T2 CAN: A Secure Active Network Prototype

Lidong Zhou ldzhou@cs.cornell.edu

Robbert van Renesse rvr@cs.cornell.edu

Fred B Schneider fbs@cs.cornell.edu

Computer Science Department
Cornell University
Ithaca, New York 14853-7501, USA

DRAFT of November 2, 1997

1 Introduction

A network consists of a set of switches and communication links among these switches, along with network protocols for interoperability. In our view, the switches are interpreters: when messages flow between switches, the switches interpret the messages and process the messages according to network protocols.

For traditional networks, interpreters simply analyze message headers and perform standard operations to transfer messages. Since message headers are all that the interpreters can understand, users have very limited control over the functionalities of traditional networks. In other words, traditional networks provide a fixed and inflexible interface to network users.

As new network applications emerge, there is always a need to update the network protocols, or deploy new protocols to support new applications. For example, new protocols to support mobility and multicast are being proposed. Unfortunately, because of interoperability and backward compatibility requirements, traditional networks suffer from slow deployments of new network protocols.

To expedite the evolution of networks, researchers are proposing *active networks* [12]. Active networks have the same structure as that of traditional networks: a set of interpreters connected with communication links. The only thing that is changed is the expressiveness of the language that interpreters can understand. A message on an active network, which we call an *agent*, carries code with it; an interpreter will activate the code when receiving the message. In this way, applications may instruct interpreters to provide customized network services. In other words, active networks provide a programmable network interface. On an active network, an application can deploy new networks dynamically and select appropriate protocols to achieve the best possible performance.

However, active networks do introduce new technical challenges, and security is a major obstacle to the wide acceptance of active networks. We classify the security problems in active networks into the following three categories: host security, agent security and inter-agent security.

When we allow untrusted foreign agents to run on network hosts, we introduce a severe security risk to hosts. *Host security* aims to preserve the integrity, privacy and availability of hosts. More specifically, we have to prohibit unauthorized agents from tampering with data on hosts; we have to monitor the information flow and make sure that no agent can exploit either standard channels or covert channels to steal secret information; we also have to guard against denial of service attacks — agents might simply consume too many resources and take hosts out of service.

Hosts are to be protected, so are the agents. When agents execute on untrusted hosts, they have to be protected against any attacks that would compromise the integrity and privacy by hosts. Users who send out agents should at least have some degree of confidence that the agents will arrive on the destinations with the desired data. We believe that agent integrity problem is equivalent to that of building fault tolerant agents, while agent secrecy needs a cryptographical solution. In this paper, we are not going to discuss solutions to the *agent security* problem. We refer to [7, 10] for a detailed treatment of the problem.

An agent should not only be protected against the hosts, but also be guarded against other agents. *Inter-agent security* makes sure that agents cannot interfere with each other, even when executing on the same host.

In this paper, we present a secure active network prototype, T2 CAN. T2 (**T**ocoma **T**oo) is the language that the active network interpreters understand, and CAN (**C**ornell **A**ctive **N**etwork) is an implementation of active network interpreters. In T2 CAN, *agents* are the "active" messages that roam on networks. Agents are active because they can execute their code when arriving on a host.

Instead of focusing on what T2 CAN can do, we will mainly describe how T2 CAN addresses host security and inter-agent security problems. We propose a general security model based on *security domains.* A security domain specifies a set of operations. Agents execute with respect to security domains, and can only carry out operations in the domains.

A security framework based on security domain is a general model to enforce security policies. A security policy specifies what sequences of operations a principal (in our case, an agent) is allowed to execute. If, at any time during the execution of an agent, the set of allowed operations is captured in the associated security domain, a runtime enforcement mechanism that implements security domains may monitor the execution of an agent and prevent it from violating the security policies.

T2 CAN implements a domain-based security framework. We will describe two main issues: domain change, i.e., how domains may change when agents execute; and domain enforcement, i.e., how to prevent agents from performing any operations that are not in the current domain.

The paper is organized as follows: in Section 2 , we present three applications for active networks: image retrieval, data fusion and TCP over active IP. We introduce the basic system model of T2 and CAN in Section 3. Section 4 shows how the applications described in Section 2 are implemented on T2 CAN. We describe in detail the T2 CAN security framework based on security domains and the built-in domain change mechanisms. 5. Section 6 presents how domains are enforced in T2 CAN. In Section 7, we revisit the examples of Section 4 and show how the security problems are solved. Finally, the future work and related work are presented in Section 8 and Section 9.

2 Applications on an Active Networks

In this section, we present three applications attempted for active networks: image retrieval, data fusion and TCP over active IP. These applications not only show the advantages of active networks over traditional networks, but also serve to illustrate security problems and solutions in the rest of the paper.

2.1 Image Retrieval

There are more and more resources on computer networks. We have many huge databases on networks, and users can fetch data from a remote server over networks. Consider a database with satellite images, a client may design a customized algorithm to retrieve some useful information from images in the database.

On a traditional network, the client has to request that all the related images be transferred from the database, and run the algorithm locally. This solution may not be feasible for both performance and copyright restrictions.

However, on an active network, instead of moving data to programs, we can move programs to data. We can ship an agent with the image retrieval algorithm to the database server, and the server will invoke the algorithm when receiving the agent. After getting the result of the algorithm, the agent can go back home with the result. Essentially, active networks support mobility of computation, and thus, provide a better solution to problems dealing with large trunks of remote data.

2.2 Data Fusion

Data fusion has been a hallmark example for active networks: in some applications, such as simulations and remote manipulations[12], a user may retrieve and integrate data from multiple sources. On a traditional network, the client can only fetch the data from each source and then fuse the data locally.

While on an active network, we have the option of fusing data on the intermediate hosts. In many cases, fusing data in the network may improve performance, or balance load over the network.

For example, to search for all the books on a certain subject, we may send search requests to multiple databases. It is likely that the results from these databases contain a lot of redundancies. Thus, if we are able to eliminate the redundancies during the transmission, we can reduce network traffic and router congestion.

Audio mixing is another application for fusion: if we are accepting audio streams from several sources and would like to hear the mixed stream, fusing streams on intermediate hosts will spread bandwidth requirements and computation costs across the active network.

2.3 TCP over Active IP

TCP/IP are the prevailing transport and network layer protocols on traditional networks. The active network paradigm does not preclude the use of TCP/IP. An agent, carrying the packet data, will simply invoke the TCP functions that are resident on the host. Moreover, an active network can support a more flexible IP service, due to the active nature of the active network.

On an active network, IP packets will be replaced with agents. These agents may perform customized IP functions such as routing. In this way, we can have a TCP service built on an active IP.

On a traditional network, a server has to listen to a specific port for connection requests, and invoke certain applications when a request comes in. On an active network, however, no server has to wait for incoming requests, because a request comes with code, and the code may perform the task of a server.

3 T2 CAN System Model

In this section, we present our active network model: T2 CAN. As indicated by the name, the active network prototype consists of two parts: T2 (Tacoma Too) is a programming language for mobile code; CAN (Cornell Active Network) implements interpreters for the active networks.

3.1 T2: Tacoma Too

T2 is a dialect of Ocaml[6], a type-safe functional language developed at INRIA.

T2 adds some new features into Ocaml to support mobile code. In T2,*agents* correspond to active "packets" that roam on the networks. Agents carry both data and code, and store them in named *folders*. T2 provides *read* and *write* operations for agents to access folders.

T2 also supports *regions*, where agents can rendezvous and exchange information. A *region* is simply a function (or procedure) with a special mechanism for activation.

Each region has three components: a region name, a list of roles, and a piece of code to execute. A region is installed using the following function:

region_install *region_name role_list code*

Roles refer to agents, and are used by the region to access the folders of these agents. The code is dynamically loaded into the interpreter and invoked when the region becomes active.

A region is activated when other already active regions *meet* in that region:

meet *region_name agent role delay_option attenuation_domain*

A *meet* request provides a region name and an agent, along with the role the agent wishes to play. The last parameter, *attenuation_domain*, is a security parameter, and is described in Section 5.

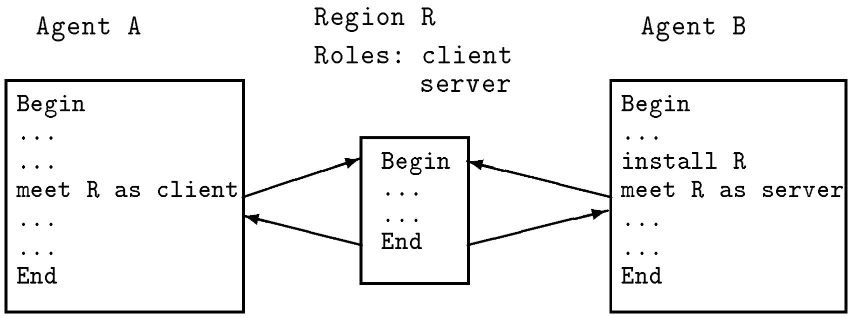

Fig. 1. Region and Region Activation

Figure 1 illustrates the concept of region and the control flow for a region activation. Agent A and agent B want to exchange information through region R. Region R has two roles: *client* and *server*. Agent B first installs region R, and then requests to activate R as *server*; while agent A requests to invoke region R as *client*. Agent A and agent B are blocked at the *meet* operations. Since region R has found both a *client* (agent A) and a *server* (agent B), it will be activated. We call a region in execution a *region activation*. After the region activation finishes executing the code of region R, the two requesters, region A and region B, will continue their executions.

It is clear from Figure 1 that an activation of a region is a n-way (n is 2 in this case) procedure call with synchronization of the requesters. T2 supports nested *meet*, i.e., region activations launched by *meet* requests can issue new *meet* requests. Nested *meet* works in the same way as nested procedure calls.

Normally, a region will be activated when all its roles have been filled with *meet* requests. Agents can use the *delay option* to specify how they would like to rendezvous in a region activation. An agent specifies the *delay option* as *Now* if it requires the region start immediately; or *Before t* if at most t time units of delay is allowed; or *Anytime* if there is no time constraint. This way, agents can achieve ADA-like rendezvous.

3.2 CAN: Cornell Active Network

We have implemented the active network interpreters, which we call CAN, in Ocaml. A CAN interpreter accepts agents carrying code written in T2, sets up secure environments for T2 code received this way, monitors execution of this code, and coordinates the possible interactions.

When an agent arrives on a host, the interpreter on the host extracts the default code folder of the agent, and activates the code for the agent. The execution of the code is regarded as the first region activation of the agent; the agent plays the only role *self* in this region activation. This region activation may install new regions with *region_install* and request to start new region activations with *meet*.

CAN interpreters also provides certain system services to agents, for example, an agent can fork by *meeting* in the *Sys.Fork* region. According to the agent's destination folder, a new copy of the agent agent is created on the same node, or is sent automatically onto a network link.

When an agent arrives, or when a region is installed, CAN compiles the provided code and loads the byte code dynamically into the interpreter. To improve performance, CAN caches the recently loaded code, and eliminates the compilation and dynamic loading costs when the code is used by other agents or regions.

4 Applications On T2 CAN

To prove the feasibility of T2 CAN, we implemented the applications described in Section 2 on T2 CAN. These examples also illustrate the security problems that we are interested in addressing.

4.1 Image Retrieval

In the image retrieval example of Section 2, a client wishes to get information from a remote image database using a customized image retrieval algorithm. On a T2 CAN active network, we can include the algorithm in an agent, and ship it to the database. The agent meets with a resident agent at the database to retrieve the images, and then invokes the algorithm to generate the results. When done, the agent travels back home with the result.

```
region Client.Main(Self):
       Self.retrieval := image retrieval algorithm;
       meet in DB.Transfer with self as Recipient;
       Self.destination := Self.home;
       meet in Sys.Fork with Self as Client;

region DB.Main(Self):
       install DB.Transfer with code of DB.Transfer;
       Self.data := ...;
       meet in DB.Transfer with Self as Provider;

region DB.Transfer(Provider, Recipient):
       Recipient.result := Recipient.retrieval(Provider.data);
```

Fig. 2. Image Retrieval

Figure 2 shows the pseudo-code for both the client agent (*Client.main*) and the resident database agent (*DB.main*), as well as the code for the *DB.transfer* region* . Region *DB.transfer* has two roles: *provider* and *recipient*. The client agent stores the code of the image retrieval algorithm in its *retrieval* folder, and requests to *meet* in the *DB.transfer* region as *recipient*. Whereas, the database agent stores the images in its *data* folder, installs the *DB.transfer* region, and then requests to *meet* as *provider*. The two agents will block at the *meet* requests, and the *DB.transfer* region will be activated. The new region activation will extract the algorithm from the *recipient*'s *retrieval* folder, and apply it to the data in the *provider*'s *data* folder. The result will be stored back to the *recipient*'s *result* folder. When the region activation finishes the image retrieval, the database agent and the client agent will resume. The client agent will then request to go home by *meeting* in the *Sys.Fork* region.

4.2 Fusion

We have implemented a simple fusion application on T2 CAN.

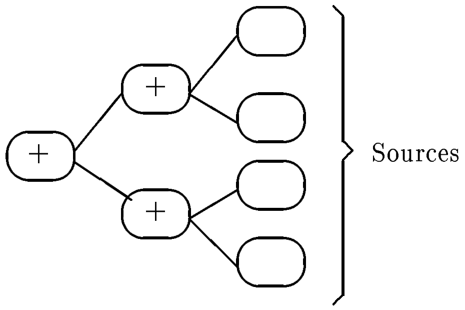

Fig. 3. Structure of a Fusion Tree

The tree structure of the hosts in this example is shown in Figure 3. The user is located at the root of the fusion tree, while all the sources are on the leaves.

First, we have to deploy the fusion protocol dynamically on the hosts from the root to the sources along the fusion tree. The protocol deployment is accomplished by the *deployment agents*.

Figure 4 shows the pseudo-code of a deployment agent. The agent starts at the root of the fusion tree. On every intermediate host, the agent finds the successor hosts on the fusion tree, and *meets* in the *Sys.Fork* to send a replica

* For simplicity, in the pseudo-code, we use assignments for *write* operations, and a folder on the right side of an assignment implies a *read* operation. We use this notation for all the pseudo code presented in this paper.

```
Deployment Agent:
    if (current host is an intermediate node)
    then
      Self.destination := the first successor host;
      meet Sys.Fork with Self as Client;
      Self.destination := the second successor host;
      meet Sys.Fork with Self as Client;
      install Fusion with code of Fusion region
    else
      Self.data := information requested;
      Self.code := code of the delivery agent;
      Self.destination := the predecessor host;
      meet Sys.Fork with Self as Client

Fusion (Source1, Source2):
    Source1.data := fuse(Source1.data, Source2.data);
    Source1.destination := the predecessor host;
    if (the current host is the root)
    then
      submit Source1.data
    else
      meet Sys.Fork with Source1 as Client

Delivery Agent:
    if (this agent comes from the left successor host)
    then
      meet Fusion with Self as Source1
    else
      meet Fusion with Self as Source2
```

Fig. 4. Fusion on T2 CAN

of the agent to every successor host, then on the current host, the agent deploys the fusion protocol by installing a *fusion* region on the host.

The *fusion* region is shown in Figure 4, and it has two roles: *source1* and *source2*. A *fusion* region activation fuses the data from both the *source1*'s *data* folder and the *source2*'s *data* folder, and stores the result back to the *source1*'s *data* folder. The activation submits the result if the current host is the root of the fusion tree; otherwise the activation forwards a replica of the *source1* agent to the predecessor host.

When a deployment agent arrives on a source node, it invokes a delivery agent to gather the information and to start delivering the information to the root. Figure 4 also shows the pseudo-code of a delivery agent. The delivery agents actually start the real fusion process, and will activate the *fusion* regions on the intermediate hosts till the root is reached.

TCP Region	Function
tcp_connect	Establish TCP connection
tcp_send	Send data over a TCP connection
tcp_recv	Submit Information to TCP from network layer
tcp_close	Close a TCP connection

Fig. 5. TCP regions

4.3 TCP over Active IP

In this example, we show how the traditional TCP protocol is realized on top of T2 CAN. The *tcp_connect, tcp_send, tcp_recv* and *tcp_close* regions form the interface of the TCP service. They all have one role, *client*, and thus behave the same way as functions, i.e., each region can be activated by a single "procedure call" (*meet*). Figure 5 lists the TCP regions with their functions.

To transfer data over TCP, an agent first *meets* in the *tcp_connect* region with the address and the port number of the destination host. A *tcp_connect* region activation then establishes the connection and returns a socket number to the agent. Now that the TCP connection is established, the agent may *meet* in *tcp_send* region to send messages. Finally the agent closes the TCP connection through a *meet* in the *tcp_close* region.

It is the IP agents who carry out all the underlying communication among hosts. Whenever TCP wants to send a packet to another host, it starts an IP agent filled with the address and the port number of the destination host, along with the content of the packet; and it is the IP agent's task to route all the way to the destination host. Upon arrival on the destination host, the IP agent results in a *tcp_recv* region activation and submits the packet to TCP. Because IP agents are programmable, users can specify their IP agents to perform customized routing.

Since it is active agents, instead of passive packets, that arrive on the destination host, a server on the destination host does not need to bind a socket to a port and listen to the port. TCP on the destination host, when receiving the packet, will activate the appropriate application to handle the messages.

4.4 Security Problems

On a T2 CAN active network, hosts do not necessarily trust agents, and agents may not trust each other either. In order to protect themselves from security attacks, hosts and agents would like to specify certain security policies.

In the image retrieval example, The database may want to restrict the execution of the untrusted image retrieval algorithm in a variety of ways to prevent it from modifying image data, or from sending back more data than allowed. The database provider may put an upper limit on how much data an agent can ship back home, or charge the owner of the agent based on the size of the result.

In the fusion example, the user that requests a fusion may want to prevent other unrelated agents from merging in unwanted data, or stealing data on intermediate hosts. The hosts in the fusion tree also do not want the fusion agents to do any harm to the hosts themselves.

For the TCP example, we have to give IP agents the access to the local routing table so that the IP agents can use the information for routing. Because IP agents are not necessarily trusted, they should not be empowered with full access to the routing table.

To address these security problems, we implemented a security framework within T2 CAN. Section 5 and Section 6 will present the security framework in detail, and we will show solutions to the security problems mentioned above in Section 7. As stated in the introduction, the T2 CAN security model only deals with host security and inter-agent security problems. We assume that hosts are trusted and that communication links are cryptographically protected.

5 T2 CAN Security Model

T2 CAN provides a security model to enforce security policies. *Security domain*, which defines a set of allowed operations†, is the key concept of the T2CAN security model. For security policies that essentially constrain the sequence of operations that can be executed, a security model with mechanisms to enforce domains and change domains is sufficient to implement any such security policy.

All execution in T2 CAN is conducted by region activations (as said before, an initial execution of an agent is also regarded as a region activation). Each activation executes with respect to a *security domain*, which restricts what operations this activation may perform.

Domain enforcement is more as an implementation issue, and we will talk about it in detail in Section 6. In this section, we will discuss domain changes in the T2 CAN security model.

Obviously, region activations cannot arbitrarily modify the associated domains. On the other hand, the interpreter should give agents certain flexibility to change their domains for their needs, under the precondition that no security policies are compromised. Two basic forms of continuous domain changes are *amplification* and *attenuation*.

For security domains D_1 and D_2, we call D_1 an *amplification* of D_2, or D_2 an *attenuation* of D_1, if $D_2 \subseteq D_1$.

Obviously, $D_1 \cap D_2$ is an attenuation of D_1 and D_2, and $D_1 \cup D_2$ is an amplification of D_1 and D_2. We now describe how domains are assigned and changed.

† The concept of *operation* here is very general, and its meaning depends the context of the corresponding security policy. For example, when we talk about file systems, the operations can be: reading a certain file, writing a certain file, creating a certain file and so on.

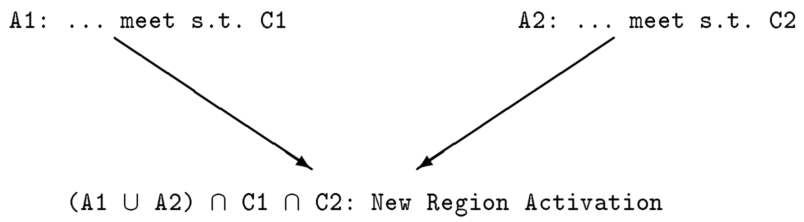

Fig. 6. Domain For a Region Activation

5.1 Initial Domain Assignment

As explained before, when an agent arrives, a region is extracted and activated. The domain that the interpreter assigns to this region activation consists of operations to read and write this agent's folders, and the *region_install* and *meet* operations.

5.2 Domain for New Region Activations

Domain change occurs when a new region activation starts. Figure 6 illustrates how the domain for a new region activation is constructed. The first region activation, who is in domain A_1, specifies C_1 as the attenuation domain in its *meet* request. The second region is in domain A_2, and specifies C_2 as the attenuation domain. The domain for the new region activation will be $(A_1 \cup A_2) \cap C_1 \cap C_2$. That is, the new region activation gets the union of the domains of the requesters. This an amplification of both A_1 and A_2. However, the requesters can attenuate the new domain using C_1 and C_2.

More generally, the security domain of the new region activation follows *the Amplification and Attenuation Rule*:

For region activations in security domains A_1, A_2, ..., and A_k, and given the attenuation domains C_1, C_2, ..., and C_k specified by the *meet* requests from these region activations, the security domain for the new region activation is

$$(\cup_{i=1}^{k} A_i) \cap (\cap_{i=1}^{k} C_i).$$

A more concrete example for *the Amplification and Attenuation Rule* is presented in Figure 9 of Section 7.

Another reasonable candidate for the rule would be: $\cup_{i=1}^{k}(A_i \cap C_i)$. The difference lies in the fact that our definition gives any region activation the ability to restrict, through the attenuation domain, not only its own security domain, but also those of other region activations. The examples shown in Section 7 illustrates why this ability is desirable.

When the new region activation runs to the end, all the requesters resume in their original domains.

A1: ... meet s.t. C1 A2: ... meet s.t. C2

(A1 ∪ A2) ∩ C1 ∩ C2: ... meet s.t. C12 A3: ... meet s.t. C3

C1, C2 transient: (((A1 ∪ A2) ∩ C1 ∩ C2) ∪ A3) ∩ C12 ∩ C3

C1, C2 permanent: (A1 ∪ A2 ∪ A3) ∩ C1 ∩ C2 ∩ C12 ∩ C3

Fig. 7. Domain for Nested Region Activation

5.3 Domain for Nested Region Activations

As we said earlier, T2 CAN supports nested *meets*. The security domain for a nested region activation can be constructed in different ways, depending on the effective period of the attenuation domains. An attenuation domain can be *permanently* effective, which means that it will serve as an attenuation domain even for nested activations; or, an attenuation domain can be *transient*, which means it applies only to the immediate activation.

Figure 7 shows how the domain for a nested region activation is constructed. Formally, suppose we have region activations in domains A_1 and A_2, and they *meet* in a region with attenuation domains C_1 and C_2. This new region activation further *meets* with another region activation in domain A_3, with attenuation domains C_{12} and C_3. If C_1 and C_2 are specified as *permanent*, then the domain for the nested region activation is $(A_1 \cup A_2 \cup A_3) \cap C_1 \cap C_2 \cap C_{12} \cap C_3$; if C_1 and C_2 are *transient*, then the domain is $(((A_1 \cup A_2) \cap C_1 \cap C_2) \cup A_3) \cap C_{12} \cap C_3$. Agents can specify the attenuation domains as *permanent* or *transient* according to their security concerns.

5.4 Security Automaton

Consider the following policy:" An agent cannot send messages after reading local files." The domain associated with the agent changes after the agent reads local files. We can enforce this policy by forbidding any message sends, or disallow the agent to read local files. However, the best solution will be to allow domains to change dynamically.

Security automaton[10] is a tool for both specification and implementation of dynamic domain changes. A security automaton is a nondeterministic finite automaton. Each state in the automaton specifies a domain, and domain changes are specified in the transition functions. A formal definition of security automaton can be found in [10].

Figure 8 shows a security automaton that implements the policy mentioned above.

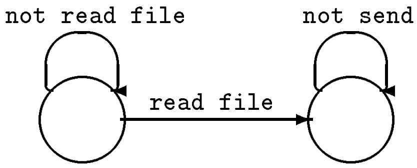

Fig. 8. A Security Automaton

We believe that security automata are a general way to specify and enforce security policies, and we are integrating security automata with other domain change mechanisms.

6 Domain Enforcement

We have talked about how domains are assigned and changed in T2 CAN. Now we come back to the implementation issue of domain enforcement.

First, domains usually specify operations (including target objects) in a logical name space, whereas region activations execute in a physical address space. We rely on a safe language to provide a mapping from the name space to the address space. Since the mapping is safe, an execution cannot escape the address space that corresponds to the visible name space of the execution.

Secondly, we can control the visibility of names to region activations statically, so that they have no access to unsafe features. Abstract types allow us to hide representation details of a type; module visibility can exclude the visibility of certain modules from dynamically loaded code; module thinning[9] exports only a desired subset out of a module; finally, sparsely chosen, secret names help to make certain names visible to a selected group of activations who know the secret names.

We have to give region activations the access to functions like *read*, *write*, *meet* and *install_region*. In order to enforce security policies concerning these operations, we have to control the interpretations of these names. More specifically, the security-sensitive operations[‡] that an region activation can invoke are dynamically wrapped with guards, which perform security checks.

By wrappers, we have actually implemented reference monitors that govern the executions of region activations. However, agents may still circumvent reference monitors through covert storage channels. T2 CAN takes care of this problem by removing shared data among region activations.

In this section, we will describe these enforcement mechanisms in detail.

[‡] Security-sensitive operations are those operations that security policies concern about, for example, when we specify policies on file accesses, the security-sensitive operations are creating files, reading files, writing files, deleting files and so on. In T2CAN, the security-sensitive operations are those that read information from an agent, write data into an agent, install a region and meet in a region.

6.1 Safe Language

Using a safe language is the basis for enforcing domains. We have implemented the prototype in Ocaml [6], and the prototype supports agents carrying programs written in the T2 source code (unlike Java byte code, Ocaml byte code cannot be verified for safety).

Ocaml is a programming language with a strong type system, that is, Ocaml does not support any type casting or pointer arithmetic. Ocaml has a built-in garbage collector, and does automatic memory management.

Ocaml not only supports dynamic loading, but also provides an interface compatibility check during dynamic loading. All the modules that the mobile code has been compiled with should have the same interfaces as those modules provided by the interpreter. This ensures that there will be no type mismatch due to dynamic loading, so that malicious agents cannot try to circumvent these interfaces by using different versions of an Ocaml compiler.

In T2 CAN, agents roam on the network with typed data. We have to enforce type safety on those typed mobile data. Ocaml does have a special *Obj* library that provides unsafe functions to cast types. Obviously, we cannot provide this module to agents, yet we need a way to transfer the data with types. Therefore, we have extended Ocaml with a safe marshaling and unmarshaling facility. For this, we made a small extension of the Ocaml language to specify the type when unmarshaling data. We implemented a translator that rewrites the code into pure Ocaml, along with additional code that checks that the data is of the required type.

Because safe marshaling and unmarshaling functions are based on the unsafe *Obj* library, we cannot totally exclude the access to the *Obj* library from mobile code. However, we have to prevent mobile code from using the unsafe functions in the *Obj* library directly. Since programs cannot access a module without knowing its name, we link the *Obj* module with a secret and sparsely chosen module name. In this case, only trusted and system-generated functions (like those safe marshaling and unmarshaling functions), which have knowledge of the secret name, can access these unsafe functions. As it were, the secret module name serves as a capability for unsafe features.

6.2 Static Name Visibility Control

To enforce domains, T2 CAN further exploits the visibility control mechanisms in Ocaml, such as abstract types and module thinning, to control statically the names that are visible to a region activation.

Abstract types hide the representation of the types, and hence forbid direct access to the inner structures of the types. Using abstract types, we can encapsulate a type and export only the functions that are allowed to be performed on the type. Details of an abstract type are invisible to users, so the exported functions are the only ways to access the data of the abstract type. Abstract types are not only useful for the construction of the restricted environments for the region activations, but are also essential to guarantee that security wrappers can not be circumvented.

Module thinning is a method that we use to remove the unsafe features from existing libraries to construct safe environments for region activations. For example, the *array* module in Ocaml contains unsafe functions, such as array access without any bound checks, for efficiency. With module thinning, we can export a safe array library with these unsafe functions excluded.

In addition, Ocaml provides a mechanism for the main program to specify the set of modules that are visible to dynamically loaded code. Because all regions are created through dynamic loading, we can take advantage of the mechanism, and export only the necessary and safe modules to regions during dynamic loading. For example, the *Unix* module, which implements all the system calls, is always invisible to region activations.

Using sparse names is a way to give a selected group of region activations the access to a certain operation (or object), because only these activations have the knowledge of the sparse names. As stated earlier, we have used this mechanism to realize safe marshaling and unmarshaling. Sparsely chosen, secret region names can also be adopted by cooperating agents to prevent others from interfering with the interactions. This technique is used to build a secure fusion application, and will be shown in Section 7.3.

6.3 Controlled Interpretation of Names

Domains are dynamic: different agents have different domains, and these domains can change during the run time. For example, An agent has an initial domain that allows the read and write operations on its own folders; the domain changes when a new region activation starts. In T2 CAN, we further control the interpretation of names to enforce domains dynamically.

Each region activation, when it starts, is endowed with a set of operations, such as *read*, *write*, *meet*, and *install_region*. However, T2 CAN controls the interpretation of these operations. When an activation performs a given operation, it may invoke the function that T2 CAN associates with the name for the activation. To be more concrete, T2 CAN adds a security wrapper to the security-sensitive operations dynamically, and gives the wrapped functions, instead of the original functions, to region activations. The security wrapper will check if the operation is in the domain, and only proceed when the operation is allowed; otherwise the wrapper will raise a security exception. In other words, we have actually implemented a custom reference monitor in T2 CAN.

However, agents may explore to circumvent reference monitors through covert channels. We present our solution to covert storage channels next.

6.4 Covert Storage Channels

In T2 CAN, agents can exchange information through *read* and *write* operations, which are supervised by the security wrappers. However, agents may exploit other ways to transfer information through a memory location without reference monitors knowing about the information flow.

Since T2 CAN is based on a safe language. Agents can share a memory location only if they share a name. To block covert storage channels, we just have to make sure that region activations do not share names. In a normal program, the region activations do share global variables and static variables. We implemented a *deep copy* routine, which makes a fresh copy of the closure of the function when a region is activated. In this way, each region activation has its own copy of global and static variables, and any data sharing is eliminated.

Deep copy may seem costly, but in the applications we build, there are only a very small number of data to be copied.

Covert timing channels are much harder to block, and we will briefly touch on this topic in Section 8.

7 Applications on T2 CAN Revisited

Now that we have discussed the T2 CAN security model, and how the mechanisms are enforced, we now revisit the examples in Section 4 to see how T2 CAN solves their security problems. Because of the safe language and the static visibility control, agents can no longer perform unsafe operations. Security policies will mainly deal with monitored operations like *read*, *write*, *meet* and *install_region*. We will concentrate our discussion on these operations.

7.1 Secure Image Retrieval

In the image retrieval example, the database server wants to make sure that the client agent cannot compromise the integrity of the database, and a client can only take home a very limited amount of data. On the other hand, the client agent wants to be assured that the database agent cannot learn anything from the client agent during the *DB.Transfer* region activation.

Figure 9 shows the modified pseudo-code of Figure 2 for a secure image retrieval. The details of A_1, A_2, C_1 and C_2 are also shown[§]. A_1 allows the *Self* agent to read and write its own folders, so does A_2. To make sure that the DB agent cannot learn anything in the *DB.Transfer* region activation, the client agent excludes any write operations on the *provider*(which will be the DB agent)'s folders[¶]. For the DB agent, the attenuation domain C_2 excludes any write to its own folders, which protects the database. C_2 also states that only the *recipient*'s *result* folder is writable, and thus the limit check can be performed only on the *result* folder. The resulting domain A is also shown in Figure 9.

[§] We only consider *read* and *write* operations in this example.

[¶] As we can see, the attenuation domain specified by the client agent restricts not only its own security domain, but also that of the other agent. Without the ability to restrict other agents' security domains, we will never able to achieve the policy we desire for this new region activation in this example. That is the main reason we did not select $(\cup_{i=1}^{k}(D_i \cap A_i))$ as the amplification and attenuation rule.

```
region Client.Main(Self):
       Self.retrieval := image retrieval algorithm;
A1:    meet in DB.Transfer with self as Recipient s.t.C1;
       Self.destination := Self.home;
       meet in Sys.Fork with Self as Client;

region DB.Main(Self):
       install DB.Transfer with code of DB.Transfer;
       Self.data := ...;
A2:    meet in DB.Transfer with Self as Provider s.t. C2;

region DB.Transfer(Provider, Recipient):
A:     Recipient.result := Recipient.retrieval(Provider.data);
       if (sizeof(Client.result)> Limit) then raise Exception

A1:{read Self.*; write Self.*}
A2:{read Self.*; write Self.*}
C1:{read Recipient.retrieval; read Recipient.result; read Provider.*;
   write Recipient.result}
C2:{read Recipient.*; read Provider.data; write Recipient.result }
A = (A1 ∪ A2) ∩ C1 ∩ C2
  = {read Recipient.match; read Provider.data; write Recipient.result}
```

Fig. 9. Secure Image Retrieval

7.2 Secure TCP over active IP

As we mentioned in Section 4, one security problem we came across in the TCP example is how to give IP agents access to the local routing table without compromising the integrity of the routing table.

We have introduced a local routing agent, which has access to the routing table. An IP agents may *meet* with the routing agent with its routing algorithm. With domain amplification, the IP agent gets access to the routing table in the new region activation. On the other hand, the routing agent can specify an appropriate attenuation domain to preserve the integrity of the routing table. The region activation may invoke the routing algorithm for the IP agent and store the result back to the agent. Since the routing algorithm executes within the constraints imposed by the routing agent, security policies can be properly enforced.

7.3 Secure Fusion

In the fusion example, we are concerned about unrelated agents getting in the way of the fusion process: they may inject unwanted information or steal fusion information on the intermediate hosts.

Rather than using a well-known region name, a fusion application uses a sparsely chosen, secret name to set up meetings between cooperating fusion agents. Unrelated agents, without knowing the secret region name, are not able to interfere. Again, amplification is used to transfer fusion data between the agents. However, no attenuation is necessary, because all the agents that *meet* know the secret region name, and thus are trusted.

8 Future Work

We are working on the integration of security automata into the T2 CAN security framework. A prototype of security automata has been implemented, and we are investigating how the prototype can live in harmony with the existing T2 CAN domain change mechanisms.

We are carrying out research on thwarting denial of service attacks towards active networks. In the current implementation of T2 CAN, we simply associate with each agent a fixed amount of resources. For example, an agent, or more precisely, the region activation that works for the agent, can execute on a host for at most 3 seconds. If an agent uses up the allocated resources, it will be automatically terminated. We are considering using admission control and resource management, which have been studied in the area of providing Quality-of-Service on networks.

Covert timing channels is another topic under investigation. Malicious agents can perform some external visible operations at a certain time, thus sending out information through a covert timing channel. Instead of imposing firm and impractical restrictions over agents, we are seeking ways, such as certain scheduling schemes, to overcome the problem.

9 Related Work

9.1 Active Networks

MIT's ANTS[14], the University of Pennsylvania and Bellcore's Switchware[11], and the University of Arizona's Liquid Software[5] are some of the pioneering research projects in the area of active networks.

In ANTS, passive packets in traditional networks are replaced by active capsules. Each capsule indicates the protocol that should be applied to it. ANTS also has a demand loading protocol for protocol installation. ANTS relies on the safety mechanisms for mobile code, which include sand-boxing and Java byte code verification. Furthermore, the association between a capsule and its protocol is cryptographically protected; i.e., the reference to the protocol is based on a fingerprint, such as the MD5 message digest of the protocol code. There is also a resource limit field which works like the generalized TTL (Time-To-Live) field in IP packets.

ANTS does not fully address the security problem that an active network faces, instead, it adopts a simple security model: all the capsules are statically restricted about what they can do. As in T2 CAN, ANTS selects a safe language to enforce the restriction, but there is no dynamic mechanism for domain change and domain enforcement.

Switchware[11, 3, 4] is the joint work of the University of Pennsylvania and Bellcore on programmable switches. In [11], the Switchware group gives an overview of the Switchware project and explores security issues in detail. It proposes a multiplexing architecture for resource protection and investigates tradeoffs between OS and Compiler-enforced Access Control.

In addition, the paper discusses how to exploit safety features in the programming languages, such as Java and SML. The paper also suggests using message hashing techniques for authentication of type-checked modules. The Active Bridge[3] presents a simple prototype for programmable bridges. The security model is based on Caml, which, like its successor Ocaml, has a strong and static type system and an automatic dynamic storage allocation mechanism. Besides the Caml-based type safety, the active bridge prototype further uses module thinning to export a limited interface to untrusted code.

As to the language, Switchware and T2 CAN are similar. However, Switchware only uses the type safe language to construct a safe environment for the mobile code. There is no further enforcement mechanisms like reference monitors that allow fine-grained specification and enforcement of security policies.

Furthermore, neither ANTS nor Active Bridge take into account information flow or covert channel analysis, and thus, are weak at preserving privacy.

PLAN[4] was released recently as a new programming language designed specifically for active networks. It bears some resemblance to ML. Due to lack of detailed information, we are unable to evaluate PLAN in further detail.

Liquid Software [5] is another infrastructure, proposed by the University of Arizona for dynamically moving functionality throughout a network. The paper proposes three categories of solutions to the safety problem: user control, implicit trust, and verified access. User control requires users to decide whether or not to grant a request to access system resources; implicit trust simply trusts the foreign code if it originates from a trusted entity, and the producer can digitally sign the software so that other hosts can verify the originality and integrity. As pointed out in [5], low level checks will still be necessary to guard against program errors. Unfortunately, we are not aware of any report of an implemented active network framework for us to evaluate.

9.2 Mobile Code Technologies

The key feature of active networks is the support of mobile code, and research on mobile code provides enabling technologies, especially those on safety and security, for active networks.

JAVA[1] is both a programming language and an application platform designed by Sun Microsystems. A JAVA program can be compiled into the JAVA byte code; the byte code can be transferred to another host on the network and executes in the JAVA Virtual Machine on that host. To ensure that these untrusted foreign JAVA programs(also called applets) will not do any harm to the hosts, the JAVA Virtual Machine puts each applet in a sand-box, which guarantees that the applet cannot gain unauthorized access to system resources and other sensitive information.

Omniware[2] is a mobile code system that supports safe and mobile C/C++ programs. Because C/C++ programs are inherently unsafe, Omniware achieves safety using a technique called Software-Based Fault Isolation(SFI) [13]. The main idea of SFI is to insert security checks to a piece of untrusted code before each access to host resources, and these run-time checks ensure that the untrusted code will not escape the sand-box that the host would like to enforce.

Proof-Carrying Code [8]is a new technique for attesting that a piece of untrusted code satisfies certain safety policies: the code producer is required to provide, along with the code, a safety proof that testifies the code respects the required safety policies; on the other hand, the host, where the code executes, can validate the safety proof, and be certain that the foreign code is safe to execute.

In T2 CAN, we use Ocaml, a dialect of ML. It has several advantages over the other alternatives. Unlike JAVA, Ocaml has a solid theoretical basis, which improves the users' confidence on the correctness of Ocaml's strong and static type system. Most type-safety checks in Ocaml can be performed statically, and thus reduces the run-time overhead, whereas in Omniware, all the checks have to be carried out during the run-time. Proof-Carrying Code is a promising technique, but is still far from being practically used.

Although T2 CAN is currently built upon Ocaml. The domain-security model is language independent, and it can be implemented by any languages listed above. We are currently investigating the implementation of a secure active network framework over JAVA and Omniware, and hope to get more experience about the tradeoffs among the languages.

References

1. Secure Computing with Java: Now and the Future. http://www.javasoft.com/security/javaone97-whitepaper.html, April 1997.
2. Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and Language-Independent Mobile Programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 127–136, Philadelphia, PA, May 1996.
3. D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active Bridging. In *Proceedings of the ACM SIGCOMM'97 Conference*, Cannes, France, September 1997.
4. Carl A. Gunter, Mike Hicks, Pankaj Kakkar, Jon Moore, Scott Nettles, and Sam Weber. PLAN. http://www.cis.upenn.edu/~switchware/PLAN.
5. John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting. Liquid Software: A New Paradigm for Networked Systems. Technical Report TR-96-11, University of Arizona, June 1996.
6. INRIA. Object caml. http://pauillac.inria.fr/ocaml/.
7. Yaron Minsky, Robbert van Renesse, Fred. B. Schneider, and Scott Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the*

Seventh ACM SIGOPS European Workshop, pages 109–114, Connemara, Ireland, September 1996.
8. Georage Necula. Proof Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Pronciples of Programming Languages*, pages 106–119, Paris, France, January 1997.
9. François Rouaix. A Web Navigator with Applets in Caml. Fifth International World Wide Web Conference, May 1996.
10. Fred. B. Schneider. Towards Fault-Tolerant and Secure Agentry. Invited paper, to appear in Proceedings of the 11th International Workshop on Distributed Algorithms, Saarbucken, Germany, September 1997.
11. Jonathan M. Smith, David Farber, Carl A. Gunter, Scott Nettles, Mark Segal, William D. Sincoskie, David Feldmeier, and Scott Alexander. SwitchWare: Towards a 21st Century Network Infrastructure. http://www.cis.upenn.edu/~switchware/papers/ieee.ps.Z.
12. David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.
13. Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, December 1993.
14. David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. Submitted to IEEE OPENARCH'98, San Francisco, CA, April 1998.