

An Agent Based Architecture for Supporting Application Level Security*

Zhaoyu Liu, Prasad Naldurg, Seung Yi, Tin Qian, Roy H. Campbell, M. Dennis Mickunas
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{zhaoyu, naldurg, seungyi, ting, roy, mickunas}@cs.uiuc.edu

Abstract

The heterogeneous nature of distributed systems raises many security issues and concerns. Traditional systems cannot provide customized security policies and mechanisms for heterogeneous applications. Historically, applications have relied on a static security architecture to provide ad-hoc security guarantees. In this paper we propose a new security architecture based on mobile agents for applications in distributed environments. Our approach allows applications to create and enforce customized policies at run time. These policies and access control requirements can be specified using programs. In addition our framework can handle dynamic requests to change or update these policies and adapt to situational requirements.

1 Introduction

Traditional systems provide security mechanisms to ensure that system resources are used and accessed as intended. They also attempt to detect and prevent accidental or intentional misuse. Typically, these mechanisms tend to be static and it is very difficult to change the security policy or the mechanisms, once the system is installed. Researchers have developed a number of new techniques and mechanisms [7, 5, 10, 9, 15] but very few systems provide support to incorporate these changes.

In a distributed computing environment, applications and users have varying security requirements. In existing systems, these applications or users have very little choice regarding the type of policy or security mechanism and must rely heavily on the underlying infrastructure to provide security guarantees. For example, delegation and security management are severely constrained by static security mechanisms. With systems that support ubiquitous

computing devices, it is reasonable to expect that different devices will need different guarantees from the underlying security infrastructure. Traditional static security mechanisms may not be expressive or flexible enough to meet the specific needs of a particular application or device. In order to provide applications (here we mean both users and devices) the ability to customize their security, we need a distributed security architecture that

- is capable of supporting various policies and mechanisms
- can add, replace or revoke policies and mechanisms
- allows applications to specify the kind of security guarantees they want from the system, on the fly
- dynamically enforces these customized policies and mechanisms
- restricts the use of policy to applications and systems that need to know the policy

In this paper we propose an architecture solution that uses mobile agents to provide the required functionality. In our design, these mobile agents are called active capabilities (ACs) [4, 1]. Basically these ACs are signed code fragments that are used to specify policies and mechanisms. Other components of our architecture provide the framework required to evaluate and enforce the policies specified by these ACs and to provide run time revocation, update and enforcement of these policies and mechanisms.

This paper is organized as follows. Section 2 gives a brief overview of the architecture and the trust model and explains each component in detail. Section 3 gives a description of different application scenarios that demonstrate the advantages of using our

*This research is supported by DARPA F30602-98-1-0192 and F30602-97-1-0281

architecture. Section 4 describes two specific implementations of this general architecture. Section 5 presents the preliminary performance results and the discussion on our implementation overhead. The last section presents our conclusions.

2 Architecture Description

This section describes the major components of our architecture and describes the trust model that forms the basis of their interaction. The most important component of our architecture is the active capability (AC). An AC carries a concise representation of security policies and mechanisms, customized or tailored for a particular application or device. The other components in our architecture include AC management, the software framework and the evaluation/enforcement engines. AC Management consists of a distributed network of AC Administrators, software framework component repositories and AC servers. These management entities are trusted. The AC Administrator is responsible for verifying, validating and certifying the code inside the AC, and for signing it.

The AC Administrator can additionally manage the distribution of the ACs using a secure channel. Alternatively, applications may contact the AC server and obtain these ACs and embed them in their code. Trusted applications may be allowed to create their own ACs. Each protection domain typically has one or more AC Administrators that are collectively responsible for the integrity of the ACs.

The AC can use a software framework for context. For example, a software framework may include a hierarchical structure of object-oriented classes of standard security policies. Typically this framework is componentized and arranged so that the components themselves can be downloaded from the software component framework repository using a secure channel. Each node, which is typically a computing device, has an evaluation/enforcement engine in its trusted address space. This engine can also be customized according to the context and its components can be downloaded using the secure channel dynamically. The ACs are evaluated in the sandbox-like environment provided by this engine by instantiating the context of the software framework, which also enforces the result of this evaluation. The subsections that follow give a detailed description of these components.

2.1 Active Capabilities

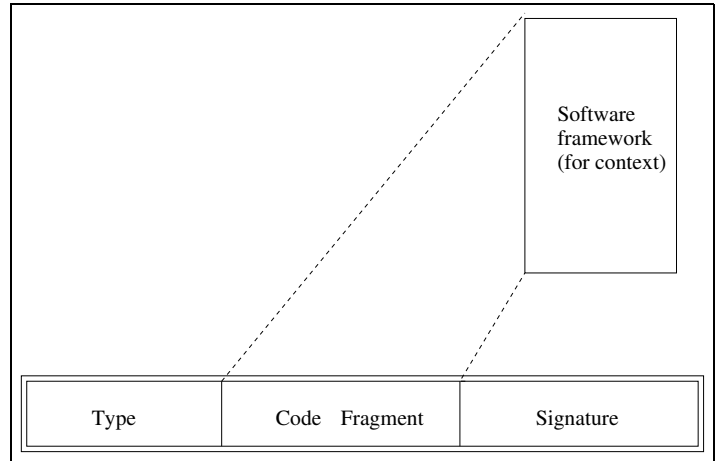


Figure 1: Generic Active Capability

The format of a generic active capability is shown in Figure 1. The first field is similar to a header and contains information about the **type** of the active capability. For instance, the AC type `ACCESS_CONTROL` indicates that the AC encodes an access control policy.

The second field is the most important part of the AC. Ideally it can contain an arbitrary piece of code, that specifies the policies or mechanisms for a particular user. The AC may carry all the code needed to make a policy decision or implement a particular mechanism. However, this approach would be too heavyweight. Instead we advocate the use of a software framework that can provide a context for the code field. The AC code can use the components of this software framework and impose additional constraints on their usage, leveraging the expressive power of the underlying framework.

The next section describes one particular instance of this framework, which implements types of access control policies in a modular and composable fashion [8]. The AC then uses the interface exported by this framework to create a code fragment that concisely represents one particular customized access control policy. In addition, the AC can use the features provided by the underlying language and add conditional processing, based on timestamps, or accumulated credits, to specify timeouts and to impose limits on resource utilization, etc. The flexibility afforded by this approach is limited only by the language syntax.

The last field is the digital signature. Typically the AC is created by an AC administrator or a trusted entity. This entity is responsible for the in-

egrity of the capability, and attests to this by signing the capability. In our distributed architecture, each protection domain has one or a small number of replicated AC administrators. The key distribution and management is simple. If we use a public key infrastructure, we need only one key-pair for AC Management. The administrator(s) can sign the message digest of the AC code using its private key and distribute the corresponding public key within their protection domain. The evaluation/enforcement engines can verify this signature using the public key of the AC administrator. This approach scales well and simplifies trust management.

The AC provides an interface that exports at least the following methods:

- `allowed`
- `revoke`
- `delegate`
- `bind`

The `allowed` method is called by the evaluation/enforcement engine. This method causes the code in the AC to be evaluated. This method returns a boolean value that controls the enforcement of the policy or mechanism requested by the application.

The `revoke` and the `delegate` methods specify interfaces to implement various revocation and delegation strategies. The implementation of these strategies can be customized to suit individual applications. The `bind` method is used to bind capabilities to applications and aids in the retrieval of the context in the evaluation/enforcement engine. This list is not exhaustive and additional methods can be added to extend the functionality and create new AC types.

2.2 Software Framework

This section describes a particular example of a software framework that can be used to provide a context for the active capabilities [8]. Traditional security systems are designed to enforce one particular security policy. In order to provide users more flexibility in terms of access control policy specification, we have implemented a composable and extensible object-oriented policy framework in Java. This framework has a GUI front-end that simplifies the process of specifying the policies. This allows

users and commercial organizations to specify access control policies tailored to their specific operational needs. The motivation and the functionality exported by this framework are explained in detail in the subsections that follow.

2.2.1 Access Control Policies

Access control is the mechanism by which a security system exercises control over the access and utilization of shared resources. Historically access control has been defined in terms of $\langle \textit{subject}, \textit{object} \rangle$ tuples and access control matrices. Typically the matrix is indexed by the name of the user (the subject) and by the resource that needs to be protected (the object). The intersection of this pair contains a Boolean value that indicates whether the access is allowed or denied. (The method is usually encoded implicitly in the Boolean value.) For example, Unix file systems use 3 bits to encode various combinations of read, write, and execute permissions for files. However, this matrix method of implementation is not sophisticated or flexible enough and does not scale when the systems serve a large number of users with a large number of resources.

The security policy associated with an access control mechanism refers to the characteristics of its specification, implementation and enforcement. Four different types of access control policies have been defined in literature. They include Mandatory Access Control(MAC), Discretionary Access Control(DAC), Double Discretionary Access Control (DDAC) and Role Based Access Control (RBAC).

The simplest form of access control is DAC. The matrix model is an implementation of this type of policy. Typically a DAC policy implementation maintains an indexed list of allowed $\langle \textit{subject}, \textit{object}, \textit{operation} \rangle$ triples. DDAC maintains two lists, an “allowed list” similar to DAC and a “denied list”. MAC policies use the concept of labeling. MAC is used by trusted operating systems, and every entity in the MAC system is assigned an immutable label. A hierarchy is defined in terms of these labels, and access control is enforced by comparing the labels. Subjects with higher labels have access permissions over objects with equal or lesser labels using a “no read up, no write down” rule [6]. This hierarchy strictly controls the flow of information.

Among these, RBAC is the most flexible type of access control policy [14]. All RBAC subjects are assigned roles. Each role represents a particular set

of objects and the allowed operations on each object. The major benefits of this aggregation are the considerable saving in terms of space and simplification in terms of management and enforcement. RBAC allows users to create policies with more sophisticated specifications than simple DAC, DDAC or MAC. A single user may have many different roles, and different permissions depending on the current role. Different constraints related to role and privilege may be enforced in RBAC.

Traditional systems provide a static implementation of any one of these access control mechanisms. Different applications with different access control policies cannot co-exist. Typically, applications cannot be ported across different systems without compromising the security guarantees offered by their access control mechanisms.

2.2.2 Policy framework

The policy framework itself is a hierarchy of classes as shown in Figure 2. It is dynamically configurable and extensible. The classes at the bottom of the framework are mostly abstract, and are mainly used to represent mathematical concepts such as sets and mappings. These classes form the basis for a hierarchy of successively specialized classes representing concepts such as labels and access control lists. Finally, at the top of the framework are classes, which can be used to represent a variety of generic policy forms [13].

Any policy framework that places a heavy burden on its users has never been popular. With this in mind, our policy framework GUI makes the process of creating new policies for ordinary users as painless as possible. Typically, to create new ACs, most users will simply select from a list of predefined policies or will use default settings chosen by a system administrator. However, it is necessary for system administrators and expert users to create and modify policies that respond to specific application needs or security threats. Therefore, our policy framework supports the enforcement of predefined policies efficiently and effortlessly, and also provides a convenient interface for policy authors to create more sophisticated, customized, and situational policies.

The policy framework supports all the following common types of access control: Mandatory(MAC), Discretionary(DAC), Double Discretionary(DDAC), and Role-based(RBAC). It is easy to extend our object-oriented framework to create more fine-grained, application specific policies. ([8] provides

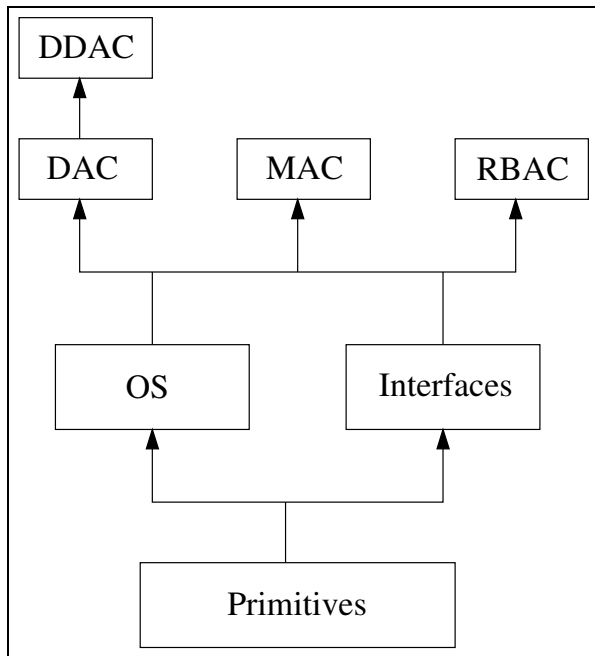


Figure 2: Component-level Map of the Policy Framework

several good examples). In our model, we can specify not only the $\langle subject, object, operation \rangle$ access control triple, but we can also include a resource limit on usage, situational decision rules, constraints and dependencies, e.g., based on current time of day or current role of the principal. The policy framework also lets users specify pre-conditions and post-conditions. Pre-conditions allow necessary security checks to be performed before evaluation takes place, and post-conditions can be used to maintain state and to perform additional checks after evaluation has been completed and when more information becomes available. The central feature of this framework is that the administration of these policies is built into the active capability and the underlying architecture itself. The section on AC management explains this process in detail.

2.3 Evaluation/Enforcement Engine

This section describes the evaluation/enforcement engine component of our architecture. Figure 3 shows a pictorial representation of this component.

The evaluation/enforcement engine consists of an AC cache, run-time resolvable references to customizable AC evaluation sandboxes, and run-time resolvable references to a customizable, componentized software framework. The AC cache is used to cache capabilities that do not change very often and provides a fast processing path for commonly used

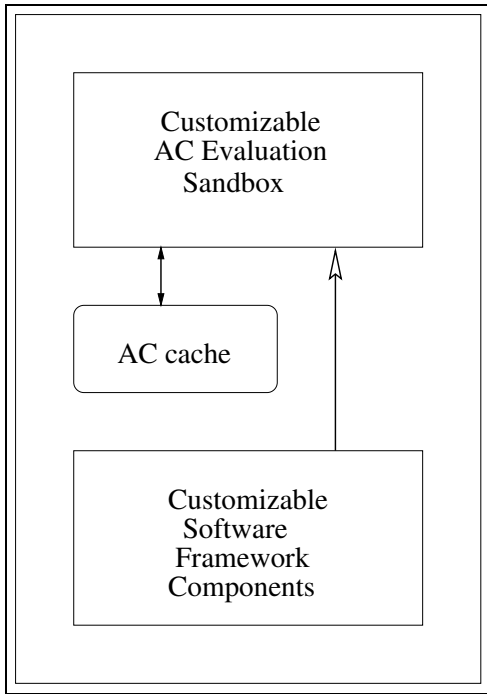


Figure 3: Evaluation and Enforcement engine

or default mechanisms and policies. Different AC types require different contexts. By providing the run-time resolvable references, we can download the software components that form the context from a trusted repository. A sandbox is a restricted execution environment and imposes static and dynamic constraints on the code that runs inside it. A typical example of a sandbox is the Java applet execution environment. This sandbox prevents arbitrary mobile Java bytecode from accessing most of the local files, sensitive data and critical applications.

The sandbox required for the administrator can also customize the evaluation of the ACs or trusted applications. The entire evaluation/enforcement engine needs to be secured in some way. It can run as a process with superuser privilege and create a cryptographically secured channel to communicate with the AC management infrastructure. This channel can be used to obtain the ACs and the downloadable framework and sandbox components. Alternatively it can also be a part of the kernel of a traditional or extensible operating system. The enforcement is done after evaluation. The evaluation/enforcement engine can subsume the concept of a traditional reference monitor. Typically when the application makes a call that accesses a specific resource or requires the use of a specific mechanism, the request is encapsulated and passed to the evaluation engine. The engine builds the context and evaluates the AC

associated with the policy or mechanism requested by the application. Depending on the result of this evaluation, the application is either granted or denied the access to the resource or allowed to use the mechanism it requested. To support the enforcement, this engine must export an interface that allows or forces applications to redirect their request to resources and mechanisms through itself. In addition, the engine must either notify applications that the access is denied or forward allowed requests to the appropriate resource, thereby implementing the policy specified in the AC.

2.4 AC Management

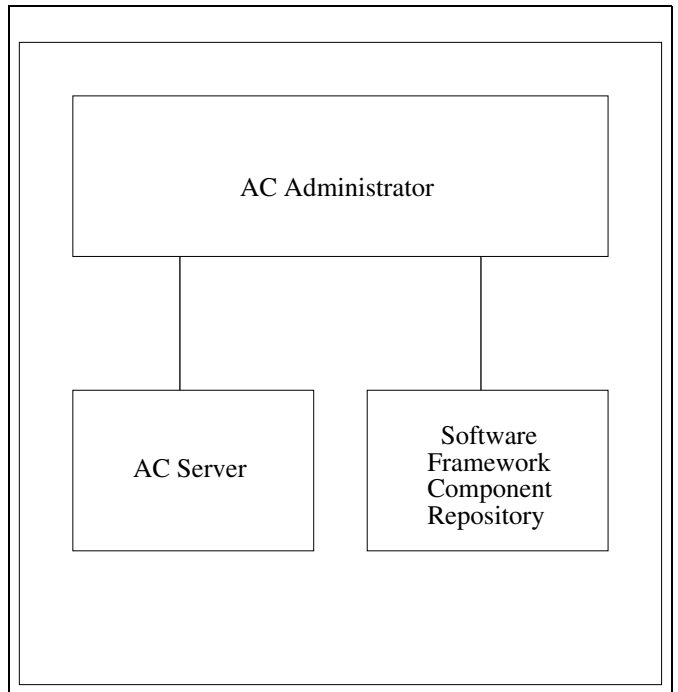


Figure 4: AC management infrastructure

The AC Administrator is equivalent to a trusted third party as in a traditional security model. It is responsible for validating and attesting to the integrity of the active capabilities. For example, using a public key infrastructure(PKI), the AC administrator can sign ACs using its private key, and other entities, like applications and evaluation/enforcement engines, can perform verification using the public key of the AC Administrator. Typically it is also responsible for the creation of the capabilities using the interface provided by the software framework. Although ACs can carry arbitrary code, the creation interface provided by the framework can restrict the capabilities to well-formed expressions and can perform static type checking and

verification to make sure that the code in the AC cannot compromise the security of the underlying system. The run-time behavior of ACs is restricted by the sandbox, which limits their access rights and resource utilization. In addition, the communication channel between the administrator and the evaluation/enforcement engine needs to be secure. The AC administrator itself may be implemented as one centralized entity, or its functionality can be distributed throughout the protection domain. There may be multiple instances of the AC administrator to achieve load balancing, scalability and fault tolerance. The AC server acts as a front-end to an AC Administrator’s AC repository. This server may be a part of the administrator or may be another entity, closely coupled with the functionality of the AC Administrator.

3 Applications

In this section we describe some application scenarios that highlight the benefits of using our system. One example we have chosen uses ACs to provide dynamic countermeasures against intrusions. In a typical system, when an intrusion or possible intrusion is detected, a security alert is issued. This security alert needs to be issued promptly at runtime without interrupting normal service. This may result in modifying the security policy at the compromised node and its neighbors, activating additional security measures, imposing additional auditing schemes and more restrictive security policies.

Consider the network configuration shown in Figure 5. In this figure, one or more compromised nodes are isolated from the rest of the network and a dynamic firewall is built around them. All nodes that are directly connected to the compromised node are sent ACs to change their existing access control and security mechanisms to minimize the risk of compromising the rest of the nodes in the network.

The traditional way of dealing with intrusion relies on detecting patterns of abnormal or suspicious behavior. Using pattern matching and analysis, or data mining etc., intrusion detection systems define a fixed set of countermeasures that attempt to minimize the damage. One drawback of this approach is that it is very hard to prepare in advance the countermeasures for every kind of attack. Since there is no way to prepare for every kind of intrusion, the fixed set of countermeasures and policies for the firewall nodes needs to be updated very frequently. In

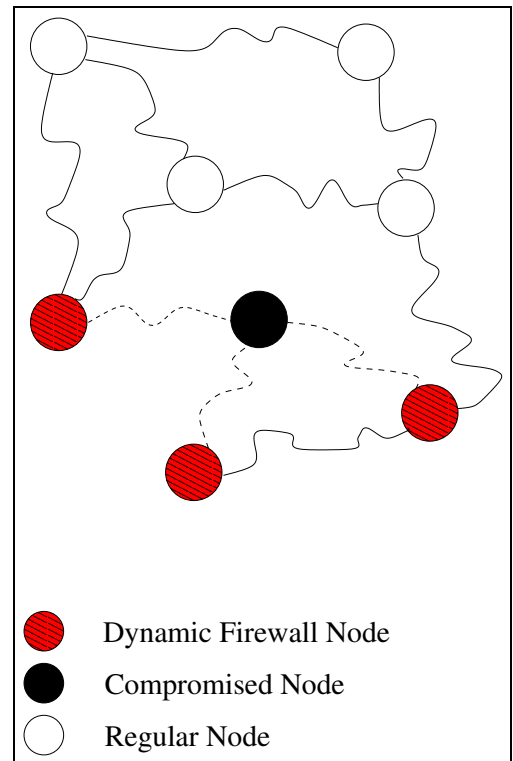


Figure 5: Dynamic Countermeasures for Intrusion

existing systems the overhead associated with this operation is substantial. Using our low-overhead system we can define dynamic policies customized to a particular attack or possible attack, which can be triggered and enforced by suspicious behavior. A static policy that denies all accesses is too restrictive. In many cases it may be worthwhile to adopt customized countermeasures and allow the services that are not compromised to continue. If we detect an intrusion that requires more drastic countermeasures, we can change the security level of the whole system by installing restrictive ACs on each node, for example by changing the current policies to MAC or by disabling a particular service or application.

Active capabilities can also be used to perform distributed computation. The results of this computation can be used to enforce situational policies. Using a weighted trust model, described in [12], applications can assign weights to different entities that reflect upon their “level” of trust in the system. The AC can evaluate the trust level for a specific entity and enforce the policy specific to that trust level. Thus the administrators may dynamically force applications to change their policies and mechanisms based on changing trust levels and force applications to adapt protective measures against nodes on less trustworthy paths.

Another example is a mobile computing environ-

ment where users wearing a network of mobile devices enter a smart room that has its own independent security mechanisms and policies. In order to use the services available in that room, these users have to adapt their network’s applications and policies and use the underlying security infrastructure. The need to integrate seamlessly the mobile system with the smart room security infrastructure implies that the mobile users must adapt to the security policies of the smart room and vice versa. The administrators use ACs to customize the user, device, network or smart room security profile to enforce these policies at run time.

4 Implementation

Based on the location of the evaluation/enforcement engines we have identified three specific implementation models of our architecture. In the first model, the evaluation/enforcement engines are implemented in the kernel of the system. The engines reside and execute in protected address space. In the second model, the engines reside in the software framework itself. For example, the engine can be built in a JVM(Java Virtual Machine or Java runtime environment) as a sandbox and can manage access to Java objects and Java security policies and mechanisms. This model relies heavily on the security and safety features of the software framework itself. In the third model the engines reside and execute in separate, independent user space, and the system protects them from unauthorized accesses and is responsible for enforcing the behavior of these engines.

In this section we describe two specific implementations of the above three models. The focus of the Cherubim¹ project [2] is to provide dynamic security support in CORBA for mobile computing that allows frequent migration of computers in and out of security enclaves and that facilitates wide area collaboration by creating dynamic sessions that stretch across organizational boundaries. In Cherubim, the ORB is in user space and is assumed to be trusted. This implementation belongs to the third model. The Seraphim² project [3] defines a dynamic, interoperable, flexible, composable security architecture for active networks. We implemented a reference monitor in the NodeOS to control NodeOS accesses.

¹The Cherubim software release is available at <http://choices.cs.uiuc.edu/Security/cherubim/software/>

²Contact authors for the Seraphim software release

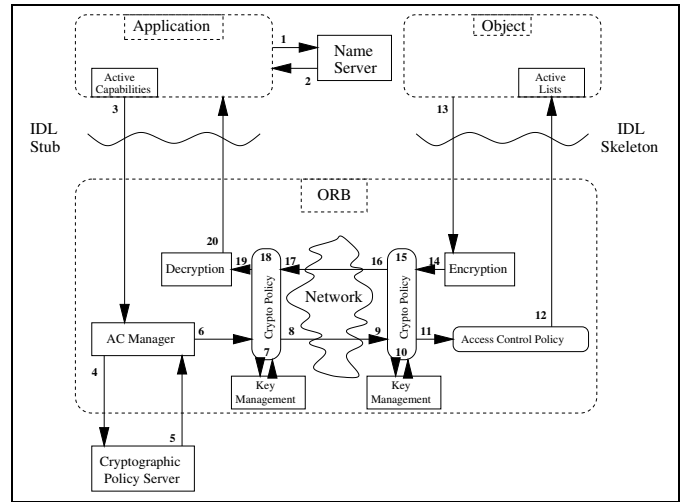


Figure 6: Object Access in *Cherubim*

This implementation belongs to the first model.

4.1 Cherubim

The design of the Cherubim security architecture [11] consists of two major parts: CORBA compliant security services with a security enhanced IDL (Interface Definition Language) providing application level security interfaces, and an agent based dynamic security framework implementing these standard interfaces. Adopting OMG’s general security reference model and Security Service Interfaces gave us an open architecture to incorporate a wide variety of security policies and services. In addition, Cherubim aims to provide fast technology evolution and deployment for both user applications and security functions. Using the standard CORBA security services and a security extended OMG IDL provides the necessary basic facilities to achieve this and easily separates security functions from the main application functions. In general, this design provides better extensibility and configurability for both applications and their security features. It also enables wide area software integration with configurable security enforcement. Figure 6 briefly illustrates the process of secure object invocation in Cherubim integrated with CORBA security services. The numbers in the figure show the sequential processing steps for a client request.

As in the OMG model, Cherubim defines a principal as a human user or system entity that is registered in and authenticated to the system. It may have one or several identities, which we call roles in Cherubim. Each role may have different privilege attributes used in making access control deci-

sions. All the information about roles and privilege attributes of a principal is maintained in a secure store object that is called a credential. In Cherubim, credentials may be embedded in the active capabilities to support delegation and to facilitate efficient access control. Objects, policies, and services are organized into security domains, each of which defines a distinct scope with a common set of security policies and mechanisms. In each domain there is a security authority, which we implemented as a policy server with administrative interfaces to security policies in Cherubim. A domain may also be divided into sub-domains to form a security domain hierarchy. To inter-operate between security domains, inter-domain security policies need to be defined and enforced. In addition Cherubim has developed a policy representation framework with role extensions to provide interoperability with RBAC policies while allowing extensibility. One of the major drawbacks of this approach is that OMG CORBA security services are implemented and enforced in user space. It is possible to make stronger guarantees if an ORB can be securely protected.

4.2 Seraphim

Active networks [17] aim to provide a software framework that enables network applications to customize the processing of their data. Applications encapsulate the methods that manipulate the data, with or without the data itself, and inject these capsules into the network. Active routers install and execute these capsules on the data dynamically, thereby facilitating fast protocol and service deployment. Securing this infrastructure against threats and exposures remains a major challenge in this paradigm.

In order to exploit fully the expressive power of the underlying active network, we felt the need for a unified security framework that allows users or applications to create and enforce their own security mechanisms, similar to customizing their own communication protocols. Seraphim [3] is a prototype of a dynamic, fully extensible, inter-operable, security architecture, based on, and built into the underlying active network architecture. This architecture allows active network routers to be configured with only a minimal set of security functions. These functions are recursively used to install and support the secure deployment of new security mechanisms. For instance, sophisticated and application or user specific security functions may be installed at run time using

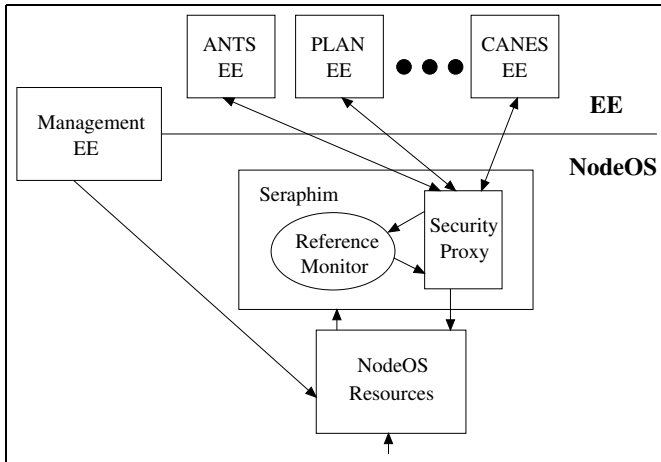


Figure 7: Secure Active Network Node

a secure recursive reconfigurable bootstrap process. Currently, our framework provides mechanisms to dynamically specify, separate and enforce a number of different, often mutually exclusive access control policies. In addition we allow applications to encapsulate credentials and to encode situational policies to alter or revoke dynamically existing access control rules and mechanisms. Applications construct capsules, which use this software environment as context and inject customized security policies into the routers.

The major components of our Seraphim architecture and their interaction, in the context of the active network architecture are shown in Figure 7.

The key component of this architecture is the reference monitor. This is similar to the evaluation/enforcement engine in the general architecture. Currently the reference monitor is implemented as a co-located extension to the NodeOS. Every node has a reference monitor through which all accesses to node resources occur. The policy framework is the software framework component. As mentioned earlier, the policy framework itself is reconfigurable and can be downloaded dynamically when required. Applications or administrators use the interface provided by this policy framework to create ACs that encode the type of access control policy and other constraints used in the access control decision making process.

5 Performance

In this section we describe some experiments we implemented in our Seraphim system. Although our prototype implementation was not built for performance, we did make some preliminary performance

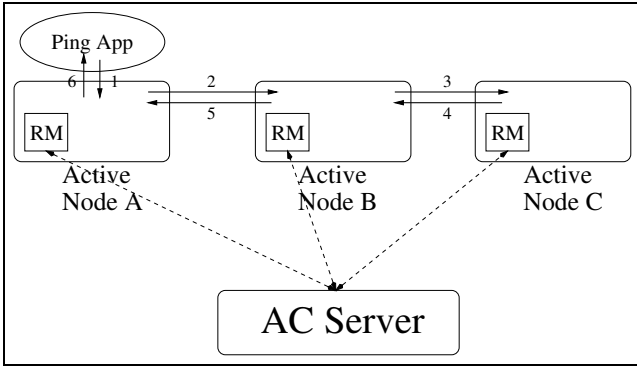


Figure 8: Ping Application Experiment in *Seraphim*

measurements. We also present a discussion on the overheads incurred in our system. In order to write applications for our Seraphim system, we developed SAINTS (Secure Active Inter-operable Network Toolkit System) which is based on the ANTS toolkit [16]. The original ANTS `Node` class was split into distinct `NodeOS` and `EE` classes and we added our reference monitor between the `EE` and the `NodeOS`. Our SAINTS is backwards compatible with ANTS and can run original ANTS applications. The next few subsections summarize the performance results for Ping, Gnipper and dynamic policy change applications. This is followed by a subsection that discusses strategies for reducing some of the overheads.

5.1 Ping Application

The first experiment measures the performance overhead associated with the modified version of the ANTS Ping application. The experimental setup is shown in Figure 8. The numbers in the figure show the sequence of steps in the transmission of an ANTS Ping capsule. The communication between the AC server and the active node is through TCP. We used three Sun SparcStation 10 machines for the active nodes and a Sun Ultra-60 machine for the AC server. All four machines are on the same 100Mbps Ethernet LAN.

We used four different system configurations to measure the performance. Our measurements exclude the time to load dynamically the active Ping protocol in active nodes. The first configuration, “No RM”, is the baseline. In this configuration the Seraphim reference monitor was installed in the NodeOS of all the active nodes, but was bypassed and no access checks were performed. The second configuration, “RM without Cache”, was the most straightforward configuration and used the

System Configuration	Ave. RTT (ms)
No RM	10
RM without Cache	1494
RM with Cache	21
RM with Decisions in Cache	10

Table 1: Performance Data for Ping Application

Seraphim reference monitor. In this configuration a reference monitor was installed in all the active nodes, and every time a Ping capsule arrived, one access check was performed. There was no cache in the reference monitor. The reference monitor at each node had to contact the AC server to retrieve the proper AC and to evaluate it for each access check. The third configuration, “RM with Cache”, was an improvement over the second configuration. Again, the reference monitor performed one access check for each arriving capsule, but in this case the proper AC was cached inside the reference monitor. Here the reference monitor did not need to contact the AC server at all. The final configuration, “RM with Decisions in Cache”, caches the reference monitors evaluation result of the AC. When the AC does not change and the inputs to the AC are the same as to the previous evaluation, a prior cached evaluation value can be used. Each access check in this configuration involved simply a cache lookup, without the dynamic evaluation of the AC.

In all configurations, 2000 capsules were sent out from the Ping application and the interval between the Ping capsules was 200ms. We measured the average round trip time (RTT) of the Ping capsules. Table 1 shows the results of our experiment.

We observed that the most inefficient implementation of our architecture is the “RM without Cache” configuration. It has much larger overhead than the base case. When the simple caching scheme was used, the average RTT was approximately twice the base RTT value. When we further assumed that the result of AC evaluation can be cached, average RTT was same as in the base case. This result suggests that by employing suitable optimization techniques, the overhead of our Seraphim architecture can be reduced to an acceptable level. The overhead of checking that the inputs to the AC had not changed and the Ping capsule certificate was the same as that of a previous capsule was negligible. Further research is required on different caching schemes and optimization strategies.

5.2 Gnipper Application

Another experiment we developed in Seraphim is the Gnipper application. This application demonstrates the creation of dynamic protection domains or enclaves. In this application, we create Gnipper vaccine, or an anti-Ping AC. This vaccine is used to disable one user’s ability to ping an active node. This vaccine is installed at the reference monitor of the active node that needs the protection. The vaccine dynamically moves one-hop at a time toward the source of the Ping, in response to Ping requests from the original sender. This is best explained with an example presented next.

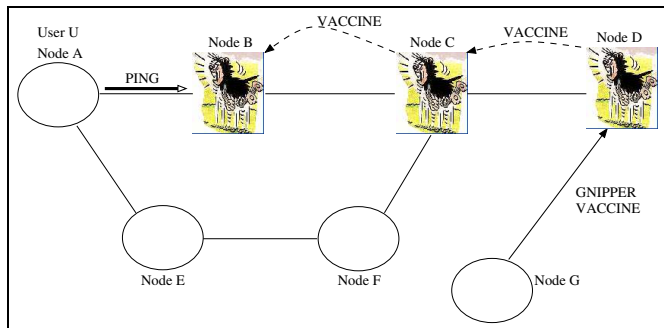


Figure 9: Gnipper Application

The example is shown in Figure 9. We create a vaccine and install it at the reference monitor of Node D. The vaccine is an AC which disables the ability of user U at Node A to ping Node D. User U sends a Ping capsule with destination address of Node D. When the capsule arrives at Node D, Node D drops the capsule and propagates the vaccine to the previous node, Node C. Now Node C is vaccinated. When user U sends another Ping capsule to Node D, the capsule will be dropped at Node C and the vaccine will be installed at Node B.

The exact node which drops the Ping capsule changes dynamically depending on the the number of Ping capsules sent by the source. By installing successively the vaccine at the nodes on the Ping path between Node A and Node D, we have succeeded in building a dynamically growing firewall around D, and also reduced the traffic and moved any denial of service attack away from the intended victim.

This experiment used the same setup as for the Ping application. The average overhead for an application running on a Sun SparcStation 10 machine to create a vaccine and install it at another Sun SparcStation 10 machine on the same local Ethernet LAN was measured as 77ms. Without creation, the aver-

age time to send a vaccine and install it under the same setup took 34ms.

This experiment can be extended to build agile and dynamic firewalls that can react to attacks at runtime. When an active node or trusted agent detects attempted attacks, it can send out an active capability carrying a “warning” message with the appropriate vaccine, to build a dynamic line of defense against outside attacks, or to raise the level of security within the domain. When the threat is gone the active node or the trusted agent can send out another active capability to resume normal operation. This can be used as a very powerful security tool in conjunction with intrusion detection and counter-measure systems.

5.3 Dynamic Policy Change

Our policy framework can dynamically change between policy types, for example, from RBAC to MAC. Section 3 mentions an application that can benefit from such a change. We measured the overhead associated with this change using our existing system. For a simple domain with 5 users, 10 objects and 5 operations, changing from RBAC with 5 roles to a MAC with 2 security levels cost about 667ms on an average using a Sun Ultra-60 machine. We have identified several places for optimization and further research is planned to reduce this overhead considerably.

5.4 Secure Multicast Application

A secure multicast application for active networks was implemented in Seraphim. Traditional secure multicast uses session keys to enforce secure data transmission and delivery. When a new member joins a secure multicast group, the sender sends the current session key to the new member so that it can receive data. When a member leaves the group, the session key has to be changed, to prevent eavesdropping. The sender has to generate a new session key and transmit it to all the remaining members.

In our secure multicast example, joining and leaving are both very simple. When a new member joins the group, a new AC is created for this member. The enforcement engine for this new member will get the new AC only when it needs this AC. When a member leaves the group, the AC administrator sends a simple revocation AC to the enforcement engine for this leaving member. The overhead is much smaller than traditional secure multicast systems.

5.5 Discussion

In traditional systems, access control is defined by policy and is enforced by enforcement engines such as reference monitors in operating systems and firewalls in networks. Individual policies can be defined at each enforcement point and managed separately. For example, each firewall in a company can be configured individually to set up a set of rules defined by policy. When there is a policy change, a human administrator can identify the affected firewalls, suspend and reconfigure them to enforce the new policy. If there are many firewalls and frequent policy changes, then this approach is not scalable.

An alternative approach is to have a centralized policy administrator. The policy administrator maintains all policy information for the whole system, and is responsible for distributing policy to individual enforcement engines. When there is a policy update, the policy administrator can distribute the new policy to the enforcement engines. Usually the policy administrator does not know in advance where the new policy is going to be used, so the administrator may have to distribute the new policy throughout the whole system. For example, when a new user is added into the system, the access privileges for this new user needs to be distributed to all enforcement engines. Also when there is a revocation of an existing policy, the revocation information needs to be distributed to all the enforcement engines. In some environments, such as the Cisco Secure Policy Manager for firewalls [5], enforcement engines are suspended in order to update policies. In traditional systems, policy update is often complicated and the overhead is large.

In our architecture, we also have a centralized AC administrator per domain. However this entity may be replicated for fault tolerance and load balancing. The AC administrator keeps track of the location of ACs in the system. When there is a policy change, this causes the ACs to change and the old ACs have to be revoked. The AC administrator sends changed ACs to only those enforcement engines that have the old ACs. When there is a policy change which causes addition of new ACs, the AC administrator does not need to send out anything. When the enforcement engine receives access requests from applications for the first time, it asks for these new ACs from the AC administrator and caches them. Therefore the overhead is for first time use only, and there is no system-wide distribution overhead. An overhead introduced by our approach is the need to maintain information

in an AC server about the location of the ACs. Since the centralized computation time usually is smaller than any network delays, the maintenance overhead is much smaller than the extra network overhead of a traditional system.

Sometimes it is difficult for the AC server to keep track of ACs for revocation. In such cases active applications can get ACs from the AC server, and distribute ACs by themselves. For example, in our Seraphim implementation, applications can encode ACs into active capsules, and distribute them through active networks along with active capsules. Applications do not need to know the details of the policy framework and the AC contents. In this way the system can provide very fine-grained customized policies for applications. One simple way to accomplish revocation is to use a time-out. Another solution is to broadcast the revocation information, as in traditional systems. In either case, the overhead is no larger than traditional systems.

6 Conclusions

In this paper we propose an agent based security architecture that allows applications to create customized and situational policies. The motivation for this architecture is that different applications (users or devices) tend to have widely different requirements in terms of security policies and mechanisms. The traditional model of security is very static and cannot support different mechanisms and policies, or change between these policies and mechanisms dynamically.

In our system, applications use the expressive power of our software framework and the flexible nature of our infrastructure to create mobile agents called active capabilities (ACs). These ACs actually carry the customized policy to evaluation/enforcement engines where they are evaluated in a sandbox-like environment and the result of this evaluation is enforced on the applications. The AC management infrastructure defines a trust model for the interaction of various components and manages the creation and distribution of ACs.

The security framework and the evaluation and enforcement engines are composable and extensible and require only a minimal set of functions and mechanisms to be installed. Additional mechanisms and policy implementation components can be downloaded dynamically using a secure communication channel. The overhead of managing policies

and mechanisms is very low as this can be handled by the ACs. AC simplifies key management and distribution, making our approach very scalable.

References

- [1] Roy H. Campbell, M. Mickunas, Tin Qian, and Zhaoyu Liu. An agent-based architecture for supporting application aware security. In *the Workshop on Research Directions for the Next Generation Internet*, May 1997.
- [2] Roy H. Campbell and M. Dennis Mickunas. An agent-based architecture for supporting application aware security. an accepted proposal to DARPA BAA9704, 1997. Also see the web site at <http://choices.cs.uiuc.edu/Security/cherubim/>.
- [3] Roy H. Campbell and M. Dennis Mickunas. Building dynamic interoperable security architecture for active networks. an accepted proposal to DARPA BAA9803, 1998. Also see the web site at <http://choices.cs.uiuc.edu/Security/seraphim/>.
- [4] Roy H. Campbell and Tin Qian. Dynamic agent-based security architecture for mobile computers. In *the Second International Conference on Parallel and Distributed Computing and Networks*, Brisbane, Australia, December 1998.
- [5] Cisco Systems, San Jose, CA. *Cisco security manager tutorial, DOC-786905*, 1999. Available at <http://www.cisco.com/warp/public/cc/cisco/mkt/security/csm>.
- [6] D. Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Company, 1982.
- [7] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 9-12 1999.
- [8] Tim Fraser. An object-oriented framework for security policy representation. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1996.
- [9] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *PLDI '96*.
- [10] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *OSDI '96*.
- [11] Tin Qian. *Cherubim agent based dynamic security architecture*. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1998.
- [12] Tin Qian. *Dynamic authorization support in large distributed systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1999.
- [13] Vijay Raghavan. On the design and implementation of a security policy administration for a dynamic security system. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999.
- [14] R. S. Sandhu and E. J. Coyne. Role-based access control models. *IEEE Computer*, 29(2), February 1996.
- [15] Dan S. Wallach. *A new Approach to Mobile Code Security*. PhD thesis, Department of Computer Science, Princeton University, January 1999.
- [16] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: a toolkit for building and dynamically deploying network protocols. In *IEEE OPE-NARCH'98*, San Francisco, CA, April 1998.
- [17] D. Wetherall, U. Legedza, and J. Guttag. Introducing new internet services: Why and how. *IEEE Network Magazine*, July/August 1998.