustworthy, *e.g.* the system's static integrity remains unchanged, is a problem since re-verification is also costly. CLIC solves this problem by using a form of induction over the dependency relations, *e.g.* levels, created by the decomposition of the system. The base case of the induction argument is the root of a derivation tree where the integrity of this level, L_i, is assumed as correct. The induction step is the verification, V, of a digital signature affixed to the next level, L_{i+1}, in the derivation tree. If the integrity of level L_{i+1} is verified, then the induction process is continued until the L_{n-1} level at which point the integrity of the system is proven to be the same as when the digital signatures were affixed. This results in the simple recurrence relation for CLIC shown in Equation 1.

$$\begin{aligned} I_0 &= True, \\ I_{i+1} &= \Big\{ I_i \bigwedge V_i(L_{i+1}) \quad \text{for } 0 < i < n. \end{aligned} \tag{1}$$

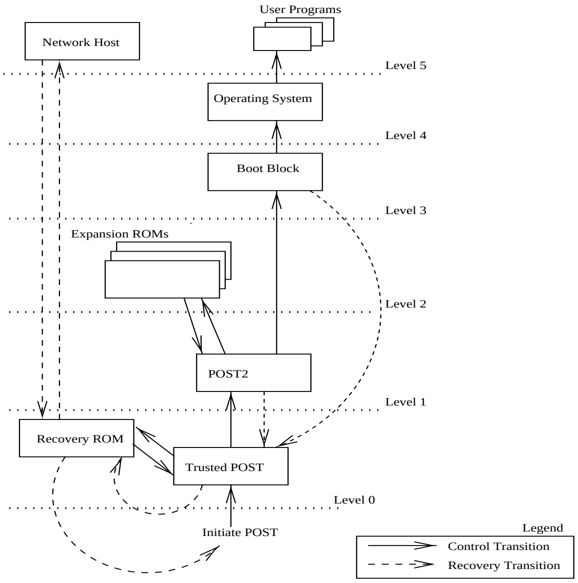

Figure 2: AEGIS boot control flow

With the proper application of CLIC as in AEGIS, the static integrity of the bootstrap process is now guaranteed thereby increasing the level of trustworthiness of the initialization process. Figure 2 depicts how AEGIS divides the bootstrap process of the IBM Personal Computer architecture into modules or levels. Level 0 is the base case and the only firmware component whose integrity is assumed valid. Level 0, or Trusted Power on Self-Test (POST), ensures that the hardware and memory located on the motherboard are operating properly. Level 1, or POST2, begins the initialization of the computer system for use by scanning for expansion ROMs located on add-in cards located on the ISA and PCI buses of the motherboard. Level 3, or Boot sector, begins the initialization of the operating system by copying the OS kernel (or secondary boot sector) into memory and passing control to the loaded image. Finally, Level 5 represents the system and user level applications that run under operating system control. Each transition to a higher level represents a control flow change in the bootstrap process. The action taken at each Level transition is shown in the pseudo code below:

```
void LevelTransition (Level NextLevel)
{
 Hash h = ComputeHash(NextLevel);
 Signature s = GetSig(NextLevel);
 PublicKey p = GetPubKey(s);

 If Verify(s, h, p)
 JUMP(NextLevel);
 Else
 Recover(NextLevel);
}.
```

When the integrity of a module representing a Level is invalid, a recovery process is called via the Recover function call. The recovery process contacts a trusted repository via a secure protocol [AKFS98] and repairs the modules stored on writable storage such as the hard disk or flash memory, or uses the memory controller to shadow modules stored on ROM to provide a temporary repair. This permits the system to boot when integrity errors occur and prevents some denial of service attacks.

4.1.2 AEGIS Services After Bootstrap

After the Active Network Environment becomes operational, the only service AEGIS provides is read access to sensitive cryptographic information such as keys and certificates. There are several reasons for limiting the services that AEGIS provides to the active environment. The first is that since the AEGIS software is embedded, it is written mostly in assembly language and C in order to optimize its space rather than its performance. Additionally because of issues involving memory management, the AEGIS routines would have to be reloaded from ROM with each use impacting overall system performance. Finally, the core of the security routines for SANE should be in Caml in order to provide the security benefits described in Section 2.1.

Providing read only access to the sensitive cryptographic information, however, is a reasonable service for AEGIS to provide since this permits secure storage of the information on the boot block of the flash ROM, which provides additional hardware protection against reprogramming [Haz95]. Storing the information in this manner provides a significant increase in security over storage of the information in the file system of the operating system or active network environment.

4.2 Cryptographic Primitives

SANE provides access to various cryptographic primitives. These can be used by other applications as-is or as building blocks for more complex protocols. The services initially provided are:

- public key signatures (DSA [NIS94])
- symmetric key encryption (DES [NBS77])
- (keyed) hashes (SHA1 [NIS95])

This set of primitives may be enriched in the future. All the algorithms have been implemented in Caml but due to performance degradation, we use a C version of SHA1. Access to this implementation of SHA1 occurs through a Caml interface, taking care to avoid potential bypassing of the type system. Hardware cryptographic support is being considered.

4.3 Public Key Infrastructure

In our architecture, every network entity (active switch or user) owns at least one private / public key pair. These keys (and the corresponding certificates) are used to authenticate these entities and authorize their actions. Although SANE depends on a public key infrastructure, it is not tied to a particular one. Certain features, such as selective authorization delegation, user defined authorizations and certificate revocation through expiration are desirable, but they can be simulated in any of they proposed public key infrastructures. In our environment, we intend to use a combination of SPKI [EFRT97] and PolicyMaker [BFL96]. For more details on the certificate format, see Section 5.

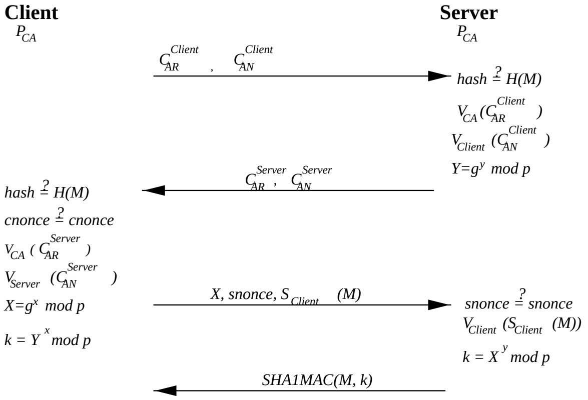
Client
P_{CA}
Server
P_{CA}
C_{AR}^{Client} , C_{AN}^{Client}
$hash \overset{?}{=} H(M)$
$V_{CA}(C_{AR}^{Client})$
$V_{Client}(C_{AN}^{Client})$
$Y=g^y \bmod p$
C_{AR}^{Server} , C_{AN}^{Server}
$hash \overset{?}{=} H(M)$
$cnonce \overset{?}{=} cnonce$
$V_{CA}(C_{AR}^{Server})$
$V_{Server}(C_{AN}^{Server})$
$X=g^x \bmod p$
$k = Y^x \bmod p$
$X, snonce, S_{Client}(M)$
$snonce \overset{?}{=} snonce$
$V_{Client}(S_{Client}(M))$
$k = X^y \bmod p$
SHA1MAC(M, k)

Figure 3: **Authentication Message Exchange**

4.4 Key Establishment Protocol (KEP)

A key element of SANE is the key establishment protocol. The protocol itself is shown in Figure 3[1], and is a strengthened variation of the Station-to-Station [DvOW92] protocol, which uses Diffie-Hellman [DH76] key exchange and public key signature authentication. The goal of the exchange is to establish a shared secret and authenticate the two protocol participants (node / node or user / node). Once the key is established, the authorizations of each party are determined, through the exchange of the appropriate certificates. An example of such authorization is the amount of memory a user is authorized to use on the active switch. The derived shared key is used to authenticate and / or encrypt further communications between the two parties.

4.5 Packet Authentication

Once a key has been established between two nodes, they can commence exchanging authenticated and / or encrypted packets. In SANE, we use the ANEP [ABG$^+$97] packet format over UDP, although in an homogenous active network a packet format would be unnecessary. We've added an authentication header, as shown in Figure 4, similar to the one used in the IPsec Authentication Header protocol [Atk95a]. The SPI is negotiated during the key establishment protocol exchange, and is used to identify the security association and corresponding cryptographic material used. The $Replay\ Counter$ is a monotonically increasing value, used to prevent packet replay attacks. The $authenticator$ is the keyed hash (HMAC [KBC97]) computed over the SPI, $replaycounter$ and packet payload. We can similarly define an encryption header similar to the IPsec ESP [Atk95b] protocol.

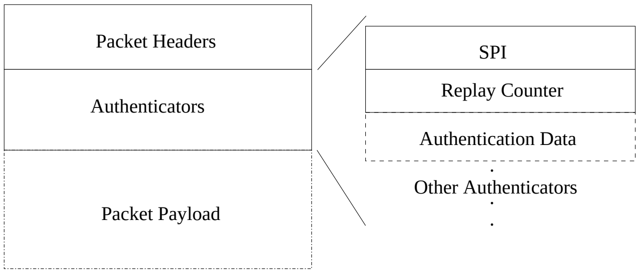

Figure 4: **Authenticator Header**

4.6 Link Keys

When a SANE node boots, it attempts to establish shared keys with each of its neighbors. It does this by running the key establishment protocol already described. In the process, the identity of the neighbors is also verified. The administrator of an active network can essentially "freeze" the network topology by specifying which nodes can be neighbors. There are certain benefits in doing this:

- Certain distributed types of protocols (such as routing) can be secured against outside attacks
- The switch offers secure forwarding services to any active packet that requests them. This is important for mobile agent types of applications that cannot depend on end to end security, but require some security guarantees on a hop-by-hop basis.
- Administrative domains and their boundaries can be established through this process. We define an administrative domain as the set of active nodes that are managed by the same entity, have a common set of access and resource management policies and, after the KEP is run, trust each other to make trust decisions on their behalf.

4.7 Administrative Domains

A user who needs to load a number of modules on a set of active nodes would typically have to contact each node individually and establish security associations (SAs) with each one. This establishment could happen in either a telescopic manner (where the user "explores" the network) or a parallel manner (if the user knows the identities of all the switches in advance). This can prove expensive both computationally (because of the public key operations) and in packet size (since there must be a separate authentication payload for each node that a packet may visit).

By taking advantage of the existence of administrative domains, we could make some optimizations:

- Once the user has established an SA with some active node in another administrative domain, that node can act as a key distribution server (KDC) similar to Kerberos [MNSS87].
- Only nodes at the perimeter of an administrative cloud need verify the cryptographic integrity of packets. They can then specify what the active packet can do in the interior of the domain. In that respect, any machine at the edge of the domain can act as a firewall. In contrast to the Internet firewalls however, policy can be specified but not enforced at the edges; enforcement of access and resource management policies has to take place in the interior [BKS98].

4.8 Resource Control

Resource control on the active switch is imposed by the runtime system, as specified by the certificates exchanged during key establishment. The protected resources include access to standard and loaded modules, CPU cycles, memory allocated, number of packets, latency and bandwidth requirements, and others. It is a subject of further research exactly what the right resources are and how to resolve conflicting resource requests.

In any case, since a tenet of our approach is controlled loading of modules, SANE must manage loading modules in a secure fashion if it is to be useful in an active network. That is, it must control which modules are loaded, and by whom. SANE associates cryptographic certificates with modules. SANE can either require

[1] Some details were left out. For more details on the protocol, see [AKS98].

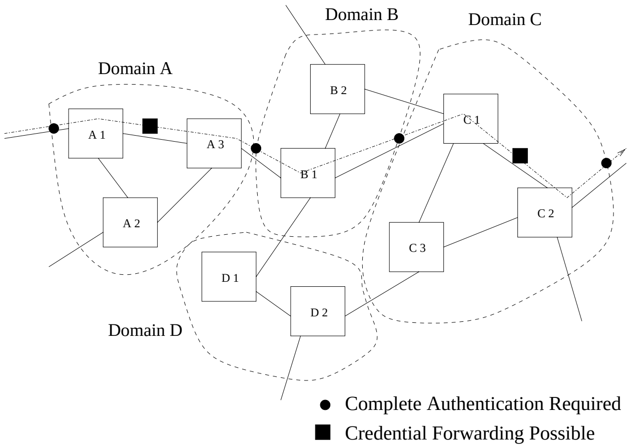
Domain A
Domain B
Domain C
Domain D
A 1
A 2
A 3
B 1
B 2
C 1
C 2
C 3
D 1
D 2
● Complete Authentication Required
■ Credential Forwarding Possible

Figure 5: **Administrative Clouds and Path Setup**

a certificate for loading a particular module, or may allow universal loading of the module. Examples where such universal loading may be useful include low-cost operations like `ping`, as well as the security operations used for bootstrapping the security relationship with remote switches. There are two classes of certificate which can be presented by a user packet requesting access to a resource via a module. An *administrative* certificate allows loading of any or all modules into the system; it is intended for management and emergencies as might arise, and can be thought of as analogous to a ”master key” granted by the switch administrator. More commonly, certificates are used to permit loading of selected modules. As a result, this scheme allows fine-grained control of switch resources.

4.9 Naming

Conceptually, loaded modules can be considered as the interfaces to user defined resources. Such resources will generally be shared between different sessions of the same principal, or even between different principals. These principals will need to identify (name) the particular resource they want to use.

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

- The name could be the one-way hash of the module code. If we assume certain properties of the hash function, this uniquely identifies the module. The two potential drawbacks to this approach are that different versions of related services have unrelated names and that users have to discover the hash value (either through access to the code or by finding a trusted source that will give the user the hash value). To use the module represented by this name, a switchlet would have to trust only the module itself.

- The name could be the module programmer's public key (or its one-way hash), along with some other identifier assigned by the programmer (such as an ASCII string). The assumption here is that the code may be signed by the programmer (who may be different from the principal who loaded it on the active element). Version control is possible (subject to the structure of the programmer-assigned identifier). The signature would have to be verified by the active node before this name becomes available. In this case, a switchlet would need to trust the programmer before using the module represented by this name.

- The name could be the public key of the principal who loaded the code onto the active element (or the one-way hash of that key), along with some other identifier assigned by the principal. Since the principal must pass an authentication / authorization check before allowed to load the code, there is no additional overhead imposed by this naming scheme. In this case, the switchlet must trust the installer before using the module so represented.

In fact, it is possible to combine these naming schemes, as they are not mutually exclusive. Different programs may access the same resource through different names, depending on the trust policies of their respective owners. Actually accessing these services depends on the node architecture and implementation; we plan to use a portmapper-like approach, but other approaches (*e.g.,* language constructs) are possible.

As an example, imagine a principal X with a public key P who loads a new service that implements IP packet forwarding on an active node. The service was written by a programmer R who signed it with his key Q. The hash of the code is also known to be H. Any user can then access this service as:

1. $\{P$, “IPv4/version1”$\}$ — the IPv4 module (version 1) loaded by X,

2. $\{Q$, “IPv4/version1”$\}$ — the IPv4 module (version 1) written by R,

3. $\{H\}$ — the IPv4 module known to the user, or

4. $\{Q$, “IPv4/version2”|“IPv4/version1”$\}$ — the IPv4 module (version 2) if available, otherwise the previous version of the same module.

5 Implementation and Performance of SANE

We have implemented SANE in the SwitchWare environment. For our experimental network we used a cluster of DEC Alpha PC 164SX machines, with 533MHz processors and 64MB memory each, connected via 100Mbit switched Ethernet. All the test machines were running RedHat Linux, kernel version 2.0.33, and a modified Caml 1.0.7 runtime system. For some of our throughput tests, we modified the Linux kernel to allow allocation of a buffer larger than 64KB per socket.

5.1 Cryptographic Primitives

Tables 1, 2, and 3 show the costs of the three cryptographic primitives provided by SANE. Each was implemented twice based on two different sets of integer primitives. This is because the garbage collector requires a bit from each integer to use as a tag bit. Thus, we have made use of a package called Int32 which supplies full 32 bit integers on both Pentium and Alpha platforms (with additional space overhead); using this package allows a single implementation of our cryptographic routines which will run on either platform. (As the tables show, this portability can come at a substantial cost in performance.) Finally, in addition to the bytecode interpreter which we use, the Caml distribution also provides a native code compiler which produces Alpha executables. Table 1 gives the average time in seconds to hash a 4MB string using either the Int32 package and using the 63 bit integers provided by Caml on the Alpha. Additionally, it shows the difference in cost between compiled and interpreted code. Table 2 shows the cost to encrypt a 4MB message using 63 bit integers with either the bytecode or native Alpha code. Finally, table 3 shows the cost in milliseconds of signing and of verifying the message “abc,” using DSA. Since a DSA signature consists of computing a SHA-1 digest followed by the signature process itself, for a longer message, one should add the cost of performing the hash.

Caml	Int32	bytecode	86.446289 s
		native	61.991894 s
	Alpha ints	bytecode	36.027246 s
		native	2.477051 s
C			0.333212 s

Table 1: **Time to SHA-1 hash 4MB of data**

Caml	Alpha ints	bytecode	99.331543 s
		native	16.723242 s
C			1.0785348 s

Table 2: **Time to DES encrypt 4MB of data**

In practice, to use the dynamic loader in Caml, we must use the bytecode interpreter. This imposes a very high overhead on authenticating packets, an operation which relies on the SHA-1 hash function, so we have resorted to a C implementation. While this greatly speeds the HMAC generation and verification operations, it may interfere with the Caml runtime thread scheduler. Specifically, when the end of a quantum occurs, if the current thread is executing C code, no call to the scheduler occurs and the thread will get an extra quantum. Furthermore, when using a C code implementation, we cannot catch type-system errors internal to that code, nor take advantage of the garbage collection mechanism available in the runtime. For these reasons, we tried to limit the amount of non-Caml code in our system. Thus we opted to keep the Caml DSA and DES implementations. In the future, we intend to investigate the feasibility of statically integrating Caml native code into the bytecode interpreter in the same way that we currently are able to integrate C code. This would allow us to regain the advantages of strong types and garbage collection with a more acceptable overhead. We also believe that in the future, “Just In Time” compilation techniques can narrow this gap in performance.

The key exchange protocol was also implemented in Caml as a three step protocol. In the first two messages, a list of SPKI-like certificates encoded as a string is exchanged. The third message contains a single certificate. Since the SPKI format has not been fully specified, we designed our own certificate format in the same spirit. The protocol was designed to be fail safe [GS95] under all circumstances. In the presence of loosely synchronized clocks, it becomes fail stop (meaning that active attacks, including replays, on the protocol, are always detected). We encode all fields in the certificates as strings before transmission, and for signing and verification purposes. This allows us to avoid complicated marshalling issues. The average execution time of KEP with a 256 bit Diffie-Hellman exponent is 2.4 seconds, and with a 1024 bit exponent, 4.8 seconds. In both cases we used a 1024 bit modulus. This time is comparable to that of the IPsec key management protocols, Photuris [KS] and ISAKMP/Oakley [MSST96].

The certificate infrastructure we used in our setup is a shallow hierarchy. A small number of keys are considered as trusted to make statements about nodes or, more specifically, what the network topology is. These same keys are also used to certify users and specify their access rights on the active nodes. It is only a matter of policy however what sort of certificate method is followed. A cyclic graph-type (such as in PGP) or a hierarchical approach (such as in X.509 [Com89]) or any other method can be used. Furthermore, there is no need for an organization’s internal certification policies to be the same as the interdomain and interorganizational policies.

sign	Caml	Int32	bytecode	27.089 ms
			native	12.954 ms
		Alpha ints	bytecode	20.907 ms
			native	11.855 ms
	C			2.800 ms
verify	Caml	Int32	bytecode	41.452 ms
			native	22.121 ms
		Alpha ints	bytecode	35.198 ms
			native	20.664 ms
	C			5.000 ms

Table 3: **Digital Signature Timings**

5.2 Cost of Active Ping and Active Data Movement

To understand the cost imposed by authentication, we measured the cost of sending an active ping (provided as Appendix A for illustration) both with and without authentication. This ping was generated at a source machine, transmitted over a crossover cable via 100 Mbps Ethernet to the target machine, loaded and evaluated, then sent back to the source machine, where it was again loaded and evaluated. An unauthenticated ping took an average of 5.084 ms versus 8.052 ms for the authenticated ping.

Finally, we measured the throughput of authenticated and unauthenticated data transfer. We used two versions of active packets: one that was dynamically linked and one that was statically linked at startup time. The reasoning for this is that since simple data transfer between active applications is a common operation, it is conceivable that a simple data delivery service, with semantics similar to those of UDP, would be standardized. Applications can still use their own data delivery packets, with a hit in performance, as shown in Figure 6. Additionally, this shows the benefit that a switchlet caching scheme could provide.

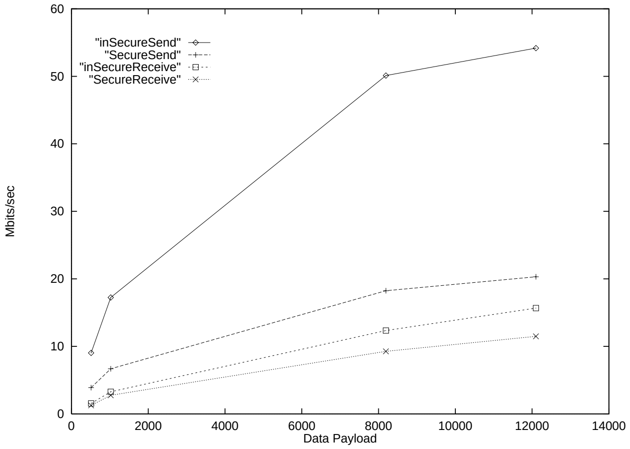

Figure 6: **Data Transfer Throughput**

5.3 Performance of the Caml Runtime

To improve performance, we added a new interface to the dynamic linking support in Caml. As distributed, one can only link files from the disk. Reading saneping.cmo, the source for the authenticated active ping, costs at least 3ms even if the file is in the buffer cache. Thus, we added an interface that links a bytecode “file” directly from memory; when a switchlet arrives over the network, we are able to link it in without having to resort to the filesystem.

Two other areas of performance problems are the thread scheduler and the symbol table support within the dynamic linker. Each time the scheduler runs, it checks the status of every thread in the system. Thus, any extra threads (*e.g.,* one waiting for the completion of another thread or one which is delaying for a fixed interval) causes measurable additional overhead. Moreover, if there is any thread which is waiting on an I/O operation, the scheduler does a select() to see if that operation can now complete. We measure a typical cost of 100 to 250 μs each time the scheduler is called. This leads to a tension between the desire to avoid this overhead with the desire to use a scheduler to prevent any thread from using more than its share of the processor. Adopting scheduling algorithms from operating systems could be of benefit here.

Updating the symbol table is currently the largest fraction of the cost in dynamically linking a new module. So, for example, to link the active ping, the total cost (including evaluating the top level forms) averages 1470 μs. Of this, symbol table updates average 891 μs. Of the symbol table update time, performing relocation operations and adding newly introduced symbols to the symbol table averages a cost of 794 μs. Since this cost is included in the ping cost for every node visited except for the initial node, in our simple, two node experiment, this overhead is incurred twice. Thus, out of our average 5084 μs for an active ping, more than half of the time is spent updating the symbol table generally and more than 30% is spent performing relocation and symbol table insertions. Worse yet, if we deduct the 990 μs generally spent in the kernel and in transmission, we find that we are spending over 70% of the time that we can control in symbol table updates and nearly 40% in relocation operations. Finally, after about 100 pings, the time starts to grow, presumably because of the growth of the symbol table. After 300 pings, the time to perform the symbol table updates has grown to an average of 1011 μs.

We believe that the symbol table costs can be addressed in several ways. First, for a switchlet doing more substantial work than ping, the overhead will be less significant. For example, this infrastructure is being used to support work in which the queueing strategy of a switch is altered dynamically. Since a queueing strategy can be expected to run for minutes at the very least, an overhead of several milliseconds can be reasonably amortized. It may be possible to decrease the cost of performing insertions and finds in the symbol table through a space-time trade-off. For example, the current balanced tree scheme could be replaced with a hash table scheme. Finally, we suspect that the increasing cost is due to the increasing size of the symbol table. If we were to distinguish between switchlets which provide services to later switchlets (and thus need to be in the global symbol table) and those which are intended to be ephemeral, Caml would allow us to update the symbol table only with long-lived switchlets. A more general approach to combating both the increase in size of the symbol table and the cost of loading switchlets is to cache frequently run switchlets [WGT98].

6 Related Work

The Secure Active Network Environment has no direct analogues in ongoing work on active networks [TSS$^+$97]. While ANTS uses MD5 hashes (“fingerprints”) to name on-demand loaded modules, the hashes provide unique names rather than security. The ANTS execution environment depends on the Java programming language for protection, a dependency shared by many active network prototypes. Unfortunately, as Wallach, et al., [WBDF97] note, Java’s security is suspect. The remote authentication and namespace security of SANE address issues ignored in these systems, and could be applied even in cases where Java is used, *e.g.,* to provide integrity checking of the JVM or layers beneath it, as well as on-demand loaded modules.

Another quite different approach to providing secure active networking is that used by the Programming Language for Active Nets (PLAN). PLAN is a special-purpose programming language appropriate for per-packet programs. PLAN’s semantics are purposely restricted to operations which are safe and bounded in resource usage, with the intention of being so lightweight that any node would be willing to run PLAN packets, including those from remote nodes, and thus would not require the security of SANE. However, as any enhanced services are added to the node as PLAN extensions, such extensions would require a SANE-like approach for security.

An architecture which extended a protection model from the local domain to a distributed environment was provided by Sansom, et al. [SJR86], who enforced protection locally with memory-protection enforced capabilities. (It is notable that capabilities can be viewed as a namespace-based protection mechanism). The capabilities were extended to remote nodes via cryptographic means. SANE provides more general mechanisms and could thus be specialized to such an application (moving memory-protected objects about the network) but more importantly guarantees local integrity before extending itself into the network.

7 Future Work

There is still work to be done to produce a mature version of SANE. The two immediate goals are to fully integrate AEGIS with the current implementation, and add the necessary hooks in the runtime system to do resource management. For the latter, we will have to identify the important resources and integrate PolicyMaker in the runtime system.

We are also looking into moving SANE from a general operating system (Linux) to a more dedicated environment. Work is being done in porting the Caml runtime to the Flux OS Kit [FBB$^+$97] environment. We believe that this will improve the performance of the system, since we will avoid boundary crossings (everything is in the same address space). Elimination of the unnecessary services and programs available in the Linux environment will also improve the security of the active nodes. Moreover, we intend to use ANEP directly over ethernet and avoid using UDP as the underlying transport mechanism. For this, we need a packet fragmentation and reassembly mechanism. We are examining the mechanism available in the PLAN environment for possible use.

Over the longer term, we intend to implement the KDC and “active firewall” functions of SANE. As mentioned before, hardware cryptographic support is also being considered. We believe that a great performance improvement could be had by adding Just In Time compilation capabilities in the Caml runtime system.

Our experiments also indicated that dynamic code generation can be of great use in minimizing the size of the active programs, especially in the case of simple data-transport packets. There is also work done in compressing the active code overhead, such as the PLAN architecture, which can prove useful in SwitchWare.

Finally, we intend to examine how the SANE architecture can be used in other active network environments. In particular, we believe that SwitchWare can benefit from the code distribution mechanism employed by the ANTS [WGT98] system and, conversely, ANTS can make use of the security services offered by SANE.

8 Acknowledgements

We’d like to thank Bill Marcus for his help in writing some of the original ANEP code, and Mike Hicks for the discussions and tools he provided us for performance analysis.

References

[ABG$^+$97] D. Scott Alexander, Bob Braden, Carl A. Gunter, Alden W. Jackson, Angelos D. Keromytis, Gary J. Minden, and David Wetherall. Active network encapsulation protocol (anep). http://www.cis.upenn.edu/~angelos/ANEP.txt.gz, August 1997.

[AFS97] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[AKFS98] William A. Arbaugh, Angelos D. Keromytis, David J. Farber, and Jonathan M. Smith. Automated Recovery in a Secure Bootstrap Process. In *To appear in Network and Distributed System Security Symposium*. Internet Society, March 1998.

[AKS98] William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. Dhcp++: Applying an efficient implementation method for fail-stop cryptographic protocols. Technical report, Department of Computer Science, University of Pennsylvania, January 1998.

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

[Atk95a] R. Atkinson. IP authentication header. RFC 1826, August 1995.

[Atk95b] R. Atkinson. IP encapsulating security payload. RFC 1827, August 1995.

[Atk95c] R. Atkinson. Security architecture for the internet protocol. RFC 1825, August 1995.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[BKS98] Matt Blaze, Angelos D. Keromytis, and Jonathan M. Smith. Firewalls in active networks. Technical report, University of Pennsylvania, February 1998.

[BZB$^+$97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation protocol (RSVP) – version 1 functional sepcification. Internet RFC 2208, 1997.

[Com89] Consultation Committee. *X.509: The Directory Authentication Framework*. International Telephone and Telegraph, International Telecommunications Union, Geneva, 1989.

[DH76] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, Nov 1976.

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

[EFRT97] Carl M. Ellison, Bill Frantz, Ron Rivest, and Brian M. Thomas. Simple Public Key Certificate. Work in Progress, April 1997.

[FBB$^+$97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: A substrate for os and language research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[GS95] Li Gong and Paul Syverson. Fail-Stop Protocols: An Approach to Designing Secure Protocols. In *Proceedings of IFIP DCCA-5*, September 1995.

[Haz95] Peter Hazen. Intel's Flash Memory Boot Block Architecture for Safe Firmware Updates. Application Brief AB-57, Intel, December 1995.

[HKM$^+$98] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Plan: A programming language for active networks. Technical report, Department of Computer and Information Science, University of Pennsylvania, February 1998.

[HPB$^+$97] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck. Joust: A platform for communications-oriented liquid software. Technical report, Department of Computer Science, University of Arizona, November 1997.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC:Keyed–Hashing for Message Authentication. Internet RFC 2104, February 1997.

[KS] P. Karn and W. A. Simpson. The Photuris Session Key Management Protocol. Work in Progress.

[Ler95] Xavier Leroy. *The Caml Special Light System (Release 1.10)*. INRIA, France, November 1995.

[MNSS87] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Technical report, MIT, December 1987.

[MSST96] Douglas Maughan, Mark Schertler, Mark Schneider, and Jeff Turner. Internet Security Association and Key Management Protocol (ISAKMP). Internet–draft, IPSEC Working Group, June 1996.

[NBS77] Data Encryption Standard. Technical Report FIPS-46, U.S. Department of Commerce, January 1977.

[NIS94] Digital Signature Standard. Technical Report FIPS-186, U.S. Department of Commerce, May 1994.

[NIS95] Secure Hash Standard. Technical Report FIPS-180-1, U.S. Department of Commerce, April 1995. Also known as: 59 Fed Reg 35317 (1994).

[Par74] D. L. Parnas. On a 'buzzword': Hierarchical structure. In *Proc. of the IFIP Congress*, pages 336–339. North-Holland, 1974.

[Pos80] Jon Postel. User datagram protocol. Internet RFC 768, 1980.

[Pos81] Jon Postel. INTERNET protocol. Internet RFC 791, 1981.

[SJR86] R. D. Sansom, D. P. Julin, and R. F. Rashid. Extending a capability based system into a network environment. In *Proceedings of the 1986 ACM SIGCOMM Conference*, August 1986.

[TSS$^+$97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, pages 80–86, January 1997.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Flexible security architecture for java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[WGT98] David J. Wetherall, John Guttag, and David L. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols. In *To appear in IEEE OpenArch*. IEEE Computer Society Press, April 1998.

A Code for the Active Ping

```
open Safeloader
open Printf
open Wf
open Safeunix
open Log
open An_marshal

type ping_packet = { start : string ;
     finish : string;
     timestamp : float}

let ping_encode pkt = "start = " ^ pkt.start
  ^ "; finish = " ^ pkt.finish
  ^ "; timestamp = "
  ^ (string_of_float pkt.timestamp)

let decode_regexp = Str.regexp
 "^start = \(.*\); finish = \(.*\)"
 ^ "; timestamp = \(.*\)$"
let ping_decode str ofs =
  if not (Str.string_match decode_regexp
  str ofs)
  then failwith "bad packet"
  else
    { start = Str.matched_group 1 str;
      finish = Str.matched_group 2 str;
      timestamp = float_of_string
                (Str.matched_group 3 str)
    }

(*
 * This is routine that starts things off by
 * handing the components of an ANEP header
 * to send_wf.
 *)
let send_ping dest name =
  let code =
     Get_bytecode.get_bytecode name in
  let next_hop = Route.get_route dest in
  let hdr =
       {do_forward = true; type_id = 20} in
  let payload = ping_encode
 { start = An.getAddress ();
      finish = dest;
      timestamp = Time.get_time()
  } in
  send_wf next_hop hdr  [] "ping_out"
code payload

let ping_out arg_string =
  let {code=code; data=datastr; func=func} =
decode arg_string 0 in
  let ping_packet = ping_decode datastr 0 in
  if  (ping_packet.finish = An.getAddress())
  then begin
    send_wf
      (Route.get_route ping_packet.start)
      {do_forward=true; type_id=20} []
      "ping_in" code datastr;
    "at the remote machine"
  end else begin
    send_wf
      (Route.get_route ping_packet.finish)
      {do_forward=true; type_id=20} []
      "ping_out" code datastr;
    "not there yet; forward packet"
  end

let ping_in arg_string =
  let {code=code; data=datastr; func=func} =
      decode arg_string 0 in
  let ping_packet = ping_decode datastr 0 in
    if (ping_packet.start = An.getAddress())
    then begin
      log_msg
       (Printf.sprintf "Success (%f sec)\n"
  ((Time.get_time()) -.
         ping_packet.timestamp));
      "back at the sender"
    end else begin
      send_wf
        (Route.get_route ping_packet.start)
{ do_forward=true; type_id=20 } []
 "ping_in" code datastr;
      "not back yet; forward packet"
    end

let _ = Func.register "ping_out" ping_out
let _ = Func.register "ping_in" ping_in
```