

AFRL-IF-RS-TR-2002-262
Final Technical Report
October 2002



BUILDING A DYNAMIC INTEROPERABLE SECURITY ARCHITECTURE FOR ACTIVE NETWORKS

University of Illinois

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. G378

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-262 has been reviewed and is approved for publication

APPROVED:



SCOTT S. SHYNE
Project Engineer

FOR THE DIRECTOR:



WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE October 2002	3. REPORT TYPE AND DATES COVERED Final May 98 – Jun 02	
4. TITLE AND SUBTITLE BUILDING A DYNAMIC INTEROPERABLE SECURITY ARCHITECTURE FOR ACTIVE NETWORKS			5. FUNDING NUMBERS C - F30602-98-1-0192 PE - 62301E PR - G378 TA - 00 WU - 01	
6. AUTHOR(S) Roy H. Campbell and M. Dennis Mickunas				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Illinois Grants and Contracts Office 109 Coble Hall – 801 South Wright Street Champaign Illinois 61820-6242			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFGA 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2002-262	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Scott S. Shyne/IFGA/(315) 330-4819/ Scott.Shyne@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) Security is viewed as one of the major obstacles to the widespread deployment active networks. A significant challenge is to develop mechanisms to change software state on routers dynamically, without sacrificing protection guarantees. The Seraphim projects leverages the inherent dynamism in the paradigm to build dynamic security mechanisms for active networks. Seraphim's security architecture is component based, dynamically extensible, and reflective, and supports a variety of policy strategies and enforcement mechanisms. This enabled the development of customizable, interoperable, domain-specific, or task-specific security policies and mechanisms, to meet the security requirements of active network entities. Administrators were able to develop security policies as active network capsules, called dynamic policies, and enforce these policies by executing them in a suitable software context on active network routers. A suite of confidentiality, integrity, authentication and access-control mechanisms was developed to secure the node of an active network. This suite was based on standardized APIs and provided support for customized Quality of Protection guarantees. Customized dynamic policies were created and installed at run-time, trading functionality for performance, to implement low-overhead solutions that were able to successfully counter threats and attack, without sacrificing protection guarantees.				
14. SUBJECT TERMS Active Networks, Security Mechanisms, Security Policies, Access-Control Mechanisms			15. NUMBER OF PAGES 90	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

1. SUMMARY.....	1
2. OVERVIEW OF CONTRIBUTIONS	2
1.1 SECURITY ARCHITECTURE FOR DYNAMIC POLICY	2
1.2 DYNAMIC ACCESS CONTROL POLICIES	3
1.3 SECURE NODE ARCHITECTURE AND SECURITY AS SERVICES.....	3
1.4 SECURE FLOW ANALYSIS.....	3
1.5 FORMAL SPECIFICATION AND VALIDATION OF DYNAMIC POLICIES	4
1.6 DENIAL OF SERVICE PROTECTION	4
3. LIST OF ACCOMPLISHMENTS.....	5
4. REFERENCES	8
APPENDICES	10
APPENDIX A AN AGENT BASED ARCHITECTURE FOR SUPPORTING APPLICATION LEVEL SECURITY.....	11
APPENDIX B SERAPHIM: DYNAMIC INTEROPERABLE SECURITY ARCHITECTURE FOR ACTIVE NETWORKS	23
APPENDIX C FLEXIBLE SECURE MULTICASTING IN ACTIVE NETWORKS	33
APPENDIX D SECURE INFORMATION FLOW IN MOBILE BOOTSTRAPPING PROCESS	41
APPENDIX E DYNAMIC, DISTRIBUTED, SECURE MULTICAST IN ACTIVE NETWORKS	49
APPENDIX F SECURING THE NODE OF AN ACTIVE NETWORK.....	55
APPENDIX G PLUGGABLE ACTIVE SECURITY FOR ACTIVE NETWORKS	69
APPENDIX H DEVELOPING DYNAMIC SECURITY POLICIES	75

1. SUMMARY

In an active network, new protocols and services can be injected into the network using *smart packets* to carry customized software components. This technology increases the degree and sophistication of the network architecture and enables fast deployment of new protocols and services. However, allowing installation of arbitrary software components on routers may cause undesirable side effects and impact the protection guarantees of the software on the routers. Administrators of active network routers may want to restrict the behavior of active capsules and preserve certain behavior guarantees at all times. These guarantees may be specified as safety properties of component mechanisms, noninterference properties of information flows, or timing guarantees in availability policies. Traditional security mechanisms and network policy management tools have limited support for changing and enforcing different types of policy strategies, let alone policies, at run-time. The ability to specify, implement and enforce these policies in a dynamic environment becomes crucial.

In Seraphim, we study the interoperability, extensibility, and configuration issues of security policies for active networks. To address these issues, we introduce the notion of dynamic policies that can be enforced by executing them on an active network node. These policies are designed by formally modeling the behavior and interactions between different components on active routers. Behavioral guarantees, expressed as temporal safety properties, form an integral part of the specification of dynamic policies and can be validated within the model framework. Policies are implemented by wrapping the mechanisms to change operational parameters with suitable guards so as to preserve these behavior guarantees. This combination of guards and commands are encapsulated in active capsules and the policy they specify is enforced by instantiating and executing these capsules in a suitable sandbox-like environment on the active router. Using our policy framework, we can change policy strategies (e.g., between MAC and RBAC) at run-time, in response to intrusions and other security violations, without sacrificing security guarantees.

We also provide a suite of customizable security mechanisms to protect the integrity, authenticity, and confidentiality of capsules exchanged between active routers. These mechanisms to secure the node of an active network are implemented as services based on standardized APIs. The services are carefully designed and analyzed to preserve noninterference properties and prevent information leaks. The dynamism afforded by the architecture also allows us to implement different Quality of Protection (QoP) levels to provide customizable security for active flows. This enables us to implement minimal security policies and deploy stronger mechanisms on a need to protect basis, and amortize performance penalties. We believe that this support for dynamism in security is our major contribution in the context of security for active networks.

2. OVERVIEW OF CONTRIBUTIONS

We believe that no single security architecture will be able to address the security issues for active networks in general. Customizable security policies and extensible security mechanisms will play an important part in addressing the security concerns with the practical deployment of active network infrastructure on routers. We argue that developing satisfactory security solutions in a dynamic environment requires support for dynamic security. We provide a framework to develop reactive security solutions in this context, on a need-to-protect basis with minimal overhead in terms of software and performance. In this section, we summarize our major contributions, which include:

- A dynamic security architecture for active networks that provides support for dynamic policies that integrates seamlessly with proposed active network architecture [Liu00-1, Liu00-2], along with a distributed secure multicast application to demonstrate these policies [Var99, Var00, Liu00-3].
- A componentized policy framework that implements different access control strategies and allows administrators to change between policy strategies at run time [Nal00].
- A suite of customizable mechanisms to secure the node of an active network along with a flow analysis to guarantee noninterference [Liu005, Liu00-6].
- Formal specification and validation of dynamic policies [Nal02].
- Investigation of denial of service prevention and implementation of certified bandwidth mechanisms.

We describe each of these contributions in greater detail the following subsections.

1.1 SECURITY ARCHITECTURE FOR DYNAMIC POLICY

The Seraphim project developed dynamic and fully extensible security architecture for active networks [Liu00-1, Liu00-2]. The architecture is based on the principles underlying active networks rather than on existing static systems. Seraphim project adopts ideas and technologies from previous Cherubim mobile agent based security architecture research, including dynamic security policies that support interoperability among different security domains, and active capabilities that provide application specific security functions. In addition, Seraphim's security architecture for active networks imposes only a minimal set of security functions on the base active network architecture to support secure deployment of new security services. More sophisticated and application specific security functions may be recursively installed using a secure reconfigurable [Liu00-4] bootstrap process. Seraphim's architecture is not constrained to one specific security scheme for securing smart packets and active nodes. This reflective design allows the maximum flexibility for building a secure active network environment. Seraphim's security architecture fits transparently into the proposed Active Network and Active Network Security Architecture. We have also integrated it with the ABone test-bed.

1.2 DYNAMIC ACCESS CONTROL POLICIES

Seraphim's Dynamic Policy Management Framework, written in Java, implements different access control strategies. In addition to DAC (Discretionary Access Control), MAC (Mandatory Access Control) and RBAC (Role Based Access Control) we have also incorporated the I-RBAC (Interoperable RBAC) [Kap00] and R^2 BAC models developed by our group. The I-RBAC model allows us to interoperate between different RBAC domains, providing us a dynamic mechanism to translate our dynamic access control policies across different domains. The R^2 BAC model allows us to change between two different RBAC instances in the same domain. This model is useful to deploy a restrictive access control policy under an attack and change it back to the default when the threat has receded.

1.3 SECURE NODE ARCHITECTURE AND SECURITY AS SERVICES

The secure node architecture includes an active node operating system security API, an active security guardian, and quality of protection (QoP) provisions [Liu00-5, Liu01, Liu02]. The architecture supports highly customized and situational policies created by users and applications dynamically. It permits active nodes to satisfy application-specific dynamic security and protection requirements. The secure node architecture can provide a fundamental base for securing the active network infrastructure. It provides a framework that adapts and implements the Pluggable Authentication Module API, Generic Access and Authorization API and Generic Security Services API for authentication, authorization, and various security services. The implementation uses DES, IDEA, and Rijndael encryption algorithms, whose keys are exchanged through RSA/X.509v3 algorithm, for dynamic customized security services. The security configuration supports various encryption algorithms and RSA key lengths. Applications can dynamically select the suitable security configuration and services at each routing hop, based on their security and performance requirements.

1.4 SECURE FLOW ANALYSIS

In addition to the secure node architecture, we also provide the analysis of secure information flow using a type system [Liu01]. Information flow control is concerned with the right of dissemination of information. Secure information flow properly restricts the propagation of sensitive cryptographic data beyond the security API to untrusted environments. The analysis demonstrates that the type system can ensure secure flow enforcement efficiently and therefore provide additional security assurance for active networks. The type system guarantees that a well-typed program satisfies the noninterference security property. This means that the program does not leak sensitive data.

1.5 FORMAL SPECIFICATION AND VALIDATION OF DYNAMIC POLICIES

We introduce formal modeling and specification in our policy development life cycle. In most existing systems, policies are implemented and enforced by changing the operational parameters of shared system objects. These policies do not account the behavior of the entire system, and enforcing these policies can have unexpected interactive or concurrent behavior. We develop a policy specification, implementation, and enforcement methodology based on formal models of interactive behavior and satisfiability of system properties. We show that by carefully designing the code to change the operational parameters our policy implementation entities, dynamically installing and executing our policies does not affect the behavioral guarantees specified by the properties. Our dynamic policy is a program consisting of a set of guards and actions, created by our policy administrator. It encodes not only the logic to modify the system implementation to change operational parameters, but also includes all the necessary guards to enforce good behavior and prevent its misuse. For example, in the access control policy example, the guard can include proofs of authorization, and the commands are programs to change parameters of an access control rule. In our Seraphim active network prototype, these programs map directly to active capsules, and can be viewed as in-line policies. We also describe other types of dynamic policies for information flow and availability, based on safety, liveness, fairness, and other properties. We believe that dynamic policies are important building blocks of reactive security solutions for active networks.

1.6 DENIAL OF SERVICE PROTECTION

In addition to our work in dynamic policies, we have also developed a behavioral model of network denial of service; especially Distributed Denial of Service attacks (DDOS). Based on the behavior analysis, we argue that the trace of a DDOS victim's behavior cannot be made DDOS resistant by implementing a suitable mechanism on the victim alone. Bandwidth agreements, similar in flavor to user agreements, are necessary to prevent denial of service. We have implemented a lightweight mechanism that was demonstrated at the December 2000 Demo meeting, to attach bandwidth certificates to legitimate traffic. We call these certificates CABs or Credentials that Authorize Bandwidth. A CAB is a small, fixed length identifier that cannot be forged easily. It certifies that the packet it is attached to is legitimate. It can be used to mark legitimate UDP or control packets for DDOS resistance. One of the ways to ensure that valid CABs can only be created by trusted entities is to use cryptography and tie in a shared secret to the CAB value. Certified bandwidth can be used to implement cooperative bandwidth agreements required to prevent denial of service.

3. LIST OF ACCOMPLISHMENTS

June 1998 – June 1999:

1. Modified ANTS active network toolkit and built the SAINTS (Secure Active Interoperable Network Toolkit System) to implement the Active network architecture with explicit NodeOS and Execution Environment (EE) objects. This modified toolkit was the test-bed for most of our experiments.
2. Designed and implemented a lean security guardian to provide access control from the EE to the shared NodeOS resources. The security guardian is a colocated extension to the NodeOS. Every node has a security guardian, through which all accesses to node resources occur.
3. Completed implementation of the NodeOS proxy to support portability. The EEs direct their requests for NodeOS resources to the NodeOS proxy that sits atop the NodeOS. The proxy acts as a wrapper to the NodeOS API and redirects access control requests to the security guardian.
4. Completed implementation of Role Based Access Control (RBAC) within the Seraphim policy framework and integrated support for DAC and MAC from our previous project into the Seraphim toolkit.
5. Demonstrated support for secure, flexible, and dynamic multicast, as an extension of the original ANTS multicast scheme. Deposited a Master's thesis titled "Dynamic Distributed Secure Multicast in Active Networks".

July 1999 - June 2000:

1. Designed a NodeOS security API to support authentication, authorization and integrity. The API includes Pluggable Authentication Module (PAM) API, Generic Security Services (GSS) API, and Generic Authorization and Access Control (GAA) API. This security API is complement to the current NodeOS Interface Specification that focuses on fast network packet-forwarding fine-grained quality of service.
2. Developed an Active Caching framework for our active capabilities. By caching the reusable active capabilities, the system reduces the overhead of retrieving the active capability every time a security decision has to be made.
3. Modified and enhanced the *Security Architecture for Active Networks* document and circulated it throughout the active network community. The modifications and enhancements show the lessons learned from our Seraphim project and reflect the view of flexible, dynamic and interoperable active network security based on active capabilities.

4. Developed the IRBAC model of secure interoperability between security domains operating under the Role Based Access Control (RBAC) policy for dynamic role translations.
 5. Produced a PhD thesis on trust management in a distributed environment. The solution proposed in this model avoids the use of global name spaces and central trust authorities. The model enables fine-grained trust specification and flexible certificate management.
 6. Implemented the NodeOS security API, and integrated it into our SAINTS platform that uses active capabilities and security guardian for active security.
 7. Extended the NodeOS security API to support Quality of Protection (QoP) in active networks. The active applications can dynamically change the security and protection characteristics while traveling from hop to hop. Some examples of security and protection characteristics are different security algorithms, key sizes, and supports of security services.
 8. Provided more input for the Security Architecture for Active Networks document
- Implemented the IRBAC model in Seraphim architecture. With IRBAC, more than one autonomous domain can seamlessly interact with each with adequate security support.

July 2000 - June 2001:

1. Developed the BARMAN (Bandwidth Authorization and Resource Management in Active Network) protocol for the NodeOS. This protocol prevents denial of service attacks that “flood” networks with unwanted packets and block legitimate network traffic.
2. Deposited Master’s thesis titled “A Componentized Framework for Dynamic Security Policies”.
3. Participated in December 2000 AN PI Meeting with Team 4 integrated demo. Seraphim package was integrated with CANES platform and provided dynamic security support. In addition, BARMAN protocol was implemented inside the CANES Bowman NodeOS to provide safety against flooding DDOS attacks.
4. Developed the Reactive Role Based Access Control model (R^2 BAC). R^2 BAC is a way to use the IRBAC model as a defense mechanism against intrusions. We used R^2 BAC to efficiently change the role hierarchy inside a security domain to counter the possible security threats. We used INFOCON (Information Operation Condition) notion from DoD to model the current threats to the networks and used R^2 BAC to dynamically reconfigure the networks for tighter defense.
5. Developed a fuzzy logic based risk model. In this model, every threat is represented as a fuzzy set. The risk analyzer calculates the network-wide overall risk based on the fuzzy logic operation.
6. Participated in June 2001 AN PI meeting with a demonstration of R^2 BAC model and the fuzzy logic based risk assessment system. In this system, every threat to the network is characterized as a fuzzy set and the current risk to the whole network is calculated with

fuzzy logic. We also adopted the notion of DoD's INFOCON (Information Operation Condition) to show how our risk assessment system and R²BAC model can be used as an efficient defensive mechanism.

July 2001 – May 2002:

1. Deposited Ph. D Thesis on "Securing the Node of an Active Network. This thesis explores the security issues and develops a security architecture for NodeOS security by implementing security as standardized services adapted to the Active Network architecture. The thesis includes a validation of the design using secure flow analysis based on a type system to validate the noninterference properties of the composition of these security services.
2. Developed a formal model of dynamic policies and introduced the notion of property preserving policies. We also used formal validation and verification techniques to make strong safety guarantees about our dynamic access control policy framework.
3. Developing a formal Model of the Distributed Denial of Service problem.

4. REFERENCES

- [1] [Liu00-1] Zhaoyu Liu, Prasad Naldurg, Seung Yi, Tin Qian, Roy H. Campbell, and M. Dennis Mickunas, An Agent Based Architecture for Supporting Application Level Security. DARPA Information Survivability Conference and Exposition, Hilton Head Island, South Carolina, January 2000.
- [2] [Liu00-2] Roy H. Campbell, Zhaoyu Liu, M. Dennis Mickunas, Prasad Naldurg, and Seung Yi, Seraphim: Dynamic Interoperable Security Architecture for Active Networks. IEEE Third Conference on Open Architectures and Network Programming Proceedings (OPENARCH'2000), Tel Aviv, Israel, March 2000.
- [3] [Liu00-3] Zhaoyu Liu, Roy H. Campbell, Sudha K. Varadarajan, Prasad Naldurg, Seung Yi, and M. Dennis Mickunas, Flexible Secure Multicasting in Active Networks. International Workshop on Group Computation and Communications, Taipei, Taiwan, April 2000.
- [4] [Liu00-4] Zhaoyu Liu, M. Dennis Mickunas, and Roy H. Campbell, Secure Information Flow in Mobile Bootstrapping Process. International Workshop on Wireless Networks and Mobile Computing, Taipei, Taiwan, April 2000
- [5] [Var00] Sudha K. Varadarajan, Tin Qian, and Roy H. Campbell, Dynamic, Distributed, Secure Multicast in Active Networks. IEEE International Conference on Communication (ICC'2000), New Orleans, Louisiana, June 18-22, 2000.
- [6] [Kap00] I-RBAC 2000: Apu Kapadia, Jalal Al-Muhtadi, Roy H. Campbell, and M. Dennis Mickunas, Secure Interoperability Using Dynamic Role Translation. Proceedings of the 1st International Conference on Internet Computing (IC'2000), Las Vegas, Nevada, June 26, 2000.
- [7] [Liu00-5] Zhaoyu Liu, Roy H. Campbell, and M. Dennis Mickunas, Securing the Node of an Active Network Active Middleware Services. Kluwer Academic Publishers, Boston, Massachusetts, September 2000.
- [8] [Liu00-6] Zhaoyu Liu, Prasad Naldurg, Seung Yi, Roy H. Campbell, and M. Dennis Mickunas, Pluggable Active Security for Active Networks. 12th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'2000), Las Vegas, Nevada, November 6-9, 2000.
- [9] [Liu01] Zhaoyu Liu, Active Security for Active Networks, Ph. D Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2001.
- [10] [Nal02] Prasad Naldurg, R. H. Campbell and M. Dennis Mickunas, Developing Dynamic Security Policies, To Appear in the Proceedings of the 2002 DARPA Active Networks Conference and Exposition (DANCE 2002), San Francisco, CA, USA, IEEE Computer Society Press, May 29-31, 2002.
- [11] [Liu02] Zhaoyu Liu, Roy H. Campbell, and M. Dennis Mickunas, Security as Services in Active Networks. To Appear in IEEE International Symposium on Computers and Communication (ISCC 2002), Taormina, Italy, July 2002

- [12] [Qia00] Tin Qian, Dynamic Authorization Support in Large Distributed Systems, Ph. D Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2000.
- [13] [Var99] Sudha Varadarajan, Dynamic Distributed Secure Multicast in Active Networks. MS Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1999.
- [14] [Nal00] Prasad Naldurg, A Componentized Framework for Dynamic Security Policies, MS Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, August 2000.

APPENDICES

Papers attached to this report:

A Zhaoyu Liu, Prasad Naldurg, Seung Yi, Tin Qian, Roy H. Campbell, and M. Dennis Mickunas, An Agent Based Architecture for Supporting Application Level Security. DARPA Information Survivability Conference and Exposition, Hilton Head Island, South Carolina, January 2000.

B Roy H. Campbell, Zhaoyu Liu, M. Dennis Mickunas, Prasad Naldurg, and Seung Yi, Seraphim: Dynamic Interoperable Security Architecture for Active Networks. IEEE Third Conference on Open Architectures and Network Programming Proceedings (OPENARCH'2000), Tel Aviv, Israel, March 2000.

C Zhaoyu Liu, Roy H. Campbell, Sudha K. Varadarajan, Prasad Naldurg, Seung Yi, and M. Dennis Mickunas, Flexible Secure Multicasting in Active Networks. International Workshop on Group Computation and Communications, Taipei, Taiwan, April 2000.

D [Liu00-4] Zhaoyu Liu, M. Dennis Mickunas, and Roy H. Campbell, Secure Information Flow in Mobile Bootstrapping Process. International Workshop on Wireless Networks and Mobile Computing, Taipei, Taiwan, April 2000.

E Sudha K. Varadarajan, Tin Qian, and Roy H. Campbell, Dynamic, Distributed, Secure Multicast in Active Networks. IEEE International Conference on Communication (ICC'2000), New Orleans, Louisiana, June 18-22, 2000.

F [Liu00-5] Zhaoyu Liu, Roy H. Campbell, and M. Dennis Mickunas, Securing the Node of an Active Network, Active Middleware Services. Kluwer Academic Publishers, Boston, Massachusetts, September 2000.

G [Liu00-6] Zhaoyu Liu, Prasad Naldurg, Seung Yi, Roy H. Campbell, and M. Dennis Mickunas, Pluggable Active Security for Active Networks, 12th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'2000), Las Vegas, Nevada, November 6-9, 2000.

H [Nal02] Prasad Naldurg, R. H. Campbell, and M. Dennis Mickunas, Developing Dynamic Security Policies, To Appear in the Proceedings of the 2002 DARPA Active Networks Conference and Exposition (DANCE 2002), San Francisco, CA, USA, IEEE Computer Society Press, May 29-31, 2002.

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, tinq, 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

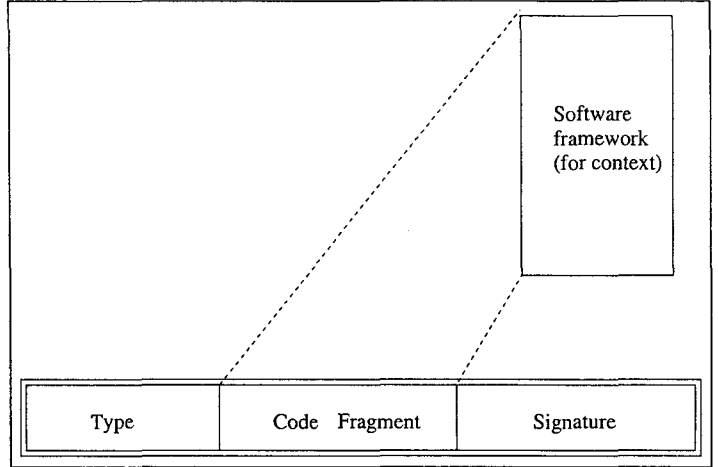


Figure 1: Generic Active Capability

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-

tegrity 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

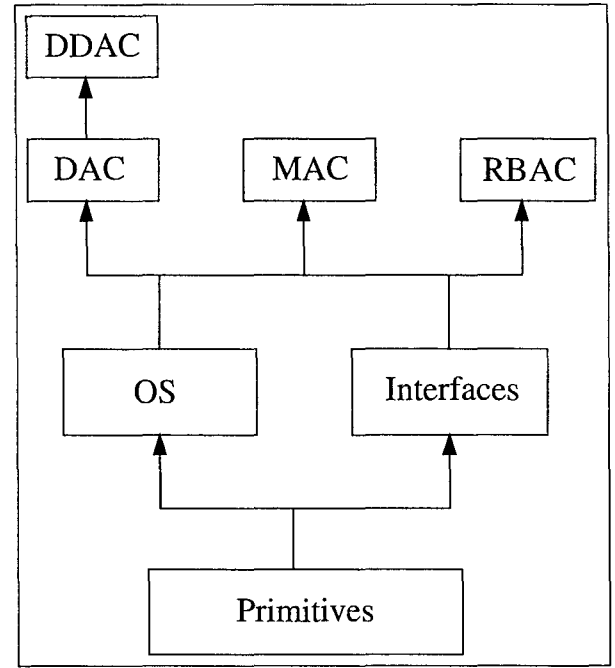


Figure 2: Component-level Map of the Policy Framework

several good examples). In our model, we can specify not only the $\langle \text{subject}, \text{object}, \text{operation} \rangle$ 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

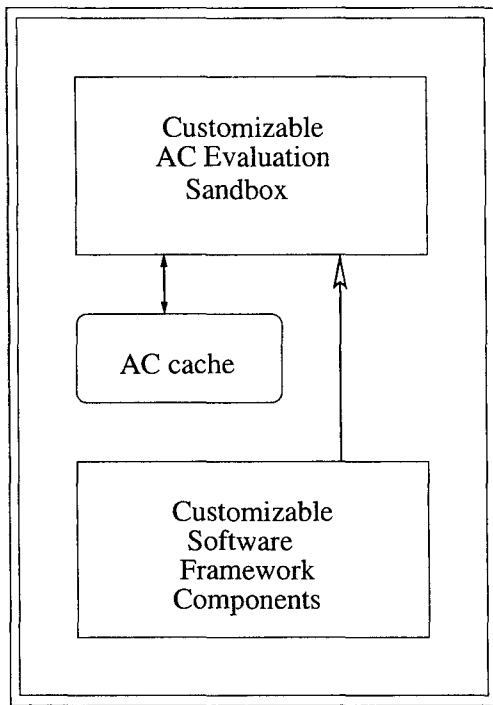


Figure 3: Evaluation and Enforcement engine

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

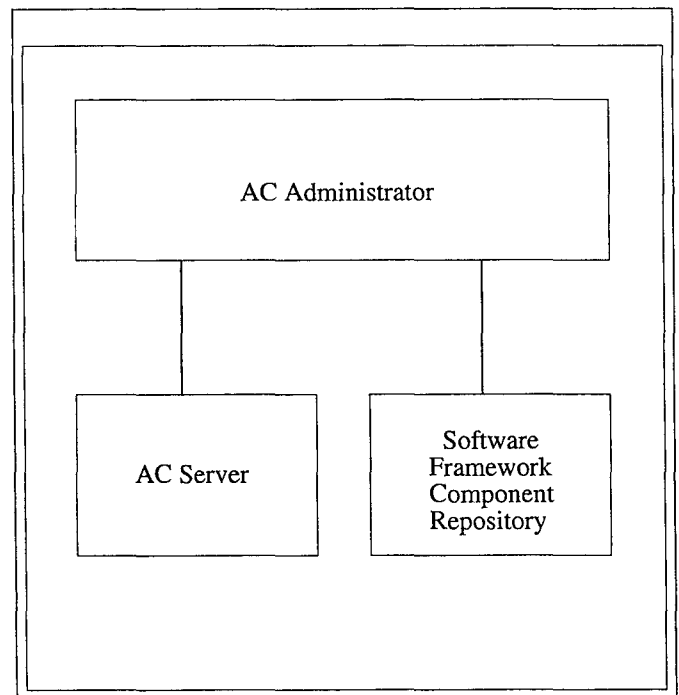


Figure 4: AC management infrastructure

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

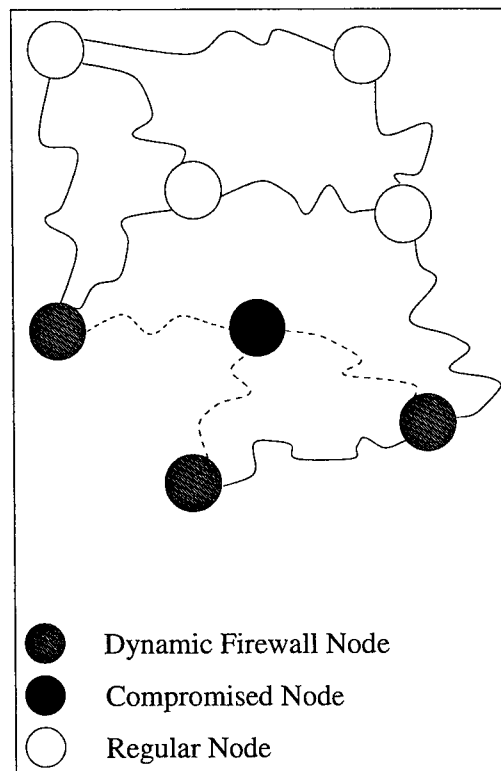


Figure 5: Dynamic Countermeasures for Intrusion

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.

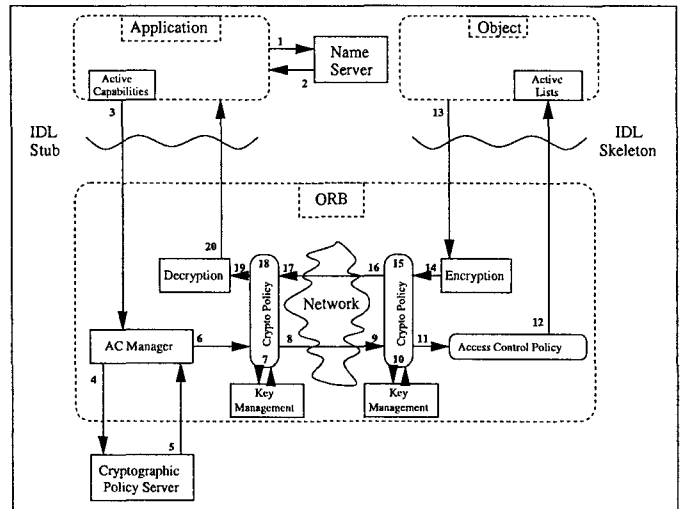
4 Implementation

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

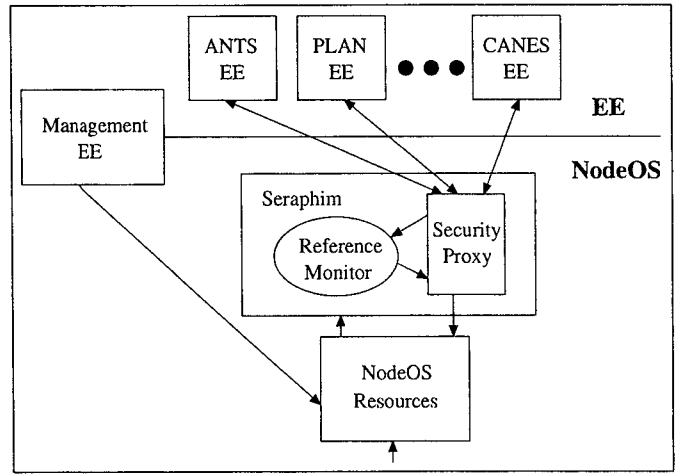


Figure 7: Secure Active Network Node

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

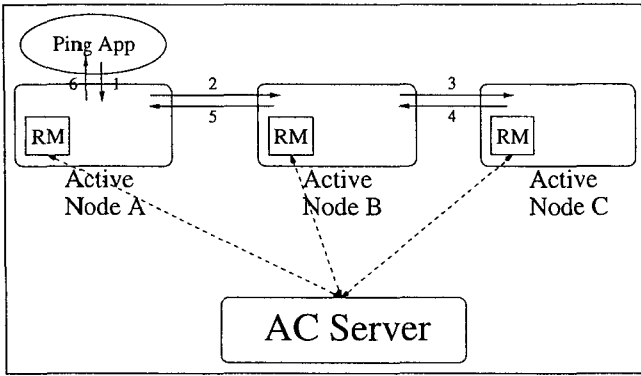


Figure 8: Ping Application Experiment in *Seraphim*

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.

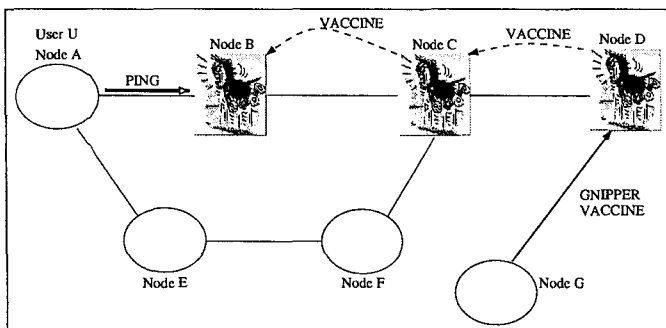


Figure 9: Gnipper Application

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. Gutttag, 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. Gutttag. Introducing new internet services: Why and how. *IEEE Network Magazine*, July/August 1998.

Seraphim: Dynamic Interoperable Security Architecture for Active Networks*

Roy H. Campbell, Zhaoyu Liu, M. Dennis Mickunas, Prasad Naldurg, Seung Yi

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, IL 61801

{roy, zhaoyu, mickunas, naldurg, seungyi}@cs.uiuc.edu

Abstract

Security is an important concern in the active networking paradigm because a breach in security can quickly compromise many systems in the network. This paper describes an extensible, reconfigurable security architecture that is flexible and accommodates a wide variety of security policies and mechanisms. It provides applications and users the ability to create and enforce highly customized and situational policies dynamically, and is well-suited to the security issues in active networks.

Seraphim [7] implements this architecture and allows the creation of dynamic security policies. Innovative applications use these policies in an exploration of the nature and scope of “dynamic security”. The implementation facilitates research of interoperability and portability security issues. Based on the experience from this effort, we are investigating a unified model for security mechanisms that preserves security guarantees across domains.

Keywords: *active networks, security, policy, access control, active capability, reference monitor, interoperability, dynamic, reconfigurable*

1 Introduction

Active networks aim to provide a software framework that enables network applications to customize the processing of their communications. 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.

The traditional definition of security includes authentication, access control, and encryption. Active network applications and routers can establish a basis for trust through mutual authentication. Encryption and digital signatures can protect the privacy and integrity of the active network capsules that contain code and data. Access control mechanisms and security policies can provide

controlled access to router resources and routed code and data. Much security research for active networks focuses on providing a secure environment for the routers that primarily

- prevents malicious behavior of arbitrary user code and
- protects the user code and data from malicious routers [22].

Research at MIT identifies the use of active networks to locate and neutralize the source of unwanted network traffic like ping and ack packets [25]. This kind of application of active networks suggests the possibility of developing “active security”, programmable security policies and mechanisms that respond dynamically to the activities on the network.

Our research complements these efforts and emphasizes the possible applications of active security in an active network. In the course of our research, we have studied applications of active security to the issues of interoperability and dynamic security policies. Most existing security provisions, especially regarding policies and access control, are static in nature. Once a security system is deployed within a network, the provisions are difficult to change and modify dynamically. For example, most security systems cannot change security policy implementations in response to a successful security attack. Though a wide range of security policy types have been proposed, most systems implement a common subset of these policies and mechanisms. Applications that require sophisticated security requirements and customized security policies must use lesser or weaker security guarantees provided by the deployed static security system. In the spirit of the motivation for developing active networks, our approach provides a unified security framework that allows users or applications to create and enforce their own security provisions and policies, similar to customizing their own communication protocols. We identify two example application scenarios that benefit from such flexible security support:

- **Dynamic Firewall Formation:** The creation of dynamic protection domains or enclaves is very useful

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

in many situations. In this way, a security system can build agile and dynamic firewalls in reaction to the detection of a security attack, thereby isolating the target of the attack from its attacker. When an active security administrator or trusted authority detects intrusion, it can send out an active capsule carrying a security agent with the appropriate vaccine. This can be used to build a dynamic line of defense against outside attacks and to raise the level of security within the domain. When the threat disappears the administrator or trusted authority can transmit another active capsule in order to resume normal operation. This can be used as a very powerful security tool in conjunction with intrusion detection and countermeasure systems.

- **Secure Emergency Multicasts:** Emergency notification of events like a storm warning must be secure to be effective. We describe a “geo-casting” scheme to alert or warn subscribers about natural disasters like tornadoes passing through a geographic region. A multicast group is formed using dynamic join and leave, based on geographic information. As the tornado moves, the multicast group can move along with it by adding and removing appropriate receivers in real time, using a fast join and leave. Active capsules can be sent to meteorological “subscribers” in a larger area, or to mobile users to warn that they are entering a danger area. Active security is used to reduce false alarms and to prevent unauthorized use of the system.

In this paper, we present a dynamic, fully extensible, interoperable security architecture based on and built into the underlying active network architecture. This architecture allows the configuration of existing active network routers with only a minimal set of security functions. These functions are used to recursively install and support the secure deployment of new security mechanisms. For instance, sophisticated and application-specific or user-specific security functions may be installed at run time using a secure recursive reconfigurable bootstrapping process. Currently, our framework provides mechanisms to specify, separate and enforce a number of different and often mutually exclusive access control policies dynamically. In addition we allow applications to encapsulate credentials and to encode situational policies that are authenticated by a trusted policy server to add, alter or revoke existing access control rules and mechanisms dynamically. Applications write active network code which uses these credentials and policies to inject customized security into the routers. Much of the architectural framework has been built and tested in Seraphim [7, 16]. Thus, applications may choose an access control policy and enforce this policy on their active network code. The framework can provide consistent security policy guarantees and platform independent enforcement of security policies across all active routers.

Section 2 of the paper gives a brief overview of the im-

portant terms and presents a self contained tutorial on some of the background material. Section 3 describes our architecture and talks about its place in the general active network architecture. Section 4 focuses on the implementation of the reference monitor and its flexible policy framework. This allows us to create and enforce dynamic policies. Section 5 talks about our testbed implementation and some of the experiments that we have performed using our testbed. Section 6 talks about related work and the final section talks about our conclusions.

2 Background

Although our research investigates dynamic security provisions for all aspects of security including authentication, access control, and encryption, in this paper we emphasize access control. 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 does not scale to systems that 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 the security that it enforces. A variety of types of access control policies have been defined in the literature including 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 which may be represented directly by the matrix model. Typically a DAC policy implementation maintains an indexed list of allowed $\langle \textit{subject}, \textit{object}, \textit{operation} \rangle$ triples. Unix file system permission is a simplified example of a DAC policy. DDAC maintains two lists, an “allowed list” similar to DAC and a “denied list”. MAC policies use the concept of labeling. A military example of labeling has labels “Top Secret”, “Secret”, “Confidential”, “Classified” and “Unclassified”. MAC is used in trusted operating systems. 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 that allow them to write to objects with equal or greater labels only. Subjects with lower labels cannot read from objects with higher labels. This is often called the “no read up, no write down” rule [11]. This hierarchy strictly controls the flow of information.

Among the policies listed, RBAC is the most flexible

type of access control policy [23]. 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. For example, system security cannot be dynamically changed from a DAC policy to a MAC policy. 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.

3 The Architecture

This section gives a brief overview of the basic active network architecture [6] as proposed by the architecture working group to provide a context for our security architecture. The software on an active router consists of three distinct, functionally separate layers: the application, the EE (Execution Environment), and the NodeOS. The NodeOS is similar to the kernel of traditional operating system. On an active network router, this component also performs resource allocation and management. Typical resources include shared memory, communication channels, and routing tables. The EE runs on a NodeOS and provides an interpreter for capsule code. An EE behaves like a user shell that has access to and can manipulate routing tables and packets. The EE provides an interface for accessing the NodeOS resources. Applications create capsules that include both the code to run on the EEs and communication data. The active network installs and executes the capsule code dynamically on remote routers.

Following is a brief and self-contained overview of our security architecture which is called Seraphim. The major components of our architecture and their interactions in the context of the active network architecture are shown in Figure 1.

The key component of Seraphim is a reference monitor. The reference monitor is implemented as a co-located extension to the Node OS. Every node has a reference monitor through which all accesses to the node resources occur. The policy framework is a component of the reference monitor. The policy framework itself is reconfigurable and it can be downloaded dynamically when required. Applications or administrators use the interface provided by the policy framework to create a customized piece of code that encodes the type of access control policy and other constraints used in the access control decision making process. This code fragment is called the *active*

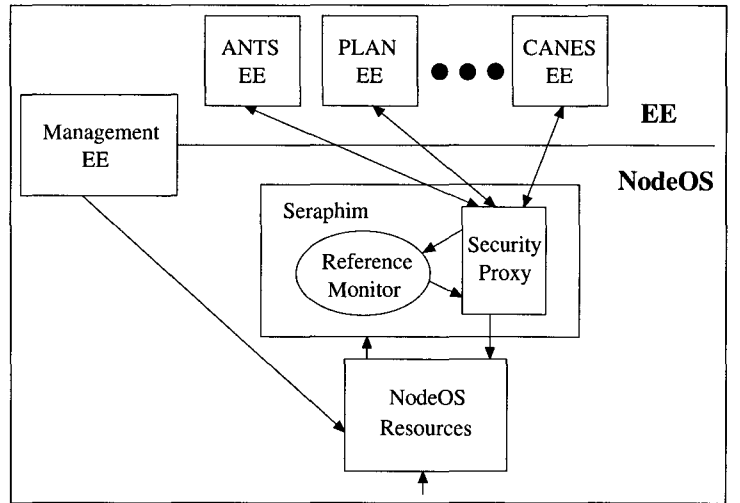


Figure 1: Secure Active Network Node

capability (AC) [16, 9, 8].

Unlike a traditional capability, which is merely a static authorization credential that encodes the principal and the permissions associated with the principal, an active capability is actually an executable Java bytecode in our implementation. In addition, an active capability is protected by digital signatures, resides in user space, and can be freely passed around. Conceptually, an active capability is a piece of unforgeable code that encodes a critical, application-specific part of the decision making code used in access control.

By using an active capability we can encode various situational policies that depend on system attributes. For instance, by writing a piece of code that checks the current system time and compares it with a value stored in the active capability we can introduce a policy that expires after a certain time deadline. Similarly, various enforcement and revocation schemes based on other attributes like quota, history, and information content can be implemented. These schemes are very useful in an open internetworking environment with diverse application requirements. An application can use quota-based revocation to limit the amount of system resources a client can consume. This is useful to counter denial of service attacks.

An active capability relies on a policy framework for context. An application presents an active capability along with its regular data or protocol capsules to the active router's reference monitor at execution time. The access control policy type and user credentials are extracted from the capability. The remote router's reference monitor recreates the context of the policy type within its policy framework. If at any point during this process, the policy framework discovers that it does not have an implementation for the type of the policy, it downloads the code dynamically into the framework, using the underlying active network. It then instantiates the run-time parameters associated with the application in its sandbox-like environment and executes the active capability in this

environment. Based on the result of the evaluation of this active capability, the access control decision is enforced.

The principal of the active capability, which is typically an application user, must be authenticated by a trusted authority. The trusted authority also acts as the policy server in our system. This entity is responsible for generating and keeping track of the active capabilities. Usually, we associate one or more policy servers with each protection domain. Application programs contact their nearest or least-loaded server and obtain the active capability dynamically.

A security proxy component was added as a temporary module in our design. Presently, the active network community is still working on the specifications of a standardized NodeOS interface [20]. In order to provide interoperability between an application written for any EE and our reference monitor, we need an entity that intercepts the requests to NodeOS resources and redirects them to the reference monitor. At this point the EEs direct their requests for NodeOS resources to the security proxy which sits on top of the NodeOS. The proxy acts as a wrapper to the NodeOS API and redirects access requests to the reference monitor. The reference monitor evaluates the request and passes the result on to the proxy. Depending on the result the proxy either forwards the request to the NodeOS or returns it to the EE with a denial notification.

The next section describes the reference monitor, the policy framework and active capabilities in detail.

4 Active Capabilities, Policy Framework, and Reference Monitor

In our approach, active capabilities distribute access control information including security policies within an active network. Security provisions are componentized so that complex policies and controls may be dynamically downloaded component by component. Active capabilities (ACs) are capsules (or part of capsules) that encapsulate security code like a security policy or access control decision. Policy servers operate as a communication front-end for distributing executable security policies in the form of ACs. An AC may either provide all the code for a security policy or access control, or it may specify a policy server from which to retrieve code. A reference monitor is used to intercept application and active network capsule code resource requests. The reference monitor applies the appropriate security access controls to resource requests. As the access controls are applied, the security code in an AC may request further policies that must be downloaded from a policy server.

Typically, traditional security systems are designed to enforce one particular type of security policy like MAC or DAC. Security policies are usually static and are not easy to change once deployed. In many cases, the security policies are specified in a policy language and com-

piled to an implementation that provides access control. In our approach, security policies are mobile agents or downloadable executable code in the form of ACs. In order to help users with policy specification, we provide an object-oriented policy representation framework in Java. The policy representation framework consists of a hierarchy of classes as shown in Figure 2.

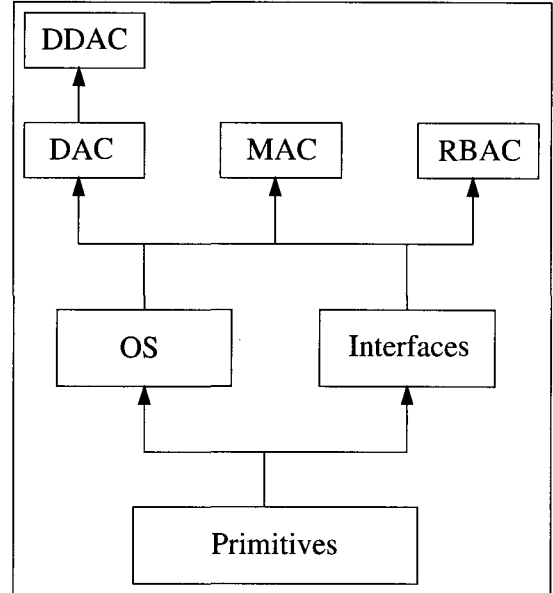


Figure 2: Component-level Map of the Policy Framework

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 more specialized classes representing concepts such as labels and access control lists. At the top of the framework are classes which can be used to represent a variety of generic policy forms.

A policy framework that places a heavy burden on its users will not be popular. With this in mind, we provide a policy framework GUI which makes the process of creating new policies or specializing existing policies as painless as possible. Typically, to use a security policies, most users will just select one of a list of predefined policies or use the default settings chosen by a system administrator. However, our approach also allows system administrators and expert users to create and modify policies that respond to specific application needs or security threats. The goals of our policy framework are to allow predefined policies to be enforced efficiently and effortlessly and also to provide a convenient interface for policy authors to create more sophisticated policies.

The current policy framework supports the following common types of access control policies: Mandatory Access Control (MAC), Discretionary Access Control (DAC), Double Discretionary Access Control (DDAC), and Role-based Access Control (RBAC) [21]. More application specific access control policy systems can be easily extended from this object-oriented framework ([14]

provides several good examples). In our model, we can specify not only the $\langle \text{subject}, \text{object}, \text{operation} \rangle$ access control triple, but also include a resource limit on usage, situational decision rules, constraints and dependences, e.g., based on current time of the day or current role of the principal.

Our framework also lets users specify pre-conditions and post-conditions. Pre-conditions allow necessary security checks to be performed before the actions take place, and post-conditions can be used to maintain state and perform additional checks after the action has been completed and when more information becomes available.

The policy framework is used in both the domain policy administrator and the reference monitor. Figure 3 shows the interaction of the various components of the policy administration mechanism.

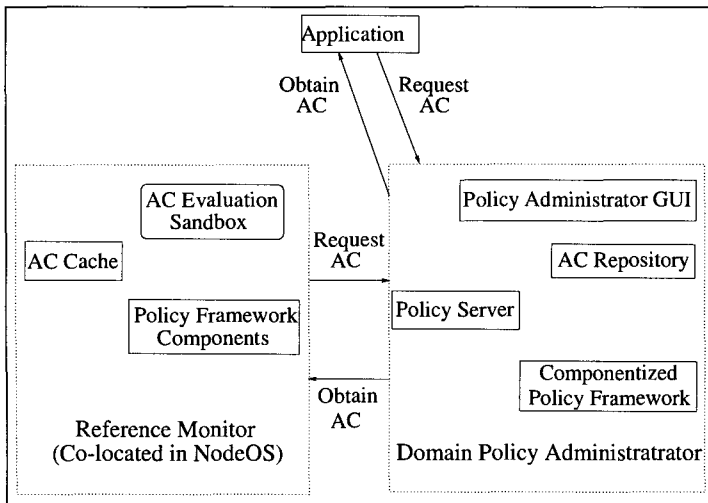


Figure 3: Policy Administration

There are three ways to pass an AC to the reference monitor:

- The applications can create application specific ACs or obtain them from the policy server and then send the AC along with active capsules. The AC may be embedded into the active capsule, or it may be an active capsule itself. When a capsule arrives at a remote node, it is demultiplexed to the appropriate EE, which maintains the state concerning the capsule and recognizes protocols and flows. The EE presents the AC to the security proxy along with its request to a NodeOS resource.
- If the application capsule does not have an AC, upon receiving the resource request via the EE, the reference monitor contacts the domain policy server directly, and asks for the AC associated with the principal of the application capsule.
- For common applications or frequent users, the policy server may distribute the ACs in advance to the reference monitors during system initialization.

For frequently occurring operations like IP forwarding, dynamically changing capabilities are not necessary. Access control rules tend to be static and caching provides a short circuit or fast path processing alternative for such requests. On the other hand, our application section demonstrates some protocols that can benefit greatly by using the expressibility afforded by the power of dynamic capabilities.

In order to improve the AC evaluation efficiency, the reference monitor uses a cache to store the ACs, or even the result of AC evaluations. Depending on the freshness and type of the AC, a request may be satisfied by a simple cache lookup instead of an expensive AC evaluation. On the other hand, for some types of capabilities, the reference monitor can always download the latest capability from the policy server. Caches are purged periodically to maintain their freshness. We plan to improve the efficiency and optimize the cache consistency protocol used in our architecture.

Another important attribute of this architecture is the ability of the trusted authority to revoke a capability at any point in time. The trusted authority can send a “purge cache” message to the relevant reference monitors and install a new capability at run time. Alternately, the application can present a properly signed new capability during run-time with a newer version number which invalidates the existing capability.

4.1 Discussion

We are using the JDK1.2 security API to do simple key generation and management and AC authentication. We plan to leverage ongoing work in this area, and integrate it with our system. Of particular interest are the systems being developed by Network Associates, Inc. [17] and KeyNote [1]. Using the attributes of the two systems we plan to define an infrastructure that again allows users to pick and choose the best attributes from either system dynamically.

Another problem that needs to be addressed is low-level code safety in the reference monitor. The minimum requirements for low-level code safety are control flow safety, memory safety, and stack safety [15]. Currently we rely on the Java byte code verifier [28] to provide low-level code safety. Before loading a class, the verifier performs data-flow analysis on the class code to verify that it is type safe and that all control-flow instructions jump to valid locations.

There are several other approaches for low-level code safety. The PLAN project [1] uses programming language techniques to address the code safety problem. Capsules are written using a strongly typed, resource limited language and dynamic code extensions are secured by using type safety and other mechanisms. Another approach is Proof-Carrying Code (PCC) [19]. Besides regular program code, PCC carries a proof that the program satisfies certain properties. The proof is verified before the execution of the code. The generation of a proof may be complex and time consuming, while its verification should

be simple and efficient. Software fault isolation (SFI) [26] provides another alternative for low-level code safety. It uses special code transformations and bit masks to ensure that memory operations and jumps access only the correct memory ranges.

Here again, a variety of different mechanisms and protocols have been proposed. Each method has its own advantages and disadvantages. Ultimately the application must be given the choice to pick the mechanism that is most suitable for its purpose. We plan that in the future our framework will be generic enough to allow all these mechanisms to co-exist, using the same principles that guided the design of our experimental policy framework.

5 Experimental Testbed

Our initial testbed implementation is based on the ANTS toolkit developed at MIT [27]. The original toolkit was written before the architecture group was formed, and did not reflect the layered architecture proposed by the architecture working group. Our first task was to split the design into layers and separate the functionality of the original *Node* class into distinct *NodeOS* and *EE* components. We added the Seraphim reference monitor between the *EE* and the *NodeOS*. We called the modified system *SAINTS* (Secure Active Inter-operable Network Toolkit System). Our *SAINTS* is backwards compatible with original ANTS and can run original ANTS applications¹.

5.1 Using the Testbed

In order to use our testbed, the policy server has to be initialized first. The policy server is the trusted third party for the testbed. It acts as a front-end to the policy framework classes and allows applications to create active capabilities. Currently we do not provide support for dynamic policy negotiation but allow multiple security domains to exist. When the policy server is started, it also starts the policy administrator. The policy administrator starts a GUI which allows users or system administrators to create and define policy-specific attributes and generate active capabilities. Users of system administrator can choose any policy type from DAC, DDAC, MAC, or RBAC. In this section, we are going to show the usage of the DAC policy, the most simple one, in the Gnipper application, and then the usage of the RBAC policy, the most flexible and complicated one, in the dynamic secure multicast application.

A screenshot of the GUI for DAC is shown in Figure 4. The user or administrator selects and sets the policy type to DAC. In order to create a new active capability, the user types in a file extension and clicks on the “New DAC File” button. To reuse existing capabilities, the “Load DAC File” button is used, after specifying the file extension. To add (remove) ACs into policy specification the user name, the object and the allowed operations on that object are all entered in the appropriate

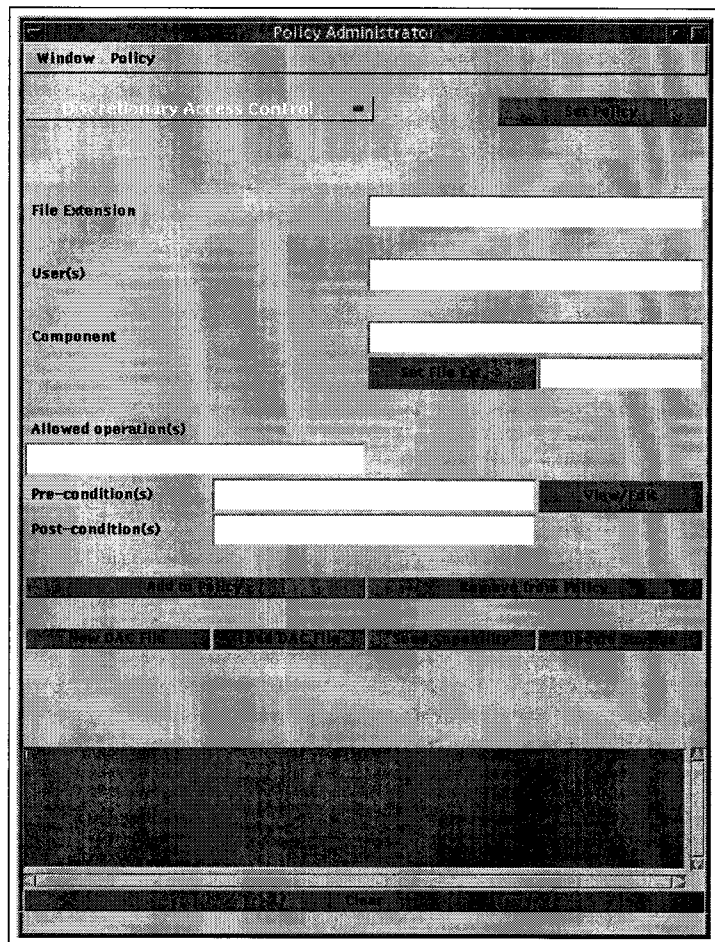


Figure 4: Policy Administrator GUI for DAC

fields and the “Add to Policy” (“Remove from Policy”) button is clicked. Post-conditions and pre-conditions are also added if necessary. The ACs can be stored using the “Update Storage” button, and be retrieved at any point in time, by using the “Load DAC File” command. The “Send Capability” button is used to send the ACs to a particular user or reference monitor. To test the policy specification the GUI also provides an “Evaluate” option in the “Policy” menu.

The GUIs for the other policy types are similar to the DAC GUI. A screenshot of the RBAC GUI is shown in Figure 5. The RBAC GUI supports more functionality and allows the administrator to create role definitions and to associate users and permissions with the role.

5.2 Applications

As a part of our testing and development phase, we developed several interesting, yet conceptually simple, applications on our testbed to demonstrate the significant advantages of our architecture. These include the Gnipper application and the secure multicast applications. The next two subsections give a brief overview of these experiments.

¹Contact authors for the Seraphim software release

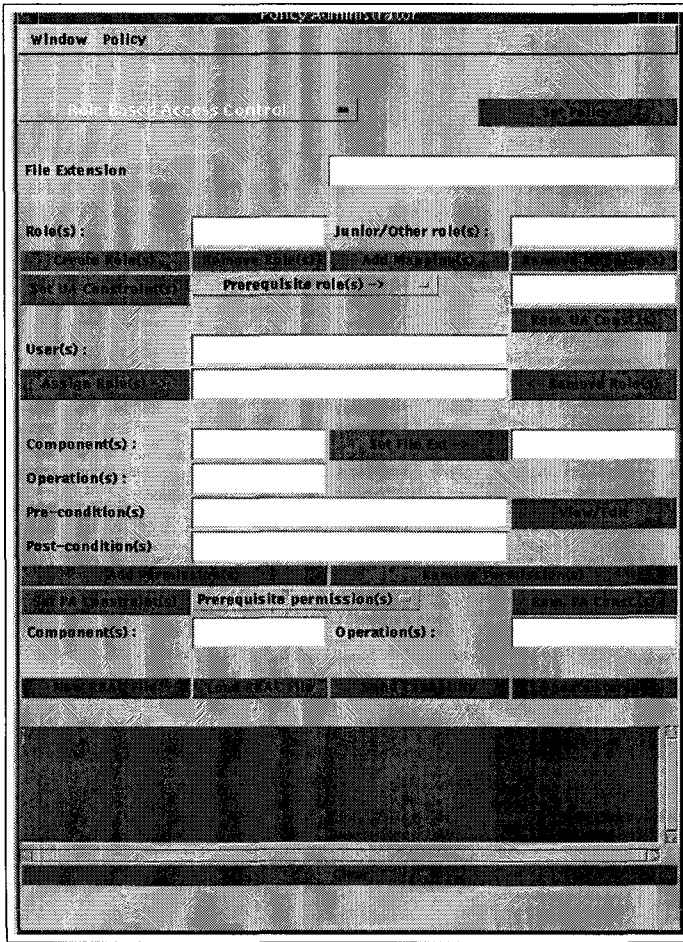


Figure 5: Policy Administrator GUI for RBAC

5.2.1 Gnipper

The Gnipper experiment demonstrates the creation of dynamic protection domains or enclaves. This is best explained with the help of an example presented below (Figure 6).

In our example, User U at Node A is trying to discover the network topology and sends out a Ping capsule with destination address of Node D. Ping packets may be unwelcome because they may be used in a denial of service attack or because of privacy. If we decide that Node D should be secure from Ping requests from User U at Node A, then we create a Gnipper vaccine and install it at the

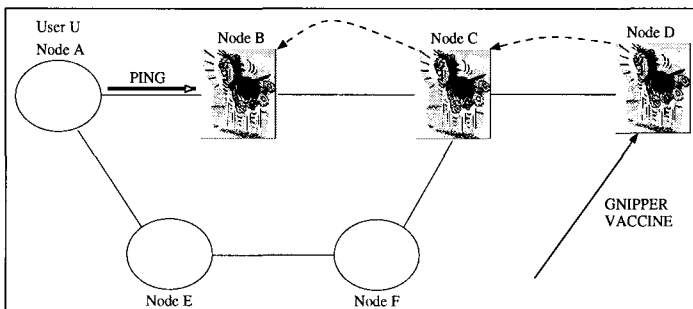


Figure 6: Gnipper Application

reference monitor of Node D. The vaccine, an anti-Ping AC, will disable the ability of User U at Node A to ping Node D. So when the Ping capsule from User U arrives 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. So the vaccine dynamically moves one-hop at a time toward the source of the Ping, in response to Ping requests for original sender. It is important to note here that the vaccine is reversible and the system administrator or trusted authority can send another active capability and install it on Node D dynamically, purge the caches on Node B and Node C and allow Ping from User U at Node A to reach Node D. Then the normal execution of the Ping is resumed.

The exact node which drops the Ping request changes dynamically depending on the number of Pings generated by the source and the pinging routes. For example, although Node E and Node F can also be used for Ping, the vaccine is not installed on those nodes because they were not used in the previous Ping attempt from Node A. By selectively broadcasting the vaccine on the frequently used routes between Node A and Node D, we have succeeded in building a dynamically growing firewall around Node D, and also reduced the traffic and moved the denial of service attack away from the intended victim.

Although our prototype implementation was not built for performance, we did make preliminary performance measurement (Refer to [16] for more performance measurements). 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 100Mbps local Ethernet LAN was measured as 77ms. Without creation, the average 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. Similarly, a firewall can be built dynamically around a compromised node to isolate the victim. 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 countermeasure systems.

5.2.2 Dynamic Secure Multicast

The dynamic multicast application was intended to showcase the benefits of using the RBAC policy implementation and to demonstrate the creation of dynamic multicast groups. In addition it demonstrates a range of different situational specifications using active capabilities.

Most of the existing secure multicast schemes are based on sharing a secret session key among the subscriber

nodes to ensure privacy of data. The main problem with this approach is the prohibitive overhead associated with the need to change the session keys whenever a person leaves the group. This is necessary, in order to make sure malicious members do not continue to listen to multicast data. In order to facilitate dynamic joining and leaving, we use active capabilities to grant and revoke users access to sensitive multicast data in our experiment. When a user joins, the trusted authority installs an active capability that gives the user privileges to receive the multicast data, at the reference monitor of the user's local node. When the user leaves the group, the active capability for the user is simply revoked by the trusted authority. Our experimental scheme has a much lower overhead compared to the traditional schemes.

The multicast program we used is a modified version of the sample multicast application in the original ANTS toolkit. Using our modified multicast application, we devised two different scenarios. The first one is a cable-TV style "Pay-Per-View". A user who wishes to receive a sequence of special multicast packets contacts the trusted authority and obtains an active capability that has a resource limit built into it. This capability is then installed in the reference monitor of the user's node. Every time a special data packet is delivered to the node, the resource limit is decremented by one. When the resource limit reaches zero, the active capability expires and user can no longer receive the special multicast data traffic. If the user wishes to receive more, then the user has to pay again and get another active capability with the appropriate resource limit.

The actual implementation was done using role based access control (RBAC) policy. Users were assigned a default role, that did not let them receive any of the special multicast data packets. Once they "paid", their role was replaced by a "special" role that gave them the access rights for a predetermined number of special multicast data packets. When the resource limit reached zero, the "special" role was expired and the original default role was resumed.

The second scenario demonstrates the use of time-stamped active capabilities for control access. This experiment is a cable-TV style "Sneak-preview". Any user in the multicast group obtains, say, two minutes worth of free multicast data. Active capabilities are obtained and installed in the similar way as in the first scenario. Once installed, the active capability keeps track of the local time and expires after two minutes have elapsed. We are assuming here that applications cannot alter the local time, as it is a protected resource.

Both the "Pay-Per-View" and "Sneak-preview" experiments dramatically reduce the amount of state maintained by the server, compared to the state maintained by existing traditional secure multicast solutions. By eliminating the need for secret session keys, these experiments distribute the server processing load among all the participating routers and allow asynchronous, client-side initiated joining and leaving.

Based on our current efficient, dynamic, and secure multicast, we plan to conduct a "geo-casting" experiment that alerts or warns subscribers about natural disasters like tornadoes passing through a geographic region. A multicast group is formed using our dynamic join and leave, based on geographic information. As the tornado moves, the multicast group also moves by adding and removing appropriate receivers in real time, using our efficient join and leave. The proper active capabilities can be supplied to meteorologist "subscribers" in a larger area, or to mobile users to warn that they are entering a danger area.

5.2.3 Other Applications

Currently we are integrating our Seraphim security system into the CANEs[2] congestion control and error recovery multicast application. We use our RBAC policy to control the signaling procedure of CANEs, and to control the installations of different protocols dynamically. We also use our framework to control access to data packets dynamically.

We also provide Seraphim security services to an NS2 simulation [3] of a new secure multicast routing protocol. In the NS2 simulation, some multicast groups need higher security routing. The MAC policy of Seraphim assigns different security clearance levels to routers. The system then obtains the security information of the routers in the network, for a particular multicast group, based on their security clearance level. The simulation uses this information to set up proper secure multicast routing trees. The Seraphim reference monitor functions as the enforcement engine and ensures that the multicast joins follow the established security level hierarchy.

6 Related Work and Discussion

Little research has been done in security policy management and domain interoperability. In traditional systems, security policy defines access control which 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 or centrally. For example, each firewall in a company can be either configured individually to establish a set of rules defined by policy, or managed by a centralized policy administrator such as the Cisco Secure Policy Manager for firewalls [10]. The policy changes are expected to occur infrequently in traditional systems. More recently, Bhatt et al. [4] used self-managed and self-organized mechanisms for automating network management. Naccio of MIT [13] provided a high-level approach for safety policy expression and enforcement, which is implemented for enforcing policies on JavaVM classes. Schneider characterized a class of enforceable security policies [24] and there was an automata implementation [12] to enforce such policies.

The security working group [18] of the active networks

research community has been instrumental in publicizing and highlighting the importance of security in active networks. The security draft emphasizes the importance of incorporating security into the initial design stage of the active network architecture itself. As mentioned earlier, we believe that we can classify security related research in this field into three general categories. The first one deals with the more traditional notion of security. It includes authentication, access control, policies and enforcement. Some examples are protection of valuable information using encryption, providing data integrity using signatures. Public key infrastructure (PKI) and key distribution and management problems fall into this category. The security working group [18] has launched some important exploratory research in this direction.

The second category is related to security associated with the mobile nature of the environment. Protection of nodes from mobile code originating in foreign domains and protection of active packets or code from malicious hosts fall in this category. The PLANet effort [1] raises some of the issues associated with these protections. In addition they also provide a bootstrapping module that ensures that the system configures itself correctly at startup or reboot time. The protection from mobile code is provided by using a type-safe, resource limited, functional programming language with dynamic type verification. Mobile code can install protocols at nodes securely by using the extensibility features provided by the language.

Our research focuses on the third category: dynamic security. We believe that our work is complementary to the other research and attempts to enhance the flexibility and to improve usability of their techniques. By componentizing the security policy framework we provide an infrastructure to enforce any kind of expressible security policy. Using our infrastructure, applications can specify, implement, and enforce fine grained access control policies. These policies can be created, changed or revoked on the fly and enforced at run-time. Traditional mechanisms can be configured as components in our systems and their context can be instantiated and enforced on demand. The safety features provided by the bootstrapping and language features can be incorporated as an integral component of our framework.

However, there is more to dynamic security than simply dynamically deploying and enforcing security policies and mechanisms. We believe that we are barely touching the surface when it comes to exploring the potential applications and the limits of dynamic or active security. The combination of the active nature of the underlying architecture and the flexibility and dynamic nature of our framework has thrown open a new frontier for exploration and discovery. Examples like the tornado-watch and dynamic multicast demonstrate a fresh, alternative and functional approach to existing problems.

We are in the process of integrating our work within our active network working group and will demonstrate the flexibility and portability of our framework by incor-

porating it into CANEs [2]. We already have a implementation version that runs on the Abone [5]. In the future we plan to refine our techniques and define the protocols for interactions between heterogeneous security domains. In addition we also plan to explore applications of our framework to non-traditional ubiquitous computing environments and to integrate the applications into the active networking architecture.

Any node that uses our security has simply to add our reference monitor as an extension. The reference monitor will provide mechanisms for accessing advanced and composable services. The reference monitor can be used in an EE or Java environment, as well as in a NodeOS. We will also support the concept of domains and will provide domain-level policy conflict resolution and negotiation in the future.

In order to make the downloading of policy framework secure and to simplify the adding of extensions, we are in the process of developing a prototype of the Management EE. The Management EE will aid in managing the NodeOS, will initialize meta-level policies and will provide a framework for secure bootstrapping. We are also working on the design of a generic framework for key management.

7 Conclusions

In this paper, we describe a prototype security architecture that complements the basic active network architecture and augments its functionality. The flexibility and expressibility afforded by this implementation framework enables us to implement a multitude of diverse, innovative and exciting applications. These applications exploit the active networking paradigm without compromising the security of the infrastructure. In addition, our architecture lays the ground rules for seamless integration with parallel and ongoing efforts in the active networks community.

With our prototype implementation, we developed applications that demonstrate the benefits of our infrastructure. In particular, the Gnipper application demonstrates the creation of dynamically growing protection domains using vaccines. The multicast experiments showcase the use of our framework as an alternate approach to tackling the key-distribution and revocation problems associated with secure multicast applications.

In summary, we believe that our approach is a step in the direction of designing a comprehensive and flexible framework to integrate various security mechanisms and services into the active network architecture. It also provides a foundation for discussing issues related to co-existence, inter-operation and portability of these mechanisms. At the same time, our architecture imposes minimum overhead on the existing infrastructure and allows applications to specify and enforce customized security mechanisms conveniently.

References

- [1] The SwitchWare Project Homepage
<http://www.cis.upenn.edu/~switchware/>.
- [2] CANEs Project Homepage
<http://www.cc.gatech.edu/projects/canes>.
- [3] UCSC Multicast Research Homepage
<http://www.cse.ucsc.edu/research/ccrg/>.
- [4] S. Bhatt, A. V. Konstantinou, S. R. Rajagopalan, and Yechiam Yemini. Managing security in dynamic networks. In *13th USENIX Systems Administration Conference (LISA'99)*.
- [5] Bob Braden and Livio Ricciulli. A plan for a scalable Abone - a modest proposal, January 1999.
- [6] K. Calvert et al. Architectural framework for active networks. AN Architecture Working Group, Draft, 1998.
- [7] 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/>.
- [8] Roy H. Campbell, M. Dennis 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.
- [9] 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.
- [10] 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>.
- [11] D. Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Company, 1982.
- [12] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *DARPA Information Survivability Conference and Exposition*, Hilton Head Island, SC, January 25-27, 2000.
- [13] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 9-12, 1999.
- [14] 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.
- [15] Dexter Kozen. Efficient code certification. Technical Report 98-1661, Department of Computer Science, Cornell University, January 1998.
- [16] Zhaoyu Liu, Prasad Naldurg, Seung Yi, Tin Qian, Roy H. Campbell, and M. Dennis Mickunas. An agent based architecture for supporting application level security. In *DARPA Information Survivability Conference and Exposition*, Hilton Head Island, SC, January 25-27, 2000.
- [17] Sandra Murphy. Active Networks Mailing List.
- [18] Sandra Murphy et al. Security architecture for active nets. AN Security Working Group, July 15, 1998.
- [19] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL '97)*, pages 106-119, January 1997.
- [20] L. Paterson et al. NodeOS interface specifications. AN NodeOS Working Group, Draft, 1999.
- [21] 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.
- [22] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agent Security, LNCS 1419*. 1998.
- [23] R. S. Sandhu and E. J. Coyne. Role-based access control models. *IEEE Computer*, 29(2), February 1996.
- [24] F. B. Schneider. Enforceable security policies. Technical Report 98-1664, Department of Computer Science, Cornell University, January 1998.
- [25] Van C. Van. A defense against address spoofing using active networks. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, May 1997.
- [26] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP '93*.
- [27] 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.
- [28] Frank Yelin. Low-level security in Java. In *WWW4 Conference*, December 1995.

Flexible Secure Multicasting in Active Networks*

Zhaoyu Liu, R. H. Campbell, S. K. Varadarajan, Prasad Naldurg, Seung Yi, M. D. Mickunas

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, IL 61801

{zhaoyu, roy, svaradar, naldurg, seungyi, mickunas}@cs.uiuc.edu

Abstract

In this paper we describe an alternative, flexible approach to multicast security in active networks. Traditional schemes for securing multicast communication have key management and scalability problems for many typical applications. In addition, traditional mechanisms are not capable of expressing flexible, situational security policies for multicast sessions and participants. Our scheme exploits the computational power of active networks to provide dynamic, flexible security for multicast applications. One of the main advantages of our scheme is the low communication and key distribution overhead associated with multicast group management.

Our approach is based on the Seraphim security architecture implementation [5], which uses active capabilities [6] for access control. Seraphim is an extensible, reconfigurable security architecture that is flexible and accommodates a wide variety of security policies and mechanisms. It also provides applications and users the ability to create dynamically and enforce highly customized and situational policies. Using these policies we have developed several secure multicast applications that demonstrate the flexible nature and low overheads associated with our architecture.

Keywords: *multicast, security, active networks, scalability, flexibility, policy, access control, active capability, reference monitor*

1. Introduction

Multicast is a useful network service that provides efficient, best-effort data delivery from a source to multiple recipients. The use of multicasting is becoming more and more widespread, as is demonstrated by the popularity of the experimental Mbone multicast service and its supporting applications. With video con-

ferencing via the Internet becoming extremely popular, multicast becomes an important technique to reduce sender transmission overhead, network bandwidth, and the latency observed at the receiver side. As multicast applications are widely deployed, the need to secure multicast communications becomes critical.

Providing security to multicast still remains a challenging problem due to the difficulties involved in securely distributing a session key. Traditional approaches involve a central key distribution center and these approaches do not scale well, especially when the multicast group members were scattered across a wide area network. These schemes rely on the existence of an asymmetric key infrastructure where every member who wishes to be part of the multicast network, needs to possess a private/public key pair. They also suggest that the new session key be unicast to each and every member, encrypted in the member's private key. The Core Based Tree (CBT) [4] and the Protocol Independent Multicast (PIM) [8] were proposed to address the issue of scalability, while papers like [7] introduced newer methods of key generation and distribution.

However, even in these approaches, one requires sender-specific keys to authenticate the sender of a message. New members must be given the session key and probably participate in a mutual authentication protocol. Also, all these methods require that the group (or session) key be changed when a particular member leaves the group, to prevent that user from eavesdropping on the rest of the traffic. Of particular concern here is the mechanisms used for distributing the new session key to the remaining users. The distribution overhead is usually very large [11]. This inefficiency makes traditional secure multicast applications very hard, if not impossible, to be used in very dynamic environment, which has frequent multicast joins and leaves.

In this paper, we propose a new approach for multicast security based on active networking technology and our dynamic policy framework. We use *reference monitors* at active routers and an *active capability* to

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

control the delivery of multicast packet. Active capabilities are essentially active capsules that are signed and concisely encode access control policy specifications and other security attributes associated with the underlying active protocol, e.g., multicast. Reference monitors provide a sandbox like environment on the routers and securely execute the active capabilities and enforce the policy specified in them. Using this approach we can flexibly support various secure multicast applications and reduce the communication overhead considerably.

Our paper is organized as follows. Section 2 of the paper gives a brief overview of our architecture and talks about its place in the general active network architecture. It also focuses on the implementation of the reference monitor and its flexible policy framework, which allows us to create and enforce dynamic policies. The next section talks about one particular type of access control policy, Role Based Access Control (RBAC) policy. The fourth part of the paper describes the dynamic secure multicast experiments in detail. The fifth part discusses the related work and the final part presents our conclusions.

2. Overview of *Seraphim* Architecture

An active network [15] provides a software framework that enables network applications to customize the processing of their data. Active applications inject capsules that contain programs (along with data) into the network. Active routers dynamically install these programs and execute them on the data. The basic software on an active router consists of three distinct, functionally separate layers: the application, the EE (Execution Environment), and the NodeOS. The NodeOS is similar to the kernel of traditional operating system. The EE is similar to a user shell. It provides an interface for accessing the NodeOS resources and an execution environment for application capsules. By using the shell provided by the EEs, applications create capsules with protocol code and/or data that can be installed dynamically in remote routers.

In order to provide security to the active networks infrastructure, and to provide dynamic, interoperable, and application customized security policy, we have implemented an architectural framework based on and built into the underlying active network architecture. The major components of our architecture and their interactions, in the context of the active network architecture are shown in Figure 1.

The security proxy component was added as a temporary module in our design, since the active network community is still working on the specifications of a standardized NodeOS interface [12]. The proxy acts as

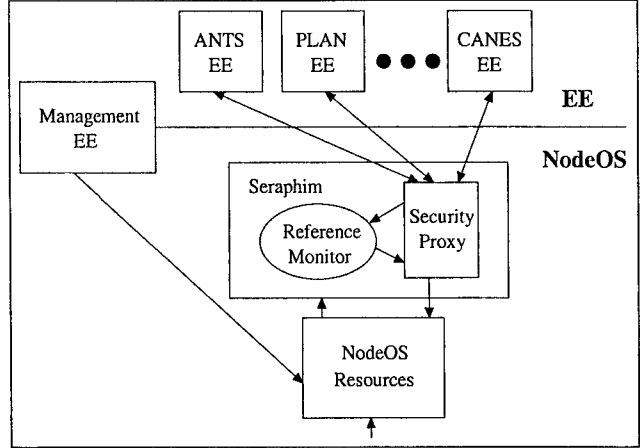


Figure 1. Secure Active Network Node

a wrapper to the NodeOS API and redirects the access requests to the reference monitor and then the evaluation results to applications.

2.1. Reference Monitor and Active Capability

The key component of our architecture is the reference monitor. The reference monitor is implemented as a co-located extension to the NodeOS. Every node has a reference monitor through which all accesses to the node resources occur. A dynamic, reconfigurable policy framework is developed as a component of the reference monitor for application customized policies. The policy framework itself is reconfigurable and it can be downloaded dynamically when required. Applications or administrators use the interface provided by the policy framework to create a customized piece of code that encodes the type of access control policy and other constraints used in the access control decision making process. This code fragment is called the *active capability* (AC) [10].

Unlike a traditional capability, which is merely a static authorization credential that encodes the principal and the permissions associated with the principal, an active capability is actually an executable Java script in our implementation. In addition, an active capability is protected by digital signatures, resides in user space, and can be freely passed around. Conceptually, an active capability is a piece of unforgeable code that encodes a critical, application-specific part of the decision making code used in access control. The active capability can encode various situational policies that depend on system attributes such as the current system time, resources, quota, etc.

An active capability relies on a policy framework for context. An application presents an active capability along with its regular data or protocol capsules to

the active router's reference monitor at execution time. The access control policy type and user credentials are extracted from the capability. The remote router's reference monitor recreates the context of the policy type within its policy framework. If at any point during this process, the policy framework discovers that it does not have an implementation for the type of the policy, it downloads the code dynamically into the framework, using the underlying active network. It then instantiates the run-time parameters associated with the application in its sandbox-like environment and executes the active capability in this environment. Based on the result of the evaluation of this active capability, the access control decision is enforced.

The principal of the active capability, which is typically an application user, must be authenticated by a trusted authority. The trusted authority also acts as the policy server in our system. This entity is responsible for generating and keeping track of the active capabilities. Usually, we associate one or more policy servers with each protection domain. Application programs contact their nearest or least-loaded server dynamically and obtain the active capability dynamically.

2.2. Policy Framework

In order to support ACs and provide users more flexibility in terms of policy specification, we have implemented an object-oriented policy representation framework in Java. This allows users and commercial organizations to specify policies tailored to their specific operational needs. The framework itself is a hierarchy of classes as shown in Figure 2.

The framework 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 incremented 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.

The policy framework supports the following common types of access control: Mandatory (MAC), Discretionary (DAC), Double Discretionary (DDAC), and Role-based (RBAC) [13]. More application specific access control policy systems can be easily extended from this object-oriented framework ([9] provides several good examples). In our model, we can specify not only the traditional $\langle \text{subject}, \text{object}, \text{operation} \rangle$ access control triple, but also include a resource limit on usage, situational decision rules, constraints and dependences, e.g., based on current time of the day or current role of the principal.

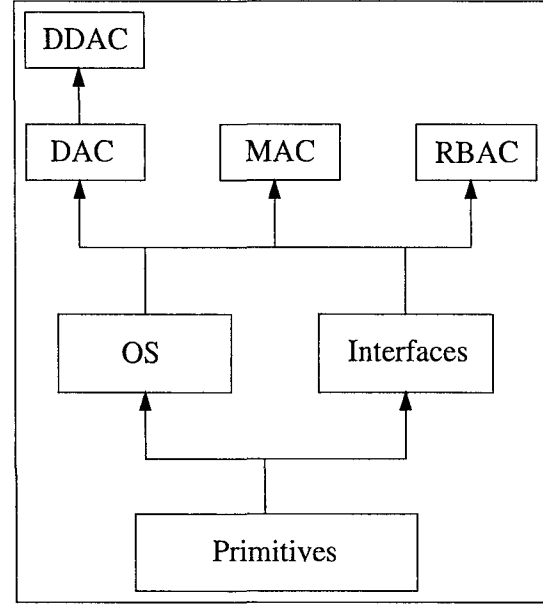


Figure 2. Component-level Map of the Policy Framework

Our framework also lets users specify pre-conditions and post-conditions. Pre-conditions allow necessary security checks to be performed before the actions take place, and post-conditions can be used to maintain state and perform additional checks after the action has been completed and when more information becomes available. An administrator GUI is provided as front end to the policy framework (Figure 3).

2.3. Dynamic Access Control

In order to apply our policy framework to active networks, we use active capabilities to distribute the permission information of the policies. We also componentize the framework so that it can be dynamically downloaded component by component. Active capabilities (AC) encapsulate a customized piece of code that encodes the type of access control policy and other constraints used in the access control decision. The framework uses the policy server as a communication front-end to accept AC requests either from reference monitors or applications and to provide the requested customized AC. Figure 3 shows the interaction of the various components of the policy administration mechanism. The policy administrator and the policy server are trusted entities.

There are three ways to manage the distribution of the ACs to the reference monitor:

- The applications can create and obtain application

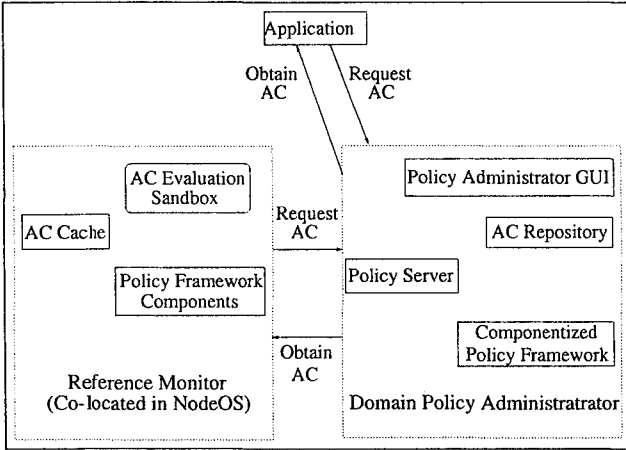


Figure 3. Policy Administration

specific ACs through the policy server GUI and send the AC along with active capsules. When a capsule arrives at a remote node, it is demultiplexed to the appropriate EE. The EE presents the AC to the security proxy along with its request to a NodeOS resource.

- If the application capsule does not have an AC, upon receiving a resource request via the EE, the reference monitor contacts the domain policy server. The policy server responds to this request with the appropriate customized AC.
- For common applications or frequent users, the policy server may distribute the ACs in advance to the reference monitors during system initialization.

Another important attribute of this architecture is the ability of the trusted authority, represented by the policy administrator or server, to revoke a capability at any point in time.

3. Role Based Access Control (RBAC) Policy

Our policy framework includes Role Based Access Control (RBAC) policy type, which is used in the secure multicasting applications presented in this paper. A Role Based Access Control policy, as the name suggests, uses the concept of a role as its basis for representing permissions [14]. It is a form of access control that emerges in the context of security policies for organizations. A role is chiefly a semantic construct that forms the basis for an access control policy. With RBAC, system administrators create roles according to the job functions performed in an organization, grant permissions to those roles, and then assign users to the

roles on the basis of their specific job responsibilities and qualifications. The idea is that the particular combination of users and permissions brought together by a role tends to change over time while the permissions associated with a role are themselves relatively more stable.

The biggest advantage that RBAC has over other forms of access control is that it is extremely intuitive to use and maps easily to real-world situations. A hierarchy of roles with senior roles inheriting all the permissions of junior roles closely follows the structure of organizations. The access control policy in RBAC is embodied in components such as role-permission, user-role and role-role relationships. These components collectively determine whether a particular user is allowed access to a particular operation on a particular component. These individual components can be easily (and intuitively) configured to provide the required degree of access control. For example, adding a new user to a system would merely involve assigning appropriate roles to the user according to the user's functions in the organization. Likewise, changing the nature of, for example, printer access, for all managers in an organization can be accomplished by merely changing the permissions with the manager role in the organization. All managers can immediately see the effects of the change.

RBAC is the most flexible type of access control policy. 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. The constraints supported in our RBAC implementation include three important ones [13]:

- Mutually exclusive roles/permissions. This is the most common RBAC constraint. The same user can be assigned to at most one role in a mutually exclusive set. This ensures separation of duties.
- Prerequisite roles/permissions. If a role is declared a prerequisite for another role in the system, it means that a user may belong to the latter role only if the user already belongs to the first one.
- Cardinality constraints. This constraint imposes a limit on the number of users that can be assigned to a role. Similarly, this constraint can limit the

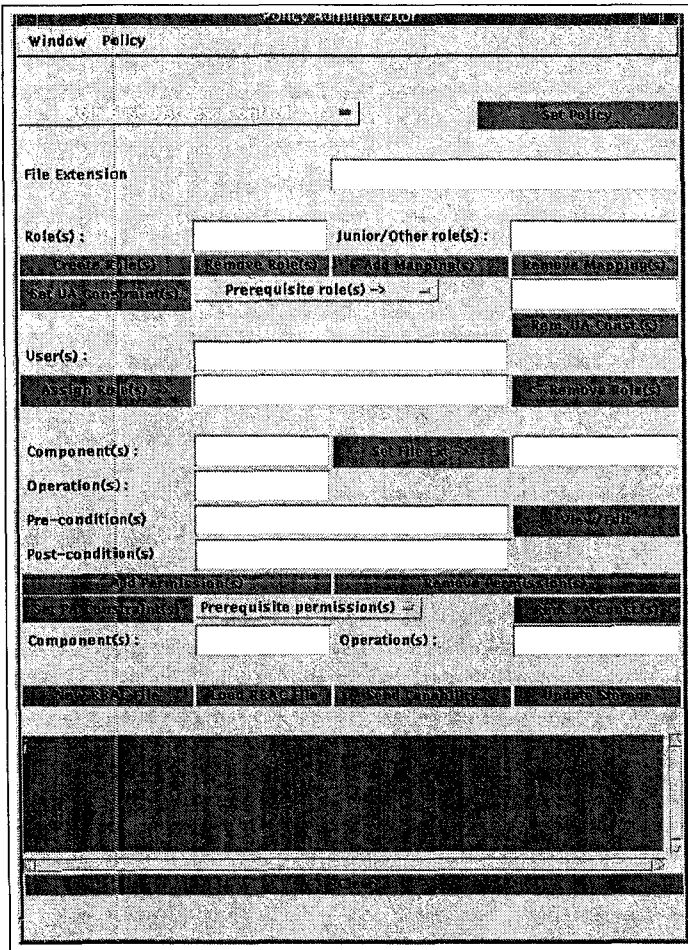


Figure 4. Policy Administrator GUI for RBAC

number of roles that a permission can be assigned to.

The policy administrator uses a GUI which allows users or system administrators to create and define policy specific attributes, and generate active capabilities. A screenshot of the RBAC GUI is shown in Figure 4. This GUI allows the administrator to create role definitions and associate users and permissions with the role, and supports other functionality (see [13] for more details).

4. Dynamic Secure Multicast

In this section, we are going to describe several dynamic multicast applications. The purpose is intended to showcase the benefits of using the RBAC policy class implementation and demonstrates the creation of dynamic multicast groups in active networks. In addition we also demonstrate a range of different policy specifications using active capabilities.

Our testbed implementation is based on the ANTS toolkit developed by MIT [15]. The original toolkit was written before the architecture group was formed, and did not reflect the layered architecture. Our first task was to split the design into layers and separate the functionality of the original Node class into distinct NodeOS and EE components. This modification is backward compatible and the original ANTS applications can run in our Secure Active Interoperable Network Toolkit System (SAINTS) [5].

4.1. The Multicast Tree Formation

While the ANTS implementation of multicast subscription was used for the multicast join, it provided no implementation for multicast leave. Modifications were made to the ANTS multicast subscription code, and the multicast unsubscription capsule was also written, to take care of dynamic multicast leave. This results in automatic construction or pruning of the multicast tree.

The multicast tree is dynamically formed and pruned by the generation of subscription and unsubscription capsules. The multicast subscription and unsubscription capsules are sent by a receiver towards the group owner (sender). These capsules dynamically construct and prune the multicast tree, respectively, when they are evaluated in the active nodes along the path from the receiver to the sender.

The Figure 5 show the steps of the multicast tree is formed when two receivers join a particular group in our implementation. The sender in the figure may refer to the Core in CBT [4] or the Rendezvous Point in PIM [8].

Similarly, when an unsubscription capsule is passed upwards, links are removed and the tree is pruned as shown in Figure 6, using active network features.

4.2. Pay-Per-View and Sneak Preview

As mentioned earlier, most of existing secure multicast schemes are based on shared session key among the subscriber nodes to ensure privacy of data. The main problem with this approach is the prohibitive overhead associated with the need to change the session keys every time a member leaves the group. In order to facilitate dynamic joins and leaves, our implementation uses active capabilities that grant and revoke access to the sensitive multicast data to the users in the multicast group. When a user decides to join, the trusted agent installs a capability that gives the user privilege to receive the multicast data, on the user's local node. The trusted agent may delegate this responsibility to the multicast source node or intermediate active routers.

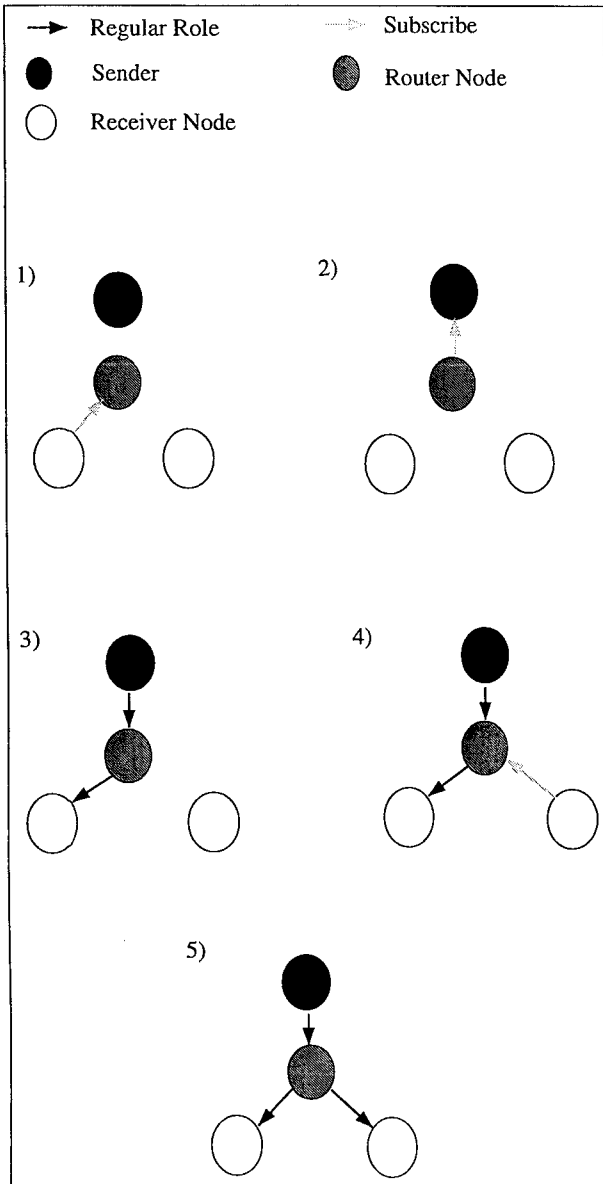


Figure 5. Multicast Subscription in Active Network

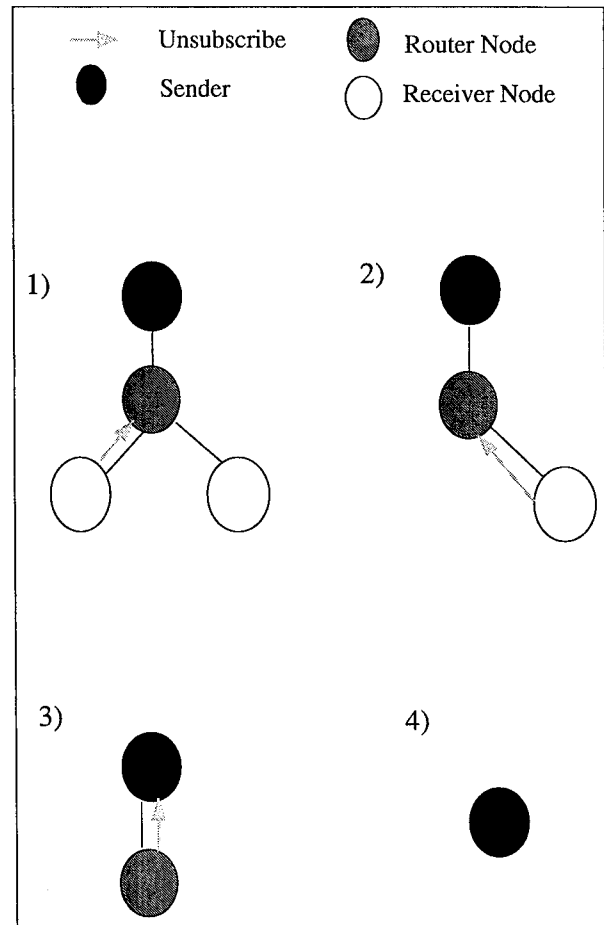


Figure 6. Multicast Unsubscription in Active Network

When the user leaves the group, the active capability for the user on the local node is revoked by the trusted agent. By doing that, we don't need to change any shared secret among the subscribers and thus can reduce the large session key distribution overhead. Our experiment scheme has a much lower overhead compared to the traditional schemes in place today.

Using our modified multicast application from ANTS, we devised two different experiment scenarios. In the first experiment or "Pay-Per-View", any user that wishes to receive a sequence of special multicast packets contacts the trusted agent and obtains an active capability that has a resource limit built into it. This capability is then installed in the reference monitor. Every time a special packet is delivered to the node, the resource limit is decremented by one. When the resource limit reaches zero, the active capability expires and the user can no longer receive the special multicast data traffic. If the user wishes to receive more, then the user "pays" and gets the another active capability with an appropriate resource limit.

The actual implementation was done using Role Based Access Control (RBAC). Users were assigned a default role (*Regular* role), that did not let them receive any of the special multicast data packets. Once they "paid", their role was replaced by a *Special* role that gave them the access rights for certain number of special multicast data packets (Figure 7). When the resource limit reached zero, the *Special* role was revoked and the default *Regular* role was resumed. A simple extension of this example is selective blocking. The sender can revoke the roles of selected nodes and implement dynamically changing selective blocking for different domains.

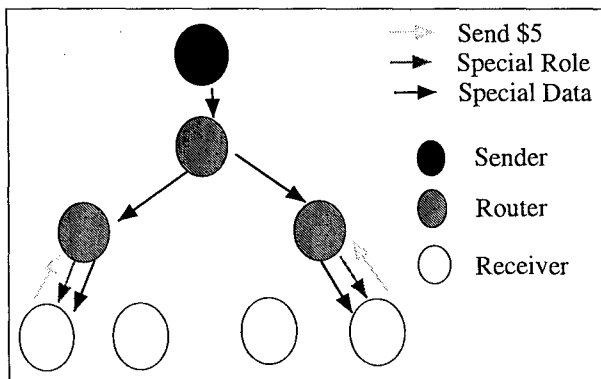


Figure 7. Pay-Per-View

The second demo used time-stamped active capabilities to control the access control decision. This experiment, or "Sneak-preview" allows any user in the multicast group to get some fixed time period (for example, two minutes) worth of free multicast data. Active ca-

pabilities are obtained and installed in the same way as in the first experiment. Once installed, the active capability keeps track of the local time and expires after two minutes have passed. The users may choose to receive any two minute period of the multicast traffic, all at once or in parts at any time. When the two minutes are used up, the active capability expires and user cannot receive multicast data anymore.

Both the "Pay-Per-View" and "Sneak-preview" experiments dramatically reduce the amount of state maintained by the server, compared to the state maintained by existing secure multicast mechanisms. And by eliminating need for using or changing session keys, these experiments distribute the server processing load among all the participating active routers and allow asynchronous, client-side initiated joining and leaving.

A possible extension of our applications is secure emergency multicasts. Emergency notification of events like a storm warning must be secure to be effective. "Geo-casting" scheme can be used to alert or warn subscribers about natural disasters like tornadoes passing through a geographic region. A multicast group is formed using dynamic join and leave, based on the geographic information. As the tornado moves, the multicast group can move along with it by adding and removing appropriate receivers in real time, using our fast join and leave. Active multicast data capsules can be sent to meteorological "subscribers" in a larger area, or to mobile users to warn that they are entering a danger area.

5. Related Work and Discussion

Currently active networks research community is applying active technology to multicast in the areas of congestion control [1] and error recovery [2]. We are integrating our Seraphim security system into them. We use our RBAC policy to control the signaling procedure of CANEs, and to dynamically control the installations of different protocols. We can also use our framework to control access of data packets dynamically.

We also use Seraphim to provides security service to an NS2 simulation [3] of a new secure multicast routing protocol. In the simulation, some multicast groups need higher security routing. The system gets the security information of the routers in the network, for a particular multicast group, based on their security clearance level. The simulation uses this information to set up proper secure multicast routing trees. The MAC policy of Seraphim is the access control policy for this application, which assigns different security clearance levels to routers. The Seraphim reference monitor functions as the enforcement engine and ensures that

the multicast joins follow the established security level hierarchy.

The overhead of our reference monitor and AC evaluation with proper cache scheme is negligible [10]. A efficient, distributed AC management is important for the overall system performance. Further research is required to design and implement such an AC management system.

6. Conclusions

This paper presents an alternative approach to secure multicasting. In the traditional approach when a user leaves a secure multicast group, the sender typically has to send messages to all the subscribers, in the entire multicast tree or subtree and change the session keys. Furthermore, during the transition period some of the sensitive information may be compromised. When a large number of users leave, and in some cases, rejoin, then this communication overhead, requiring reliable distribution¹ of new session and updating becomes prohibitive.

In our proposal, active capabilities carry the security information, using point-to-point secure channels. When a receiver leaves, an active capability changing the receiver's role is sent to the corresponding receiver router. Our solution does not rely on the multicast infrastructure for reliable delivery of an active capability. The reference monitor which is a co-located extension to our NodeOS kernel in active routers provides a safe, sandbox like environment for the execution of these active capabilities and dynamic enforcement of the policy associated with the active capabilities. The overheads associated with key distribution and scaling also dramatically diminish.

In addition, by embedding situational policies like pay-per-view, sneak-preview, selective blocking etc., we have demonstrated that our framework is flexible enough to satisfy a variety of secure multicast specifications. The dynamic nature of our security makes it very attractive for interactive sessions and video-conferencing.

References

- | | | |
|---|--------------------|----------|
| [1] CANEs | Project | Homepage |
| http://www.cc.gatech.edu/projects/canes . | | |
| [2] PANAMA | Project | Homepage |
| http://www.tascnets.com/panama/ . | | |
| [3] UCSC | Multicast Research | Homepage |
| http://www.cse.ucsc.edu/research/ccrg/ . | | |

¹Reliable delivery in multicasting is in itself a hard problem and various solutions that have been proposed have high overheads associated with them

- [4] T. Ballardie, P. Francis, and J. Crowcroft. Core based trees: an architecture for scalable inter-domain multicast routing. In *Proceedings of the ACM SIGCOMM '93*, San Francisco, CA, September 1993.
- [5] Roy H. Campbell, Zhaoyu Liu, M. Dennis Mickunas, Prasad Naldurg, and Seung Yi. Seraphim: dynamic interoperable security architecture for active networks. In *IEEE OPENARCH 2000*, Tel-Aviv, Israel, March 26-27, 2000.
- [6] 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.
- [7] G. Caronni, M. Waldvogel, D. Sun, and B. Plattner. Efficient security for large and dynamic multicast groups. In *Proceedings of the Seventh Workshop on Enabling Technologies, (WET ICE '98)*, IEEE Computer Society Press, 1998.
- [8] D. Estrin, D. Farinacci, et al. Protocol independent multicast – sparse mode (PIM-SM): protocol specifications. TETF draft, March 1997.
- [9] 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.
- [10] Zhaoyu Liu, Prasad Naldurg, Seung Yi, Tin Qian, Roy H. Campbell, and M. Dennis Mickunas. An agent based architecture for supporting application level security. In *the DARPA Information Survivability Conference and Exposition*, Hilton Head Island, SC, January 25-27, 2000.
- [11] Suvo Mittra. Iolus: a framework for scalable secure multicasting. In *Proceedings of the ACM SIGCOMM '97*, Cannes, France, September 1997.
- [12] L. Paterson et al. NodeOS interface specifications. AN NodeOS Working Group, Draft, 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] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: a toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH '98*, San Francisco, CA, April 1998.

Secure Information Flow in Mobile Bootstrapping Process*

Zhaoyu Liu, M. Dennis Mickunas, Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{zhaoyu, mickunas, roy}@cs.uiuc.edu

Abstract

The security of bootstrapping is very important for mobile computing. In this paper, we present the bootstrapping process of the Cherubim security system which uses a smartcard to allow a mobile Cherubim client to universally access remote standard services. In order to prevent any leakage of sensitive information, we apply a type system for secure flows to the bootstrapping source codes. The type system guarantees that well-typed programs satisfy a noninterference security property. This means that the program does not “leak” sensitive data. The type system also produces principal types for type-correct programs that characterize how programs can be called securely. The analysis demonstrates that the type system can ensure secure flow enforcement.

Keywords: bootstrapping, mobile, security, type system, information flow

1 Introduction

In a mobile environment, mobile clients usually need to download codes/services from the base server. Users can develop customized protocols to do the downloading. Preventing unintentional leakage of information is an important security issue here, although it is rarely considered by protocols developers.

The problem of secure information flow within systems having different sensitivity levels has been recognized widely and studied extensively. The early work was by Bell and LaPadula [2], and it was extended by Denning’s lattice-model [5, 6]. Denning used a program certification, an efficient form of static analysis that could be incorporated into a compiler to verify secure information flow in programs. Liskov *et. al* [9] proposed a method of using labeled types to control information flows. The labeled type consisted of a regular type such as `int`, and an added static label which was used to control information flows through standard static type-checking and notions of subtyping. Other more recent

efforts tried to extend the analysis to special language features like procedures and nondeterminism, while others focused on integrity only.

So far these efforts have not had much impact in practice, and there has not been a satisfactory treatment of the soundness of Denning’s analysis. The soundness would assure that if the analysis succeeds for a given program on some inputs, then the program executes securely. Although Denning provides some intuitive arguments and some account of information flow in terms of classical information theory, no formal soundness proof is attempted.

Recently, Volpano *et. al* [11, 13] took a type-based approach to the analysis and proved soundness which resembles traditional noninterference [8]. The certification conditions of Denning’s analysis are formulated as a simple type system. Basically a type system is a formal system of type inference rules for making judgments about programs. Traditionally, these rules are used to reason about type correctness of strongly-typed programs. However, they can be regarded, in general, as logical systems in which to reason about different program properties. Secure information is one such property.

Using a type system to analyze secure information flow has many advantages. It serves, in programs, as a formal specification that cleanly separates the security policies from the algorithms for enforcing them. Unlike other systems, the soundness of the type system can be formalized and proved. So type-correct programs have secure information flow. The secure flow type system provides a uniform framework exploiting traditional type checking in programming languages to ensure secure flow enforcement. So the issue of secure flows is no longer orthogonal to the more traditional type correctness issue of whether a program is well formed. Furthermore, standard type inference techniques can be applied to automate secure flow analysis. This makes the type system more practical than other models.

In this paper, we present a concrete bootstrapping process, and show how to practically apply Volpano’s type-based approach to it to improve security. We begin by the overview of bootstrapping process in the Cherubim security system and secure type system. Then we

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

apply the type system to examine information flow in the bootstrapping process. We conclude this paper by some discussions and future work.

2 Bootstrapping in Cherubim Security System

The *Cherubim* [10, 4] security architecture consists of CORBA compliant security services with security enhanced IDL providing application level security. It is all implemented in Java. The bootstrapping process [14] in Cherubim system allows mobile Cherubim clients, using a smartcard, to have universal remote access to standard services of the home server. This allows us to build a security system with maximum configurability and extensibility which is essential to many emerging applications like active networks and mobile computing. The bootstrapping process includes Diffie-Hellman key negotiation protocol, jurassic classloader, and supporting core security services. The core security services encapsulate the basic facilities, including encryption, digital signatures, and authentication, based on the top of standard Java Cryptographic API. They provide a uniform interface to an implementation that can be built with a variety of standard security components. In general, the classes in the Cherubim system are split into three categories that are loaded by different classloaders:

- **Primordial Classes:** The primordial classes contain the Java core classes (packages that begin with `java.`), as well as the necessary cryptographic code from the Cherubim and Java cryptographic packages. The Java core classes are assumed to be present and reliable on the client machine, while the other primordial classes are taken to the client machine by the user on a smartcard or similar device.
- **Jurassic Classes:** The jurassic classes consist of those classes present on the user's home machine. This includes the classes that make up the JacORB[3] object request broker, the classes that make up any applications that reside on the user's home machine, and the Cherubim policy library [7] (which consists of the basic blocks from which specific policies are constructed).
- **CORBA Classes:** The CORBA classes are loaded by a classloader using CORBA. These classes include specific policies that need to be evaluated prior to access remote objects and the classes that make up applications that do not reside on the users home machine. These classes are loaded using a Cherubim policy that is located on the user's home machine.

The focus of this paper is for jurassic classes, whose security concerns are essentially shared by general mobile applications. We concisely give an overview of our

bootstrapping process in the remaining of this section. For more details, please refer to [14]

The initial booting of the Cherubim System proceeds as follows:

1. Client machine boots its operating system and Java Virtual Machine (this may be done with AEGIS[1] or a similar system)
2. User runs *boot* program
3. *Boot* program prompts for passphrase
4. *Boot* program hashes passphrase using SHA-1 hash algorithm to an IDEA symmetric key
5. *Boot* program uses IDEA key to decrypt smartcard (including private keys)
6. Client machine makes a socket connection to the user's home machine
7. Home machine spawns a connection thread to communicate with the client
8. Client begins key negotiation with the server

In order to securely load the jurassic classes over the network to the client machine, encryption, authentication, and digital signatures are necessary. The bootstrapping system in Cherubim implements session key negotiation using the Diffie-Hellman protocol:

1. Client machine sends a SignedDHMessage (a signed Diffie-Hellman message, signed by the Smartcard) to the server
2. Server verifies the signature, timestamp, and destination on the message
3. Server sends a SignedDHMessage to the client
4. Client verifies the signature, timestamp, and destination on the message
5. Client and Server generate the shared secret
6. Client and Server hash shared secret using SHA-1 hash algorithm to an IDEA secret key

After this authentication has taken place, the client can then begin to use core security interfaces to request classes as follows:

1. JurassicClassLoader receives request for a class on the client
2. JurassicClassLoader checks to see if requested class is in the class cache and, if so, return it
3. JurassicClassLoader checks to see if the primordial classloader can load the class (i.e. if the class is in the CLASSPATH) and, if so, return it

4. JurassicClassLoader checks if existing session key is more than one hour old and, if so, negotiate a new one as above
5. JurassicClassLoader sends a SEClassRequest (a signed, encrypted class request, signed by the Smartcard and encrypted with the IDEA session key) to the home server over the existing socket
6. Server verifies the signature, timestamp, destination, and sequence number on the SEClassRequest
7. Server loads the requested class off the local disk, if available
8. Server sends the class in a SEClassResponse (a signed, encrypted class response, encrypted with the IDEA session key) to the client
9. JurassicClassLoader verifies the signature, timestamp, destination, and sequence number on the message
10. JurassicClassLoader adds the class in the message to the class cache
11. JurassicClassLoader returns the class in the message to the process that called it

In order to show clearly the information flow in bootstrapping, we show the call trees in Figures 1, 2, and 3. Later we will use secure type system to analyze bootstrapping based on these call trees.

3 Type Inference and Polymorphism

In secure flow type system, the basic types are security classes, which we denoted here by τ . The type inference rules are used to enforce secure information flow. For example, the inference rule for an assignment statement $x := e$ is:

$$\frac{\gamma \vdash x : \tau \text{ var}, \quad \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau \text{ cmd}}$$

So for the assignment to be well typed, it must be that

- x is a variable of type $\tau \text{ var}$, meaning x is capable of storing information at security level τ , and
- expression e has type (security class) τ .

Moreover, a command c has type $\tau \text{ cmd}$ only if it is guaranteed that every assignment within c is made to a variable whose security class is τ or higher. Similarly, the rule for the condition is as:

$$\frac{\gamma \vdash e : \tau, \quad \gamma \vdash c : \tau \text{ cmd}, \quad \gamma \vdash c' : \tau \text{ cmd}}{\gamma \vdash \text{if } e \text{ then } c \text{ else } c' : \tau \text{ cmd}}$$

```

proc decrypt(in key:int, inout charge:int,
            inout cipher,clear:array of char)

  var i := 0
  var unit := unit rate of constant
begin
  charge := unit;
  while cipher[i] > 0 do
    if encrypted(cipher[i]) then
      charge := charge + 2 * unit;
      clear[i] := D(cipher[i], key)
    else
      charge := charge + unit;
      clear[i] := cipher[i]
    fi;
    i := i + 1;
  od
end

```

Figure 4. The Library Decryption Procedure

and for the **while** loop:

$$\frac{\gamma \vdash e : \tau, \quad \gamma \vdash c : \tau \text{ cmd},}{\gamma \vdash \text{while } e \text{ do } c : \tau \text{ cmd}}$$

The reference[13] gives more complete treatment of inference rules.

The partially ordered security classes of flow policy naturally corresponds to the subtyping relations:

$$\frac{\tau \leq \tau'}{\tau \subseteq \tau'}$$

and (*antimonitonic* or *contravariant*):

$$\frac{\tau \subseteq \tau'}{\vdash \tau' \text{ cmd} \subseteq \tau \text{ cmd}}$$

A major advantage of the secure flow type system is that it can be implemented using a powerful type inference technique. A type inference algorithm not only proves whether a procedure has a secure type, or is free of illegal flows, but it also produces a principal type [12, 11], which succinctly conveys how the program can be executed securely. A principal type is a constrained type scheme with a constraint set of flat subtype inequalities among security levels. As long as the specification of the procedure's calling context satisfies the constraints, the procedure can be executed without causing any illegal flows. So the principal type of a procedure constrains the security classes of its formal parameters. In this sense, it is polymorphic type.

As an example from [11], consider the library decryption procedure in Figure 4. The encrypted character array *cipher* is decrypted using *key* and stored in the

```

static public void main(String[] args)

    public SecretyKey passphraseToSecretKey(String passphrase,
                                             String algorithm,int keylength,
                                             String hashAlgorithm,
                                             String characterEncoding)
                                             throws NoSuchAlgorithmException,
                                             UnsupportedEncodingException

        public byte[] hash(byte data[], String algorithm)
            throws NoSuchAlgorithmException

        public RawSecretKey(String algorithm, byte data[])

    public byte[] decrypt(byte data[], Key key)
        throws NoSuchAlgorithmException,
        IllegalBlockSizeException,
        BadPaddingException, KeyException,
        NoSuchPaddingException

    public JurassicClassLoader()

        public void newKey()

    public static void init(Core C, CherubimPrincipal P,
                           Network n, JurassicClassLoader j)

    public Class loadClass(String className)
        throws ClassNotFoundException

        public synchronized Class loadClass(String className,
                                             boolean resolveIt)
            throws ClassNotFoundException

        protected byte[] loadClassBytes(String className)

```

Figure 1. Main Call Tree

```

protected byte[] loadClassBytes(String className)

    private void newKey()

    public ClassRequest(String className, int sequenceNumber,
        Date timeSent, String destination)

    public byte[] sign(byte[] data)
        public byte[] sign(byte[] data, PrivateKey key,
            String algorithm)
            throws NoSuchAlgorithmException,
                InvalidKeyException, SignatureException

    byte[] encrypt(byte[] data)
        public byte[] encrypt(byte[] data, Key key)
            NoSuchAlgorithmException, KeyException,
            NoSuchPaddingException,
            IllegalBlockSizeException,
            BadPaddingException

    public SEClassRequest(String source, byte[] message,
        byte[] signature, String signatureAlgorithm)

        public SignedEncryptedMessage(String source, byte[] message,
            byte[] signature, String signatureAlgorithm)

            public SignedMessage(String source, byte[] message,
                byte[] signature, String signatureAlgorithm)

    public decrypt(byte[] data)
        public byte[] decrypt(byte[] data, Key key)
            throws NoSuchAlgorithmException,
                IllegalBlockSizeException,
                BadPaddingException, KeyException,
                NoSuchPaddingException

    public boolean verifyHome(byte[] data, byte[] signature)
        public verify(byte[] data, byte[] signature,
            PublicKey key, String algorithm)
            throws NoSuchAlgorithmException,
                InvalidKeyException, SignatureException

```

Figure 2. loadClassBytes Call Tree

```

private void newKey()

    public DHMessage start(String destination, String algorithm,
        int keylength, String hashAlgorithm)

        public DHMessage(String d, String a, int k, String ha,
            BigInteger m, Date t)

    public byte[] sign(byte[] data)

        public byte[] sign(byte[] data, PrivateKey key,
            String algorithm)
            throws NoSuchAlgorithmException,
            InvalidKeyException, SignatureException

    public SignedDHMessage(String source, byte[] message,
        byte[] signature, String signatureAlgorithm)

        public SignedMessage(String source, byte[] message,
            byte[] signature, String signatureAlgorithm)

    public boolean verifyHome(byte[] data, byte[] signature)

        public verify(byte[] data, byte[] signature,
            PublicKey key, String algorithm)
            throws NoSuchAlgorithmException,
            InvalidKeyException, SignatureException

    public DHMessage receive(DHMessage dhm, String source)

        public DHMessage(String d, String a, int k, String ha,
            BigInteger m, Date t)

    public SecretKey getkey()

        public byte[] hash(byte data[], String algorithm)
            throws NoSuchAlgorithmException

        public RawSecretKey(String algorithm, byte data[])

```

Figure 3. newKey Call Tree

clear array. The actual decryption is done by procedure D and the cost of doing the decryption is stored in variable `charge`. The inferred principal type for this procedure is as follows:

$$\forall \alpha, \beta, \nu, \gamma \text{ with } \beta \subseteq \nu, \beta \subseteq \alpha, \gamma \subseteq \beta . \\ \beta \text{ proc}(\nu, \gamma \text{ arr}, \nu \text{ arr}, \alpha \text{ var})$$

Any call of the procedure can be executed securely provided the arguments of the call have security classes that satisfy all the constraints. The call itself will have type $\beta \text{ cmd}$. To show the usefulness of the principal type, let's change the expression

```
charge := charge + 2 * unit;
```

in procedure of Figure 4 to the expression

```
charge := charge + key + 2 * unit;
```

in an attempt to leak the key variable information to the `charge` variable. Then the principal type of the procedure becomes:

$$\forall \alpha, \beta, \delta, \nu, \gamma \text{ with } \beta \subseteq \nu, \beta \subseteq \alpha, \gamma \subseteq \beta, \delta \subseteq \nu, \delta \subseteq \alpha . \\ \beta \text{ proc}(\delta, \gamma \text{ arr}, \nu \text{ arr}, \alpha \text{ var})$$

So one of the two additional subtype constraints $\delta \subseteq \alpha$ clearly says that the security level of the `charge` parameter must be at least that of the input `key`. Therefore any leakage is prevented.

4 Analysis of Bootstrapping Process

Type inference can formally produce principal types. Such principal types can be simplified by a formal type simplification scheme. In our case, we will manually deduct the principal types for essential functions of bootstrapping from the source codes. For simplification reasons, we only consider two security levels, *high* and *low*. In addition, there are several subtle points to be clarified as follows:

- we don't consider covert channels, including timing, and exception output from the Java language.
- A Diffie-Hellman key negotiation protocol introduces some randomness in its computation. In general, the soundness of the type system will no longer hold for a random computation. In our special case, we assume the protocol is correct and the session key generated from the protocol has a high security level. Therefore, soundness is not affected.
- Some information must be properly classified. Sometimes an algorithm will produce sensitive data from insensitive inputs. The type system will not detect that such data are actually sensitive. But we can package the algorithm as a procedure whose type can be asserted to reflect the different security

levels of the inputs and outputs. Key generation is one such example. On the other hand, sometimes an algorithm will produce insensitive data from sensitive data. Again we need to assert the correct type for such an algorithm. Encryption is one such example.

- We assume that Java cryptographic class library is trustworthy.

Let's first look at a simple example. With imported necessary Java packages, here is the source code of *decrypt*:

```
public byte[] decrypt(byte[] data, Key key)
    throws NoSuchAlgorithmException,
           IllegalBlockSizeException,
           BadPaddingException, KeyException,
           NoSuchPaddingException {
    Cipher cipher =
        Cipher.getInstance(key.getAlgorithm());
    cipher.init(Cipher.DECRYPT_MODE, key);
    return cipher.doFinal(data);
}
```

and simply the principal type is (here *var* actually means object):

$$\forall \alpha, \beta \text{ with } \alpha \subseteq \beta . \beta \text{ arr}, \beta \text{ proc}(\alpha \text{ arr}, \alpha \text{ var})$$

But the situation is quite different for *encrypt*. The source code of *encrypt* is:

```
public byte[] encrypt(byte[] data, Key key)
    throws NoSuchAlgorithmException,
           KeyException, NoSuchPaddingException,
           IllegalBlockSizeException,
           BadPaddingException {
    Cipher cipher =
        Cipher.getInstance(key.getAlgorithm());
    cipher.init(Cipher.ENCRYPT_MODE, key);
    return cipher.doFinal(data);
}
```

A casual comparison may conclude that the principal type should be the same as *decrypt*. This is not true. As mentioned at the beginning of this section, some information must be properly classified. The type of this procedure should be asserted to reflect the different security levels of the inputs and outputs. So the correct principal type is:

$$\forall \alpha, \beta \text{ with } \beta \subseteq \alpha . \beta \text{ arr}, \beta \text{ proc}(\alpha \text{ arr}, \alpha \text{ var})$$

A similar case holds for the *sign* function.

For subsequent analyses we simply give the principal types for relevant functions, without showing the source code. The call tree structures are shown in Figure 1, 2, and 3 for clarification.

```

oos.writeObject(new SEClassRequest(
    SystemState.getPrincipal().getName(), encrypt(cr), SystemState.getPrincipal().sign(cr),
    SystemState.getPrincipal().getSignatureAlgorithm()));

```

Figure 5. The Invocation of *SEClassRequest*

The principal type is useful in analyzing secure information flow caused by procedure invocations. For example, the principal type for *SEClassRequest* is:

$$\forall \alpha, \beta \text{ with } \alpha \subseteq \beta . \\ \beta \text{ var}, \beta \text{ proc}(\alpha \text{ var}, \alpha \text{ var}, \alpha \text{ var}, \alpha \text{ var})$$

An invocation of *SEClassRequest* is shown in Figure 5. There, the call is

$$L \text{ var}, L \text{ proc}(L \text{ var}, L \text{ var}, L \text{ var}, L \text{ var})$$

if all four parameters have a low security level. This is acceptable since the request will be sent over the network and its security level should be low. Assume that we pass *cr* (instead of *encrypt(cr)*) as one of the parameters, and the security level of *cr* (clear text) is high, then the call is (using subtyping relations)

$$H \text{ var}, H \text{ proc}(H \text{ var}, H \text{ var}, H \text{ var}, H \text{ var})$$

Thus the request will be in the high security level, which is unacceptable. Therefore passing clear text as one of the parameters to *SEClassRequest* is a security violation.

5 Conclusion and Future Work

From the analysis in the previous section, we show that the secure type system is powerful, although there are some unsolved problems. It has been shown that the secure flow problem for a typical programming language is undecidable [6]. Therefore any sound and recursive logic for proving that programs have no secure flow violations is incomplete. This partly explains the previous problems mentioned in Section 4 such as randomness and proper classifications of information. Further research is needed to address this issue.

There are other things that the current secure type system doesn't handle. For example, sometimes practical and useful programs need explicitly to lower the security class for specific variables in a well controlled way. We don't know how to handle such cases in the type system. More research needs to be done in this area.

We also rely on Java Cryptographic API and Java cryptographic class library. Since Java is a strongly typed language, the interfaces of the Cherubim bootstrapping should be declared with more specific types, rather than the generic byte arrays.

Overall, we think type system is an efficient way for enforcing secure information flow. With automatic type inference techniques, we can easily apply it to active

networks and mobile computing environment. Such type inference can provide a finer level of dynamic security policy.

References

- [1] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proc. of the 1997 IEEE Symposium on Security and Privacy*, May 1997.
- [2] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corp., 1973.
- [3] G. Brose. JacORB - a Java object request broker. Technical report, Berlin Free University, Apr. 1997.
- [4] R. H. Campbell and T. Qian. Dynamic agent-based security architecture for mobile computers. In *the Second International Conference on Parallel and Distributed Computing and Networks*, Brisbane, Australia, Dec. 1998.
- [5] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236-242, 1976.
- [6] D. E. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504-513, 1977.
- [7] T. Fraser. An object-oriented framework for security policy representation. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Dec. 1996.
- [8] J. Goguen and J. Meseguer. Security polities and security models. In *Proc. of the 1982 IEEE Symposium on Security and Privacy*, pages 11-20, 1982.
- [9] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP-16*, Oct. 1997.
- [10] T. Qian et al. *Cherubim* agent based dynamic security architecture. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1998.
- [11] D. Volpano and C. Irvine. Secure flow typing. *Computer and Security*, 16(2):137-144, 1997.
- [12] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. Theory and Practice of Software Development*, volume of 1214 of *Lecture Notes in Computer Science*, pages 607-621, April 1997.
- [13] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167-187, 1996.
- [14] C. Willis. On the design and implementation of security services for dynamic security systems. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1998.

Dynamic, Distributed, Secure Multicast in Active Networks

Sudha K. Varadarajan*
Microsoft Corp.
Seattle, WA

Tin Qian
Dept. of Computer Science
University of Illinois, Urbana-Champaign

Roy H. Campbell
Dept. of Computer Science
University of Illinois, Urbana-Champaign

Abstract—This paper proposes two frameworks for secure multicast on active networks. The frameworks exploit the computational power of active networks to provide the security desired for multicast, while removing drawbacks in traditional approaches. The main security component in the frameworks is the Active Capability(AC) which replaces the passive session key. The main advantages of using an AC are lack of an asymmetric key pair requirement for authentication, lack of session key modification requirement when a member leaves the group and a highly distributed and scalable key distribution mechanism independent of availability of a group owner.

I. INTRODUCTION

The benefits of multicasting are becoming ever more apparent, and its use much more widespread. While traditional approaches did not scale, CBT[1] and PIM[2] were proposed to address scalability and other papers like [3] introduced newer methods of Key Generation and Distribution. However, these methods still required that the session key be changed when a member leaves the group and most protocols rely on the existence of an expensive asymmetric key infrastructure for secure session key management. The frameworks proposed in this paper attempt to overcome these drawbacks by exploiting the computational power of active networks. Section 2 describes the key components of the underlying security framework. Section 3 describes and analyzes the two models whose trust assumptions vary considerably. Section 4 concludes with a comparison of the proposed models with traditional secure multicast models built on "passive" networks.

II. SECURITY FRAMEWORK COMPONENTS

1) *The Certification Authority(CA)*: The CA is responsible for authenticating individual entities. All applications/routers first register with the CA and obtain an Identity Certificate(IC) from the CA. This IC should be obtained in a secure manner since a malicious eavesdropper should not be able to steal the IC and impersonate as the other user. The IC will be used to authenticate the user whenever required in the frameworks

2) *Active Capability(AC)*: An Active Capability is an active object that carries out security functions for protecting and controlling access to the object(s) it is associated with [4]. In the proposed security framework, permissions are assigned based on roles. A role is associated with a set of permissions and an AC is associated with a role. Hence, if a user has a particular role, it implies that he possesses the AC associated with that role. ACs are critical objects and are fully trusted to provide the user with no more and no less access than his role dictates. Hence, ACs are generated by a fully trusted *Policy Server* which has complete knowledge of role mappings of the system. However, it is impractical to require that every user obtain his AC from a centralized Policy Server. This framework proposes ACs that can spawn themselves and delegate

access. Every AC possesses a delegation count that limits the amount of delegation. An AC also carries the name of the Principal in whom it resides. This prevents a malicious eavesdropper from stealing the AC and trying to utilize it for himself. An AC also possesses a timeout count after which the AC invalidates itself. In addition, the AC also caches the Public Key of the CA to decrypt ICs presented to it. The following illustrates how an AC may be obtained by delegation and the high scalability hence achieved:

→ : ACRequest Capsule
○ : Host with AC
R: Active Router
r: Receiver

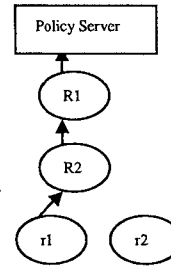


Fig. 2.1. r1 requests PS for AC

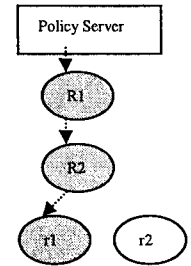


Fig. 2.2. AC delegated downwards

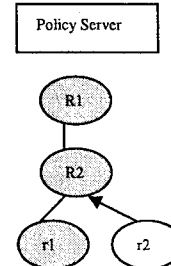


Fig. 2.3. r2 requests for the AC.

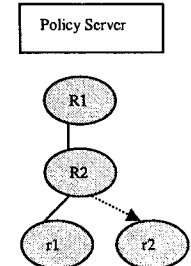


Fig. 2.4. R2 delegates a portion of its AC to r2.

3) *The Multicast AC (MAC)*: The Multicast AC extends the AC class and carries the multicast session key and the following methods as its private data:

- `byte[ ] EncryptMulticastData( byte[ ] data )`: This appends a header to the data, encrypts the header and the data and returns the encrypted byte array. The header contains the following fields: The Principal of this AC (this will authenticate the sender to the receiver), the Role of this AC (which is "Regular Multicast"), the revocation flag (set to false) and the revokee (set to null).
- `byte[ ] DecryptMulticastData( byte[ ] encryptedData )`: This decrypts the encrypted data, and strips the header. If the revocation flag is set to true in the header, it checks if its Owner is the Principal to revoke. If not, it just returns the decrypted data. Else, it revokes itself by timing itself out (sets its timeout count to zero) and returns null.
- `Principal AuthenticateMulticastData( )`: This returns the Principal that identifies the sender of the most recently decrypted data. Hence, one could decrypt the multicast data, and if one desired, call the authenticate method to learn who the sender was.

* This work was supported by DARPA under grant no. F30602-98-1-0192

* Primary Author for contact regarding questions in the paper - sudhakv@microsoft.com

- `byte[ ] RevokeMulticastCapability( byte[ ] data, Principal toRevoke )`: This is identical to `EncryptMulticastData` except that the revocation flag is set to true and the revokee is set to the Principal toRevoke.

Henceforth an AC will refer to the Multicast AC.

III. THE FRAMEWORKS

Assume the following Active Network Configuration:

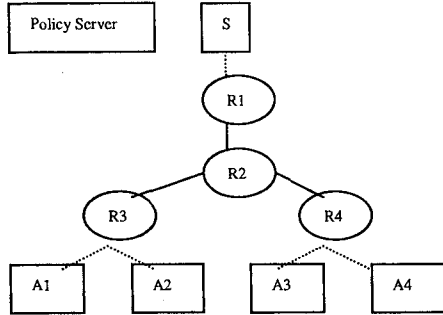


Fig 3.1. An Active Network

R1, R2, R3 and R4 are active routers. S is the sending multicast application while A1 to A4 are receiving multicast applications attached to their respective active routers. Note that the application termed as the sender is none other than the multicast group owner or the Rendezvous Point in PIM [2] or the Core in CBT [1].

A. Framework 1 - Trusted Execution Environment for ACs, Non-trusted Channels - Encrypted ACs.

This section details the first model for secure multicast on Active Networks. It first describes the *BlackBox*, a trusted storehouse and execution environment for an AC followed by a description of the capsules used and the part they play in the protocol. Next, it illustrates the protocol and finally analyzes the framework.

1) *The Blackbox*: Since the MAC carries the session key for multicast, it is imperative that an AC cannot be broken into. Hence an AC is encrypted and transmitted and is decrypted at its destination. This model assumes the presence of a long term key pair (call it the AC key) to encrypt and decrypt ACs. The initial AC is generated by the Policy Server and is encrypted in the Public Key of the AC key. Every AC also knows the Public Key of the AC key. When an AC spawns/delegates a new AC, it encrypts the new AC in its Public Key before passing it to the outside world. This requires that there exist a "BlackBox" that resides in every active node that serves as the home and execution environment for an AC in that node. Every BlackBox knows the long term decryption key (Private key of the AC key) of the AC. When it receives an AC that it is expected to house, it decrypts the AC and stores it within itself. The BlackBox could be either some sort of hardware (say a smart card) or tamper-proof software running in each host. The BlackBox is trusted to not break into the AC and reveal its contents to the outside world. It is trusted to not alter the AC in any manner, but instead just provide a place for the AC to reside and serve as its interface to the outside world. The BlackBox in each active node stores the AC for each application running in that node.

An AC needs to ensure that it is being utilized for the correct application and that one application does not impersonate as another. Every AC carries the name of the Principal it is meant for. Authentication of the Application is done using the Identity Certificate (IC) of the application. Before the BlackBox replaces the AC for an application, it presents the IC of the application to the

AC, which verifies that the Principal in the IC and the Principal it is meant for are the same. If not, the replacement is denied.

2) *The ACRequest Capsule*: This is a capsule that originates at one of the end applications and is sent towards the sender. The sender is assumed to always possess the MAC, and if he does not, then the sender will contact the Policy Server to obtain the AC. The ACRequest Capsule stores within itself the name of the requesting application and the router in which it resides. It also carries the role for which the AC is desired and a vector of requestors. Let us assume that A1 generates an ACRequest Capsule for the role "Regular Multicast", and let us assume that only S and R1 possess the Multicast AC (MAC) at this moment. The ACRequest Capsule will be sent towards S with a request for the AC for the role "Regular Multicast" (refer to the section on Multicast AC). Since the capsule is an active capsule, it will be processed at every intermediate node on its way to the sender. At each such evaluation, the ACRequest capsule will check if the intermediate node possesses the required AC. If it does not, the ACRequest capsule adds the current node to its requestor list and routes itself towards the sender. In our above example, this capsule will generate an ACReplyCapsule at R1 with the requestor Name as "A1 at R3" and the requestors vector will contain { R3, R2 }.

3) *The ACReply Capsule*: This capsule first delegates a Multicast AC for each of the Principals in the requestor vector and the final requesting application. If any of the delegation fails, it sends a new ACRequest Capsule with the requestors vector and the requesting application towards the sender. Once all the ACs have been delegated, the ACReply Capsule retraces its path towards the requesting application. On the way, it is evaluated at each active node. If the active node does not contain a Multicast AC and if the ACReply Capsule is carrying a Multicast AC for this active node, it will pass on the node's AC to the node. Note that the route taken by the ACReply Capsule need not be the same as that taken by the ACRequest Capsule since internal routers need possess the AC only for the sake of distribution of the AC by delegation. So, in our example, it is only necessary for A1 to receive the MAC. However, if R2 and R3 also possess the MAC, it will be easier for A2, A3 and A4 to obtain the MAC.

4) *The Multicast Subscribe and Unsubscribe Capsules*: The Multicast Subscribe and Unsubscribe Capsules are sent by a receiver towards the group owner. These capsules dynamically construct and prune the multicast tree, respectively, when they are evaluated in the active nodes along the path from the receiver to the sender. Refer [5] for a diagrammatic sketch of this process. The Multicast Subscribe Capsule is periodically generated by users who wish to subscribe for Multicast Data and remain subscribed (hence periodic). When an application generates a Multicast Subscribe Capsule, the capsule checks with the local node and verifies that the application possesses the required Regular Multicast AC. If not, it will generate a new ACRequest Capsule on behalf of the application and target it towards the sender.

5) *The Protocol*: The protocol is illustrated below:

Joining the Multicast Group and Obtaining an AC: The Sender requests for and obtains an encrypted MAC. He passes it onto the BlackBox in his node, along with his IC. The BlackBox decrypts the AC and passes the IC to the AC. The AC verifies if the Principal it is meant for is the Principal represented in the IC and if so it notifies the Blackbox which will then store this AC for the Principal. The sender then delegates a portion of his AC to his node R1. At the end

of this step, S and R1 possess Multicast ACs as depicted in figures 3.2 and 3.3. It should be noted that while S will use the AC to encrypt and decrypt multicast data, R1 will use his AC to only spawn new capabilities.

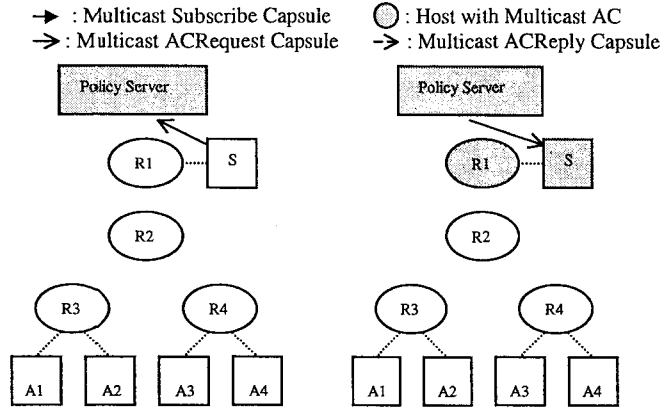


Fig. 3.2

Fig. 3.3

Application A1 wishes to subscribe to the multicast group and sends a multicast subscribe capsule towards the sender. As a result of this, the Multicast Tree gets dynamically formed. Also, the Multicast Subscribe capsule, when evaluated in A1's node(R3), will check if A1 has the MAC. Since it does not, a new ACRequest Capsule will be generated and sent towards the sender. The ACRequest Capsule will reach R1 who has the required AC. R1 will delegate AC's for R2, R3 and A1 and send them across through the ACReply Capsule. The ACs will be handed over to the appropriate nodes when the ACReply Capsule is evaluated in each node. The nodes will hand over the ACs to their BlackBoxes where the ACs will be decrypted, deserialized and stored. Figures 3.4 and 3.5 illustrate this. Figures 3.6 and 3.7 are self explanatory.

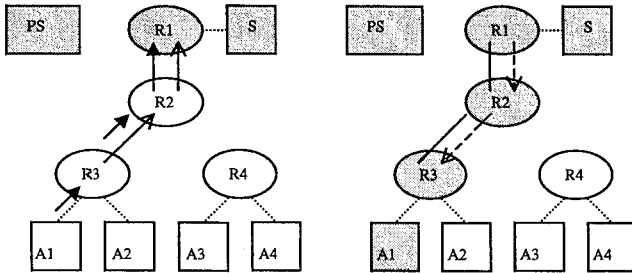


Fig. 3.4

Fig. 3.5

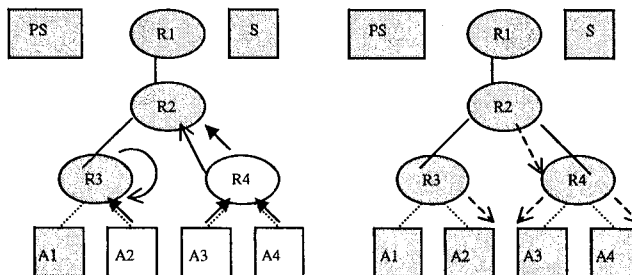


Fig. 3.6. A2, A3, A4 subscribe to the group.

Fig. 3.7. Multicast tree is formed and they receive their ACs.

Sending and Receiving Multicast Data: The sender will request (through his node's black box) his MAC to encrypt the message he wishes to multicast. The MAC will append a header with

authentication information to the data, encrypt them and return the encrypted contents. The sender can then multicast the encrypted contents to the group. At the receiving end, each receiver application, through his node's BlackBox will request his MAC to decrypt the message. The MAC will decrypt the data, strip the header away and return the decrypted payload. If the receiving application wished to authenticate the sender, he could call the "AuthenticateMulticastData" method of the MAC.

Leaving the Multicast Group – Revoking a user's AC: The sender may choose to revoke an Application's (say A1) AC either on A1's request or of his own accord. In this case, he would call the RevokeMulticastCapability method of the MAC with A1 as the Principal to revoke. This method returns a regular Multicast Capsule, with some extra information in the encrypted header (Refer description of MAC above). The sender can now multicast this capsule. When A1 sends the encrypted data to its AC for decryption, the AC will notice the revocation flag set to true for its owner A1, authenticate the revoker and will revoke itself by timing itself out. All other applications' ACs will ignore this flag.

6) Analysis of the model: This framework is mixture of both DAC (Discretionary Access Control) and RBAC (Role Based Access Control). This is because an AC grants role based privileges, but contains the Principal it is meant for and so is user-specific. The reason it contains the Principal is to prevent theft. Assume that users A and B both have the "Special Multicast" role. The Special Multicast role allows a user to only view 10 packets of special data (something like a pre-paid view). User A has already viewed 5 special packets. Now, user B is about to get a Special AC, with a fresh count of 10. If the AC did not know it was meant for B and not A, A could very well eavesdrop on the channel, steal the AC and present it to his BlackBox as his own. The installation will be allowed by RBAC since user A has the Special Multicast role.

Trust Assumptions made by this model:

The Active Node is trusted to not reveal one application's IC to another application.

When an application requires some functionality from its AC (say encrypt/decrypt data), it will have to contact the appropriate API of the BlackBox. Since only the node has a handle on its BlackBox, the application will have to use the APIs of the active node to communicate with the BlackBox. The active node is trusted to not redirect such communication to a malicious application, or modify the communication in any way, instead just be a direct intermediary.

Only the BlackBox sees a decrypted AC and is trusted to neither reveal the contents of nor modify the AC.

Some Other Issues:

To verify authenticity of AC: An AC can only be created anew by the Policy Server. All other AC's can only be spawned from existing ACs. Moreover, ACs are encrypted and transmitted and are decrypted only in the Black Boxes. Since the AC that is created by the Policy Server is fully trustworthy and is never modified to perform other than how it is supposed to, there is no need for this.

Source Integrity: By data encryption

Authentication of sender: Achieved by generating an AC for a particular Principal (the Owner of the AC) and verifying that only the Owner uses the AC. The verification is done using the Principal's IC and the CA's Public Key.

Non repudiation of message: The assumption in this model is that a Principal initially receives an IC from the CA in a secure manner and that an active node never reveals the IC of one Principal to another. Also, once obtained, an IC is just used locally in the node and is presented only to the BlackBox on the active node. Thus ICs

cannot be stolen and so an AC of a particular Principal can only be used by that Principal. When a user asks his AC to encrypt a message before relaying it, the AC appends the Principal to the message. Since the AC is completely trustworthy, non repudiation of messages is achieved.

Prevent Spamming: Before a user attempts to send (or flood) blank Multicast Data Capsules to a multicast group, the data capsule on its first evaluation (in the application's active node) should verify with the BlackBox (through the node) if the user contains the MAC. This will verify the user as an authorized subscriber to the group. If the user does not contain the said AC, the Multicast Data Capsule can discard itself.

Advantages of the model:

It provides a framework for secure multicast where the session key management framework is distributed and the model is scalable.

Unlike most other models, group leave does not involve changing the session key and is computationally inexpensive. The original session key may be used for multicast for as long as necessary.

Individual entities do not require an Public/Private Key Pair to receive a session key. They only require a non-repudiable IC generated by a trusted authority to identify the Principal.

The Session Key is a symmetric key. There are only two asymmetric key pairs in the entire framework. One is that of the CA. Its Public Key is used to verify the signature in an IC and is cached in every AC. The other is that of the AC. The Encryption Key is stored in the Policy Server and the ACs, while the Decryption Key is stored in all the Black Boxes. Note that both keys are long-term keys. The AC key is used only during the time of obtaining and delegating ACs. The session key is used to encrypt/decrypt all multicast data. Hence, encryption and decryption costs in the framework are relatively low.

Drawbacks and Alternatives:

i) Changing the long term keys is a problem. It may either have to be done manually, or there should be some other secure protocol (for example, by using asymmetric key pairs for each individual entity) to do this. This framework will be ideal for a closed system, say a cable company or a corporation, that can periodically (say once a year) change the keys in every node's BlackBox.

Alternative 1 - Sender (Policy Server) initiated: Manual change of the key in the BlackBox may neither be feasible nor acceptable. Instead, the BlackBox could allow the AC decryption key be modified by a special AC. This special AC should be generated by the PolicyServer, encrypted in the old AC key, signed by the PolicyServer and broadcast to every Active Node in its domain. The Active Node will pass on the AC to the BlackBox for decryption, where it is activated. This AC has to be signed by the PolicyServer for double protection. One problem in this approach is a BlackBox obtaining the AC decryption key the very first time. It could either be through a trusted boot-strapping routine or alternative 2. Another problem is what happens if the PolicyServer broadcasts a change while an active node is down? If the other nodes commit the change, the BlackBox in this active node will once again require a trusted method to obtain the new key.

Alternative 2 - Receiver (BlackBox) initiated:

- The BlackBox stores the AC decryption key and a time for which the key is valid.
- The BlackBox does not store the AC decryption key, but instead stores a special AC that stores this key. This AC timesout periodically.

In either case, after the timeout period, the BlackBox has to refresh the key from the Policy Server by making a request to it. The communication between the BlackBox and the PolicyServer must be

secure. One way is to require that each BlackBox have an asymmetric key pair. The Policy Server gives the special AC, encrypted in the BlackBox's public key, to the Black Box. The other way is to use Kerberos or Sesame that generates a session key that may be used to encrypt and transfer the new key value. Obviously both alternatives 1 and 2 have their pros and cons. Alternative 1 requires an additional asymmetric key pair for the PolicyServer. It is cheaper since the special AC is broadcast. However, the drawback lies in obtaining the AC key the very first time and may also require a two-phase protocol to change the AC key. Alternative 2 on the other hand has much more expensive requirements and is a costlier process, counting the number of messages one would have to exchange to make sure all BlackBoxes have the new key. However, it does not have the disadvantages of Alternative 1.

ii) If one of the BlackBoxes is compromised, the entire framework fails. The assumption is that the BlackBox cannot be compromised. It must be noted that this is true in traditional approaches to secure multicast, too. If a single client is compromised and the multicast session key revealed, then the entire multicast communication fails and has to be started afresh with new initial values.

Alternative: An alternative approach will be to have an asymmetric key pair for the entire network and a centralized directory (say per domain) where the public keys are stored. In this case, the AC, when delegated, should be encrypted in the receiver's public key. Note that the centralized directory is a bottleneck. Also, it is far more expensive to maintain asymmetric key pairs for every node in the network. Moreover, obtaining the public key from the centralized directory should be a secure process.

B. Framework II - Trusted Active Nodes and Trusted Channels between Active Nodes - Non Encrypted ACs

This framework addresses the issue of requiring a Black Box and a long term key with which to encrypt and decrypt ACs.

1) *The Model:* The multicast protocol does not change in this framework. The section uses the example in framework I (fig 3.2) to describe a model of a multicast in which the routers and channels are trusted. In this framework, R1, R2, R3 and R4 are trusted to not reveal or modify the ACs that they acquire for the applications attached to them. Also, the channels between R1, R2, R3 and R4 are secure channels that cannot be eavesdropped upon. In this case, everything would be the same as in framework I, except that ACs need never be encrypted and transmitted. Also, the BlackBox is just a dummy storehouse of ACs in this model. Note that the channel between the end applications and their routers is not trusted and it is because of this channel that multicast data is encrypted. The BlackBox itself resides in the Active Node or Router and not in the receiving applications or clients.

2) *Analysis of the model:*

Trust Assumptions made by this model:

The Active Node is trusted to not reveal one application's IC to another application.

When an application requires some functionality from its AC (say encrypt/decrypt data), it will have to contact the appropriate API of the BlackBox. Since only the node has a handle on its BlackBox, the application will have to use the APIs of the active node to communicate with the BlackBox. The active node is trusted to not redirect such communication to a malicious application, or modify the communication in any way, instead just be a direct intermediary.

ACs are never encrypted. The Active Nodes are trusted to neither reveal the contents of the AC nor modify the AC. The channels between active nodes are trusted to be secure.

Some Other Issues:

To verify authenticity of AC: There is no need for this as explained in framework I (note that the channels are trusted here and AC encryption /decryption is unnecessary).

Source Integrity, Authentication of sender, Non Repudiation of message, Spamming: As in framework I.

Advantages of the model:

All the advantages of framework I are applicable here. Moreover, there is only one asymmetric key pair in this framework - that of the CA, whose public key is used to verify the signature in an IC.

It does not require a long term key pair to encrypt/decrypt ACs. This reduces computational and maintenance costs of the protocol.

Drawbacks:

The Active Nodes and their channels must be totally trustworthy. This may be useful in the Internet where backbone routers and gateways may be trusted. Also, the hardware channels between the backbone routers are most likely dedicated and can be made secure.

IV. CONCLUSIONS

This paper provides an innovative approach to the problem of dynamic, secure multicast. While traditional approaches also achieve secure multicast, they are extremely restricted in their protocol since they cannot delegate control (to handle multicast session events like join and leave) in a trust-worthy manner. The frameworks proposed in this paper exploit the computational power of active networks and deploy Active Capabilities that carry out distributed trust management. As far as the author knows, the frameworks proposed in this thesis are unique in utilizing the computation power given by active nets to solve the problem of dynamic secure multicast in a way that surpasses traditional schemes. The following lists out the key advantages of the proposed frameworks over traditional approaches.

1) *Session Key Management:* This is the key area in which the proposed frameworks are far superior.

In traditional approaches, there exists a centralized "group owner" from whom the multicast session key is obtained. This makes the group owner a "hot spot". In the proposed frameworks, the multicast session key is stored in Active Capabilities that are cached by routers in the dynamic multicast tree. Hence, a client that subscribes to the group and needs to obtain the session key (or the AC containing it), makes a request to its parent router and gets it. There is no centralized key distributor. Key distribution is distributed.

In traditional approaches, the centralized group owner is trusted to pass on a multicast key to a valid subscriber. Similarly, in the proposed frameworks, ACs are trusted to delegate ACs to valid subscribers. The amount of trust is equivalent in both approaches.

In traditional approaches, leaving a group involved two parts: a) changing the multicast session key, and b) transmitting the new key to all other valid members of the group. Part a involved generating a new session key and in most approaches, part b was solved by unicasting the new key securely (using another pair of keys) to every other member. The proposed frameworks do not require a change in the group key when a member leaves the group. Instead, the leaving member's AC is revoked. This approach is possible due to the computational power of active nets. The greatest advantages here are that i) by removing part a, the computational complexity on the group owner reduces enormously if group join/leave is a frequent event, and ii) part b is replaced by a single revocation of the leaving member - a large savings on the bandwidth and computation.

2) *Improved Security with lesser trust:* In the traditional approach, decrypted multicast data may be revealed by subscribed receivers to un-subscribed hosts. Then, the un-subscribed host is dependent on the subscribed host to receive the data. Also, the subscribed host knows the multicast session key which he can reveal to the un-subscribed host. In this case, the unsubscribed host can view the multicast data independently, for the lifetime of the session key.

In the proposed frameworks, the receivers do not know the session key and so cannot reveal the session key to anybody else. This, by far, reduces the trust on the receivers. Also, in the active network approach, the original multicast data capsule can contain a digital watermark of the originator of the message. If a subscriber tries to broadcast/unicast this data to an un-subscribed host(s), the packet will have to go through his router. In Framework II, the routers are trusted and their functionality can be extended to check if the watermark on the message identifies the sending client or not. Based on this, the packet could be forwarded or dropped.

3) *Greater Availability and Anti Denial of Service Attacks:* In traditional approaches, if the group-owner gets temporarily disconnected from the multicast tree, the entire multicast halts since no new members can join the group while existing members cannot effectively leave the group. In the proposed approaches, even if the group owner is temporarily disconnected, multicast join and leave can proceed provided other routers (along the path from the member to the group owner) possess the Multicast AC.

Since the MAC is distributed along the multicast tree, a denial of service hack is not possible. In traditional approaches, a hacker can disconnect the group owner from the rest of the tree, while this won't be of much consequence in the proposed approaches.

4) *Simpler Authentication:* In traditional approaches, clients were required to possess a public/private key pair for authentication. A message would be signed in the sender's private key and his public key would be used to verify the message at the receiving end. This involves an additional encryption and decryption for every message sent and received, over the usual data encryption. The proposed models do not require this since ACs are trusted. When a client wants to encrypt a message before multicasting it, he requests his AC to do this. The AC appends its Principal's identity which it has originally verified (a one time decryption) using the client's IC, encrypts the entire payload and this is then multicast. At the receiving end, the client's AC decrypts the entire payload, strips the header and reveals the Sending Principal and the original message.

5) *Trust Assumptions:* For all the above advantages to be gained, the AC cannot/should not be compromised. This is the same amount of trust in traditional networks where if the session key is compromised, the secure multicast fails.

Future Work: Given the structure of today's networks, the information dissemination would be highly improved by a secure multicast protocol. As discussed in the earlier chapters, framework I is appropriate for networks where hardware channels and nodes cannot be trusted. However, framework II is easily applicable to backbone networks where one can trust the nodes and channels. A hybrid protocol could combine both the frameworks and produce an efficient distributed secure multicast protocol that reduces both computational and maintenance costs.

REFERENCES

- [1] A Ballardie. Scalable Multicast Key Distribution. RFC 1949, May 1996.
- [2] D. Estrin, D. Farinacci, et al. Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification. IETF draft, March 1997.

- [3] Yang-hua Chu, Adrian Perrig and Dawn Song. SMIF: A Framework for Secure Multicast Intercommunication. <http://gs193.sp.cs.cmu.edu:8002/yhchu/research/secure-multicast.html>, unpublished.
- [4] Roy H Campbell, Tin Qian, Willy Liao and Zhaoyu Liu. AC: A Unified Security Model for Supporting Mobile, Dynamic and Application Specific Delegation. White Paper, February 1996.
- [5] Sudha K. Varadarajan. Dynamic, Distributed, Secure Multicast on Active Networks. Master's Thesis, Jul 1999.

Securing the Node of an Active Network*

Zhaoyu Liu, Roy H. Campbell, M. Dennis Mickunas
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{zhaoyu, roy, mickunas}@cs.uiuc.edu

Abstract

Active networks aim to provide a software framework that enables network applications to customize the processing of their communications. Security is of critical importance to the success of active networking. This paper discusses the design of securing the node of an active network using active networking principles. The secure node architecture includes an Active Node Operating System Security API, an Active Security Guardian, and Quality of Protection (QoP) provisions. The architecture supports highly customized and situational policies created by users and applications dynamically. It permits active nodes to satisfy the application specific dynamic security and protection requirements. It aids the application of the “need-to-know” security principle and associates quality of protection with network software and application security. The secure node architecture can provide fundamental base for securing the active network infrastructure.

Keywords: *active networks, security API, active capability, active security guardian, quality of protection*

1 Introduction

Active networks aim to provide a software framework that enables network applications to customize the processing of their communications. The current active network research focuses on the support of flexible, dynamically changing, fine-grained quality of service. There is little research on dynamic, flexible, and application specific security features that exploit active networking. Similar to traditional networks, active networks rely heavily on the underlying operating system for network security. Current active network operating systems do not have explicit security support and applications can not flexibly request security and protection requirements. The inflexibility of the systems makes security policy and service customization complex and often leads to security holes.

In this paper we present the design of securing the node of an active network using active networking principles. We term this approach active security. The secure node architecture is integrated into the active node operating system and includes:

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

- A node operating system security API
- An active security guardian
- Quality of protection (QoP) provisions

The secure node architecture supports highly customized and situational policies created by users and applications dynamically and provides fundamental base for securing the active network infrastructure.

The rest of the paper is organized as follows. Section 2 discusses the current security research on active networks. Section 3 describes the architecture of a secure active node in detail. It discusses the principles to design the node operating system security API, and describes the design of active security guardian and the support of quality of protection. Section 4 presents the current implementation and the future work of this active security research and then the final section concludes the paper.

2 Related Work

This section surveys the current security research on active networks. It provides background and motivation for the secure node architecture presented in the next section.

2.1 Active Network Security

It is difficult and complicated to retrofit security into Internet infrastructure [22]. The active network research community considers security as an important part of the initial design. The security working group [23] of the active networks research community has been instrumental in publicizing and highlighting the importance of security in active networks. The group emphasizes the importance of incorporating security into the initial design stage of the active network architecture itself. The current security related research in this field can be classified into two general categories. The first one deals with the more traditional notion of security, which includes authentication, access control, policies and enforcement. The security working group [23] has launched some important exploratory research in this direction. The second category is mostly about protection of nodes from mobile code originating in foreign domains and protection of active packets or code from malicious hosts [32]. The PLANet effort [1] raises some of the issues associated with these protections. In addition the effort also provides a bootstrapping module that ensures that the system configures itself correctly at startup or reboot time. The protection from mobile code is provided by using a type-safe, resource limited, functional programming language with dynamic type verification. Mobile code can install protocols at nodes securely by using the extensibility features provided by the language. Naccio of MIT [10] also belongs to this category. The high-level application specified policies limit Java mobile code capability and thus provide the necessary protection to mobile code execution host.

2.2 Active Node Operating Systems

The high-level architecture for active node is shown in Figure 1 [5]. A node operating system (NodeOS) manages the resources such as memory regions, CPU cycles

and link bandwidth, and multiplexes packets among multiple execution environments (EEs) running on the node. In order to support the porting EEs to multiple underlying NodeOSes, a NodeOS interface is specified by the NodeOS working group [28].

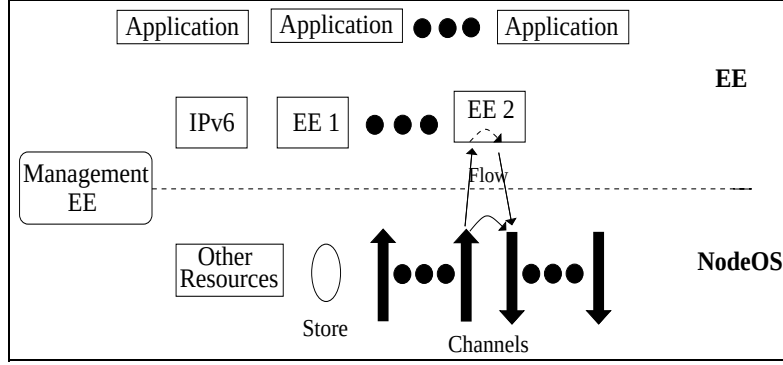


Figure 1: Active Network Node Architecture

The objectives of current NodeOS interface are to support fast network packets forwarding and fine-grained quality of service. The interface doesn't explicitly specify any security API. It defines the following five primary abstractions of system resources:

- Thread Pool: computation resource.
- Memory Pool: memory resource
- Channel: communication resource, including not only network bandwidth, but also CPU cycles and memory space.
- File System: persistent storage resource.
- Flow: Generally speaking, a flow is a sequence of packets satisfying some pre-defined attributes of interests. Typically flows are related to routing [27] and quality of service [37], where groups of packets would receive similar treatment in their network transport. Traditionally the flow concept can be used in both datagram and connection-oriented communications. In active networks, the flow concept is used to aggregate control and scheduling of the above four abstractions. It provides abstraction for accounting, admission control and scheduling in the system. A flow can contain sub-flows and this results a hierarchical flow structure.

Currently there are several NodeOS implementations in active networks research community. They all comply to the general NodeOS interface specifications in various degree:

- **Joust:** Joust [11] is a small, fast JavaOS implemented in Scout [21]. It includes an efficient Java virtual machine and a Java JIT compiler. It explores how Java's various features interact with Scout's modular approach to building systems.

The current NodeOS interface for active network nodes is mostly based on the experiences with Joust.

- **Janos:** Janos is a Java-oriented active network operating system [4]. Its objective is to develop a principled local operating system for active network nodes, which is oriented to executing untrusted Java byte code. The primary security focus is resource management and control, with secondary objective of other information security, performance, and technology transfer of broadly and separately useful software components. Janos interface provides a sample Java binding of the NodeOS API abstractions.
- **AMP System:** AMP's NodeOS is based on Exokernel operating system [12], and uses Exokernel's hierarchically-named capabilities [19] to support flexible access control. Each Exokernel environment (similar to a Unix process) holds a number of hierarchically-named capabilities, known as CAPs. The kernel maintains an array of CAPs and the environment specifies which CAP to use for each system call or IPC operation.
- **Bowman:** The Bowman node operating system is built to support the CANEs EE. It is designed around three key abstractions: channel, a-flow, and state-store [20]. A channel is the primary abstraction for communication and an a-flow is the primary abstraction for computation. The state-store provides a mechanism for a-flows to store and retrieve state that is indexed by a unique key. The Bowman NodeOS interface can be extended to provide support for additional abstractions such as queues, routing tables, user protocols and services.

In summary, the current active node operating systems research focuses on high performance, extensibility, and resource management. There is little research on explicit security support for authentication, authorization, integrity, and dynamic access control. The secure node architecture presented in the next section addresses the above security problems in active networks. It is complementary to the current NodeOS research and augments its functionality. It can be seamlessly integrated into the current NodeOS implementations to provide dynamic security services and access control.

3 Securing the Node

The architecture of the secure node of an active network is shown in Figure 2. The secure node architecture includes a NodeOS Security API, an Active Security Guardian, and Quality of Protection (QoP) provisions. The API provides support of authentication, authorization, integrity and access control services to EEs and active applications. It is implemented by a security library. An *Active capability* (AC) [18, 8, 7] is used to support flexible distributed dynamic security policies. Essentially an AC is an executable Java code which concisely represents dynamic security policies and mechanisms. The security guardian evaluates ACs in a secure sandbox environment and enforces the security requirements of AC evaluation results. It obtains ACs securely through the AC communication protocol. By using the NodeOS security API, active security guardian, and active capabilities, it is feasible to provide quality of protection to active applications.

The rest of the section is organized as follows. We first briefly describe the active capability, Role Based Access Control policy type and active security guardian concepts. These concepts are developed and used in the Cherubim and Seraphim projects [7, 18, 6]. Then we present the NodeOS security API and quality of protection provisions in detail. Finally we discuss the low-level code safety and the EE security.

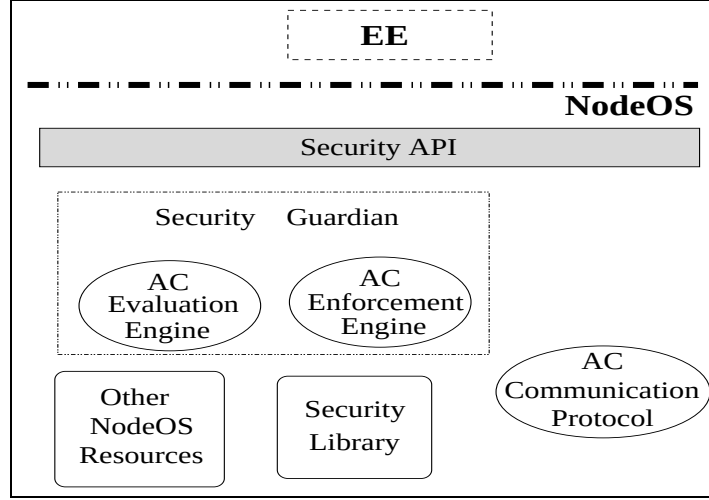


Figure 2: Secure Node Structure

3.1 Active Capability

Active capabilities are used to support flexible distributed dynamic security policies and services control, based on the similar active principles employed by active networks [18, 7]. Unlike a traditional capability, which is merely a static authorization credential that encodes the principal and the permissions associated with the principal, an active capability is a customized piece of code that encodes the type of access control policy and other constraints used in the access control decision making process. In our implementation, an AC is an executable Java code which concisely represents dynamic security policies and mechanisms. In addition, an active capability is protected by digital signatures, resides in user space and can be freely passed around.

By using an active capability various situational policies that depend on system attributes can be encoded. For instance, by writing a piece of code that checks the current system time and compares it with a value stored in the active capability one can introduce a policy that expires after a certain time deadline. Similarly, various enforcement and revocation schemes based on other attributes like quota, history, and information content can be implemented. These schemes are very useful in an open internetworking environment with diverse application requirements. An application can use quota-based revocation to limit the amount of system resources a client can consume.

An active capability could carry all policy information of decision in its code. This heavy way of implementation is not elegant and efficient. A better way is to have a

generic policy framework to support different various policy types and ACs rely on it for context. An application presents an active capability along with its regular data or protocol capsules to the active router's security guardian at execution time. The access control policy type and user credentials are extracted from the capability. The remote router's security guardian recreates the context of the policy type within its policy framework. If at any point during this process, the policy framework discovers that it does not have an implementation for the type of the policy, it downloads the code dynamically into the framework, using the underlying active network. It then instantiates the run-time parameters associated with the application in its sandbox-like environment and executes the active capability in this environment. Based on the result of the evaluation of this active capability, the access control decision is enforced.

The principal of the active capability, which can be a user, a role, or other principal, must be authenticated by a trusted authority. The trusted authority acts as the policy server in our system. This entity is responsible for generating and keeping track of the active capabilities. Usually one or more policy servers are associated with each protection domain. Application programs contact their nearest or least-loaded server and obtain the active capability dynamically.

3.2 Role Based Access Control (RBAC) Policy

The policy type used for dynamic access control in the architecture is Role Based Access Control (RBAC) policy type, which is the most flexible type of access control policy [33]. A Role Based Access Control policy, as the name suggests, uses the concept of a role as its basis for representing permissions [33]. It is a form of access control that emerges in the context of security policies for organizations. A role is chiefly a semantic construct that forms the basis for an access control policy. With RBAC, system administrators create roles according to the job functions performed in an organization, grant permissions to those roles, and then assign users to the roles on the basis of their specific job responsibilities and qualifications. The idea is that the particular combination of users and permissions brought together by a role tends to change over time while the permissions associated with a role are themselves relatively more stable.

The biggest advantage that RBAC has over other forms of access control is that it is extremely intuitive to use and maps easily to real-world situations. A hierarchy of roles with senior roles inheriting all the permissions of junior roles closely follows the structure of organizations. The access control policy in RBAC is embodied in components such as role-permission, user-role and role-role relationships. These components collectively determine whether a particular user is allowed access to a particular operation on a particular component. These individual components can be easily (and intuitively) configured to provide the required degree of access control. For example, adding a new user to a system would merely involve assigning appropriate roles to the user according to the user's functions in the organization. Likewise, changing the nature of, for example, printer access, for all managers in an organization can be accomplished by merely changing the permissions with the manager role in the organization. All managers can immediately see the effects of the change.

RBAC is the most flexible type of access control policy. 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. The RBAC constraints supported in our system include three important ones: mutually exclusive roles/permissions, prerequisite roles/permissions and cardinality constraints.

3.3 Active Security Guardian

The security guardian in the architecture is to support AC evaluation and enforcement. All accesses to node resources must go through security guardian which use the security library services to verify the signature on the active capability.

The security guardian's functionality is similar to traditional reference monitor, with several major differences. In traditional systems, a reference monitor is interposed between the subjects and objects to control subjects' access to objects based on access authorizations (Figure 3). The traditional reference monitor is passive in the sense that it never initiates actions but only reacts when it receives an operation message. Access through the reference monitor is either granted or denied corresponding to a yes or no access evaluation result. The power and functionality of the traditional passive reference monitor are limited [2].

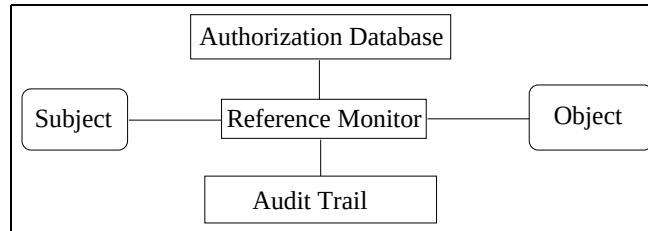


Figure 3: Reference Monitor Concept

With the use of ACs, the security guardian is no longer passive. To make it active, first we need to extend ACs. In addition to access control decision information, ACs may carry other security information. For example, an AC may specify a particular encryption key length for a particular region or country together with access control information. To carry out the the intended security operations specified by ACs, an evaluation engine and an enforcement engine are included in the security guardian. The evaluation engine evaluates ACs in a secure sandbox. The enforcement engine interacts with other NodeOS components to enforce faithfully the security operations, using the security library services. The enforcement engine can initiate security actions based on ACs requirements. So the security guardian may trigger or initiate security actions. The triggers can be intrusion detection alarms, or explicit requests by EEs or applications that use active networking features. For example, the security guardian can initiate installing firewalls dynamically.

3.4 NodeOS Security API

As mentioned above, the current NodeOS API [28] focuses on fast network packet-forwarding and fine-grained quality of service. It provides mainly an interface for resource management without explicit security support. As a complement, the NodeOS security API is designed to provide explicit security support to EEs and active applications. It exports security services including authentication, authorization and integrity to EEs and active applications. The security API is defined as generically as possible to accommodate a wide variety of implementations.

A standard, generic security API promotes easy, widespread development and use of secure applications utilizing security. It allows combinations of cryptographic security that support a range of protection levels. The API and different protection levels support the needs of secure international software applications utilizing cryptography, factoring law enforcement and national security interests. They enable flexible, low-cost methods for cryptographically protecting sensitive information.

An API should satisfy the needs of both simple and sophisticated applications and should be easy to use. It should require applications to have a minimal degree of cryptographic awareness. According to NSA [34], there are several considerations for security API design:

- Algorithm Independence
- Application Independence
- Cryptomodule Independence
- Degree of Security Awareness
- Modular Design and Auxiliary Services
- Safe programming
- Security Perimeter

We advocate a NodeOS security API that is generic and compatible with available security API standards. Currently several related high-level APIs are available in the research community:

1. Generic Security Service API (GSS API): The GSS API is designed specifically for network communication protocols and provides additional support for securing network communications after authentication [15]. It provides protection for communication using authentication, integrity, and/or confidentiality security services. Its extensions support access control and delegation [26].
2. Pluggable Authentication Module API (PAM API): This supports pluggable authentication in stand-alone, non-connection-oriented environments for users and provides system level authentication service [31]. It also provides a uniform interface for authentication that is compatible with many authentication provisions, and thus provides complementary functionality to the GSS API. The Java Authentication and Authorization System API (JAAS API) bases its authentication on the PAM API in the Java language environment [14].

3. Generic Authorization and Access Control Application Program Interface (GAA API): The GAA API supports authorization decisions for applications in a distributed environment [30, 29]. An application invokes the GAA API functions to determine if a requested operation or set of operations is authorized or if additional checks are necessary. An application can also use the GAA API to request access control information about a particular resource. The GAA API can be used to obtain a principal's access rights on an object or a resource and supports the needs of most applications. Developers don't need to design their own authorization mechanisms.

The NodeOS API combines the above APIs to support authentication, authorization, integrity, and access control. A security library implements the NodeOS security API. The API is based on the active network flow concept and supports end-to-end security, hop-to-hop security, and the active network protocols including routing protocols.

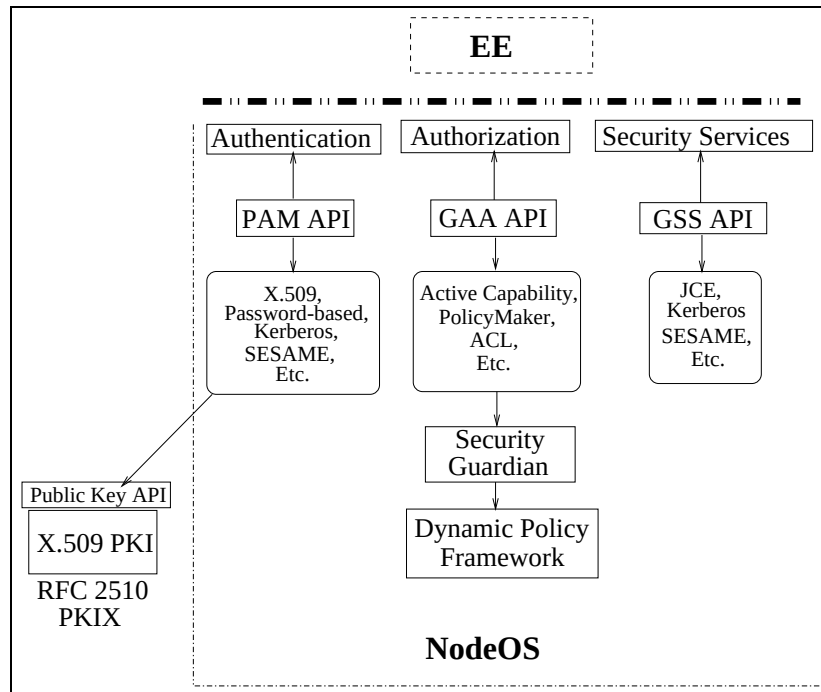


Figure 4: NodeOS API Design

The NodeOS security API has three major components as shown in Figure 4, the authentication API, the authorization API, and the security services API:

- The authentication API authenticates EEs, AAs, or users. It is based on the PAM API. As shown in Figure 4, a possible implementation of the authentication API uses the X.509 public key infrastructure (PKIX). RFC 2510, the Internet X.509 Public Key Infrastructure Certificate Management Protocol, provides a detailed description of the security functions supported by PKIX.

- The authorization API helps protect NodeOS resources. It is based on the GAA API. The security guardian in the Figure 4 supports access-control policy evaluation and enforcement. The security guardian's functionality is similar to a traditional reference monitor or to the role of the checking software that is invoked when a user process requests a supervisor privilege in a traditional operating system like UNIX. All accesses to node resources must go through security guardian. One possible implementation of an access control mechanism is the active capability described previously.
- The security services API provides security services such as encryption and digital signatures. The security services API is based on the GSS API.

Our focus is to export core and essential security functionality to the EEs and active applications while securing the active network infrastructure. Thus, the EEs, the active applications, and the NodeOS itself can use this API for security services, for example, to support hop-hop authentication and security. The implementation of the API must be secure if key management and principal identification are to be secure and thus we locate the implementation of the API within the NodeOS and below the security guardian to take advantage of any hardware protection available to the NodeOS implementation.

The NodeOS Security API we have described is comprehensive but not exhaustive. It can be extended easily for future security enhancements. For example, it can be extended to include the IDUP-GSS-API later, if necessary. The IDUP-GSS-API, Independent Data Unit Protection Generic Security Service API, is similar to GSS API, but is designed for independent data unit protection [3]. It extends the GSS API for applications requiring protection of a generic data unit (such as a file or message). The protection of one data unit is independent of the protection of any other data unit and independent of any concurrent contact with designated receivers of the data unit.

3.5 Quality of Protection

By using the NodeOS security API, active security guardian, and active networking features, it is feasible to provide quality of protection to active applications. Similar to QoS, QoP supports customized, flexible security and protection requirements of applications. For example, applications can specify routing paths based on security and protection requirements.

To provide quality of protection, the NodeOS API needs to be enhanced with different security and protection options. These options are supported by the underlying security library implementation in the NodeOS. In addition the security and protection features need to be characterized. Some sample QoP characteristics include:

- Key length of security algorithms
- Robustness or strength of security algorithms
- Security mechanisms for authentication and privacy
- Trust values for developers/vendors of security implementations: One may trust more the implementation of security algorithms by reputable vendors.

- Assurance level of a router NodeOS: The orange book defines the assurance class for an operating system as D, C1, C2, B1, B2, B3 or A1 [9]. A router NodeOS with higher assurance class is more trustworthy.
- Geographical location of routers: One country may not trust the protection provided by the routers in enemy countries.

Active capabilities are used to specify, control and manage QoP. A trust party creates ACs upon the requests of applications.

With a NodeOS Security API, an Active Security Guardian, and Quality of Protection (QoP) provisions, the secure node can provide active security features to applications. Applications of active security include a security-customized routing path specified by an application and stronger protection under intrusion. For quality of service applications, both time constraints and security features are important [24]. The QoP allows dynamic reconfiguration and tradeoffs between security protection and satisfaction of the QoS constraints. The protection may be provided on per-service, per-flow, or per-capsule base to optimize performance overhead.

3.6 Low-level Code Safety

The evaluation engine of security guardian relies on Java language for low-level code safety. The minimum requirements for low-level code safety are control flow safety, memory safety, and stack safety [13]. Currently we use the Java byte code verifier [36] provided by Java language for low-level code safety. Before loading a class, the verifier performs data-flow analysis on the class code to verify that it is type safe and that all control-flow instructions jump to valid locations [17].

There are several other approaches for low-level code safety. The PLAN project [1] uses programming language techniques to address the code safety problem. Capsules are written using a strongly typed, resource limited language and dynamic code extensions are secured by using type safety and other mechanisms. Another approach is Proof-Carrying Code (PCC) [25]. Besides regular program code, PCC carries a proof that the program satisfies certain properties. The proof is verified before the execution of the code. The generation of a proof may be complex and time consuming, while its verification should be simple and efficient. Software fault isolation (SFI) [35] provides another alternative for low-level code safety. It uses special code transformations and bit masks to ensure that memory operations and jumps access only the correct memory ranges.

In summary, there are a variety of different mechanisms and protocols proposed. Each method has its own advantages and disadvantages. Ultimately the application must be given the choice to pick the mechanism that is most suitable for its purpose. The secure node architecture is generic enough to allow all these mechanisms to co-exist.

3.7 Discussion

An execution environment can also implement security requirements within itself [23]. It can set up security policies for active applications running inside it. While this paper

does not focus on the EE security, the same design principles discussed in this paper can be applied to the EE security.

4 Current Status and Future Work

We have a prototype implementation of secure node architecture, with a simplified version of security guardian. The security guardian is used in the Seraphim architecture framework [18, 6]. The security guardian of NodeOS can obtain ACs from a trusted policy server and evaluate them. The evaluation result of a AC is either a *yes* or *no*. The AC is used to control the access to the NodeOS resources, such as channels. Two innovative applications [18, 16] are implemented to show the benefits of the proposed research. They add little performance overhead to the network.

We are currently extending the prototype to a full implementation of the secure node architecture. We plan to demonstrate the power of active security by various applications. The applications include secure routing protocols, security-customized routing paths specified by an application and strengthened protection under intrusion. We also plan to investigate the dynamic reconfiguration and tradeoffs between security protection and satisfaction of the QoS constraints.

5 Conclusions

This paper describes the design of securing the node of an active network. It shows that such a secure node architecture, based on active network principles, can provide fundamental base for securing the active network infrastructure and supporting application specific dynamic security requirements and policies. The research in this paper complements the current active network research and augments its functionality. The secure node architecture provides authentication, authorization, integrity, dynamic access control, and quality of protection for active applications.

The flexibility and expressibility afforded by the secure node enables us to implement a multitude of diverse, innovative and exciting applications. These applications exploit the active networking paradigm without compromising the security of the infrastructure. In addition, our architecture lays the ground rules for seamless integration with parallel and ongoing efforts in the active networks community. The same design principles can be applied to the security support for the execution environment of an active node.

6 Acknowledgments

The authors would like to thank other current Seraphim project members, Prasad Naldurg and Seung Yi, for their contributions to the design and implementation of the Seraphim system. Part of the system is presented in Section 3.1, Section 3.2 and Section 3.3. The authors would also like to thank Jalal Al-Muhtadi for the useful discussions on NodeOS security API design.

References

- [1] The SwitchWare Project Homepage <http://www.cis.upenn.edu/~switchware/>.
- [2] M. D. Abrams and J. D. Moffett. A higher level of computer security through active policies. *Computer & Security*, 14(2):147 – 157, 1995.
- [3] C. Adams. *Independent Data Unit Protection Generic Security Service Application Program Interface (IDUP-GSS-API)*. RFC 2479, December 1998.
- [4] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Java operating systems: design and implementation. Technical Report 98—015, Department of Computer Science, University of Utah, August 1998.
- [5] K. Calvert et al. Architectural framework for active networks. AN Architecture Working Group, Draft, 1998.
- [6] Roy H. Campbell, Zhaoyu Liu, M. Dennis Mickunas, Prasad Naldurg, and Seung Yi. Seraphim: dynamic interoperable security architecture for active networks. In *IEEE OPE-NARCH 2000*, Tel-Aviv, Israel, March 26–27, 2000.
- [7] Roy H. Campbell, M. Dennis 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.
- [8] 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.
- [9] National Computer Security Center. *The Interpreted Trusted Computer System Evaluation Criteria Requirements*, July 1995. Also available at <http://www.radium.ncsc.mil/tpep/library/tcsec/ITCSEC.ps>.
- [10] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 9-12, 1999.
- [11] John Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd Proebsting, and Oliver Spatscheck. Joust: A platform for liquid software. *IEEE Computer*, April 1999.
- [12] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malô, France, October 1997.
- [13] Dexter Kozen. Efficient code certification. Technical Report 98–1661, Department of Computer Science, Cornell University, January 1998.
- [14] C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers. User authentication and authorization in the Java platform. In *15th Annual Computer Security Applications Conference*, Phoenix, AZ, December 6-10, 1999.
- [15] J. Linn. *Generic Security Service Application Program Interface, Version 2*. RFC 2078, January 1997.
- [16] Zhaoyu Liu, Roy H. Campbell, Sudha K. Varadarajan, Prasad Naldurg, Seung Yi, and M. Dennis Mickunas. Flexible secure multicasting in active networks. In *International Workshop on Group Computation and Communications*, Taipei, Taiwan, April 2000.
- [17] Zhaoyu Liu, M. Dennis Mickunas, and Roy H. Campbell. Secure information flow in mobile bootstrapping process. In *International Workshop on Wireless Networks and Mobile Computing*, Taipei, Taiwan, April 2000.

- [18] Zhaoyu Liu, Prasad Naldurg, Seung Yi, Tin Qian, Roy H. Campbell, and M. Dennis Mickunas. An agent based architecture for supporting application level security. In *the DARPA Information Survivability Conference and Exposition*, Hilton Head Island, SC, January 25-27, 2000.
- [19] David Mazières and M. Frans Kaashoek. Secure applications need flexible operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 56–61, Chatham, Cape Cod, Massachusetts, May 1997. IEEE Computer Society.
- [20] S. Merugu, S. Bhattachajee, E. Zegura, and K. Calvert. Bowman: A Node OS for active networks. In *Proceedings of INFOCOM 2000*, March 2000.
- [21] D. Mosberger and L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of OSDI '96*, pages 153–168, October 1996.
- [22] S. Murphy, O. Gudmundsson, R. Mundy, and B. Wellington. Retrofitting security into internet infrastructure protocols. In *the DARPA Information Survivability Conference and Exposition*, Hilton Head Island, SC, January 25-27, 2000.
- [23] Sandra Murphy et al. Security architecture for active nets. AN Security Working Group, July 15, 1998.
- [24] Klara Nahrstedt and Duangdao Wichadakul. QoS-aware active gateway for multimedia communication. In *Proceedings of 6th International Workshop, IDMS '99*, Toulouse, France, October 1999. Lecture Notes in Computer Science 1718, Springer.
- [25] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL '97)*, pages 106–119, January 1997.
- [26] T. Parker and D. Pinkas. *Extended Generic Security Service APIs: XGSS-APIs Access control and delegation extensions*. Internet-Draft, November 1998.
- [27] C. Partridge. *Using the flow label field in IPv6*. RFC 1809, June 1995.
- [28] L. Paterson et al. NodeOS interface specifications. AN NodeOS Working Group, Draft, 1999.
- [29] T. Ryutov and C. Neuman. *Access Control Framework for Distributed Applications*. Internet-Draft, March 2000.
- [30] T. Ryutov and C. Neuman. Representation and evaluation of security policies for distributed system services. In *the DARPA Information Survivability Conference and Exposition*, Hilton Head Island, SC, January 25-27, 2000.
- [31] V. Samar and C. Lai. Making login services independent from authentication technologies. In *Proceedings of the SunSoft Developer's Conference*, March 1996.
- [32] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agent Security, LNCS 1419*. 1998.
- [33] R. S. Sandhu and E. J. Coyne. Role-based access control models. *IEEE Computer*, 29(2), February 1996.
- [34] NSA Cross Organization CAPI Team. *Security Service API: Cryptographic API Recommendation*, July 1996. Second Edition.
- [35] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP '93*.
- [36] Frank Yelin. Low-level security in Java. In *WWW4 Conference*, December 1995.
- [37] L. Zhang, S. E. Deering, D. Estrin, S. Shenker, and D. Zappala. RAVP: A new resource ReSerVation Protocol. *IEEE Network Magazine*, (5), 1993.

PLUGGABLE ACTIVE SECURITY FOR ACTIVE NETWORKS

ZHAOYU LIU, PRASAD NALDURG, SEUNG YI, ROY H. CAMPBELL, M. DENNIS MICKUNAS

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{zhaoyu, naldurg, seungyi, roy, mickunas}@cs.uiuc.edu

ABSTRACT

Security is of critical importance to the success of active networking. In addition, we argue that active security based on active networking principles can offer a wide range of opportunities to build better security systems. This paper describes the integration of active security into a software system implementing the active network architecture. The paper demonstrates that an extensible, reconfigurable security architecture based on active networking is flexible and accommodates a wide variety of security policies and mechanisms. The active security provides users the ability to dynamically create and enforce highly customized and situational policies for their applications. The active security also permits security systems to react to intrusion and can aid the application of the "need-to-know" security principle to network software and application security.

Keywords: active networks, security, reconfigurable, active capability, interoperability

1 INTRODUCTION

An active network provides a software framework that enables network applications to customize the processing of their data [1, 2]. Active applications inject capsules that contain programs (along with data) into the network. Active routers dynamically install these programs and execute them on the data. Though this facilitates fast protocol and service deployment it also makes the routers vulnerable to attacks from arbitrary user-code. Securing the routing infrastructure against threats and exposures remains a major challenge in this paradigm [3].

Traditional networks rely on the underlying operating system to implement security mechanisms and policies. The traditional definition of security in a network environment includes authentication, access control, and encryption. Applications and routers establish a basis for trust by mutual authentication. To protect the integrity of the contents of the capsules, encryption and digital signatures can be employed. Access control mechanisms or policies are defined and enforced to

provide controlled access to the router resources. In addition, active network routers have to provide support to

- prevent malicious behavior of arbitrary user code and
- protect the user code and data from malicious routers

Though a wide range of policy types [4] and mechanisms [5] have been proposed, underlying operating systems implement only a static subset of these policies and mechanisms. Applications that want to use sophisticated or customized policies have to make do with lesser or weaker security guarantees. The overhead associated with adding new policies and mechanisms can also be prohibitive.

In order to exploit the active network flexibility, we have developed a dynamic, fully extensible, interoperable security architecture based on and built into the underlying active network architecture [6]. We term this approach active security [7]. The security architecture enables both static and runtime application-aware reconfiguration [8]. Adaptation allows the security provisions of the network to meet specific individual security requirements within different application scenarios. Applications can request specific security policy instantiations on specific parts of the network, distributing the relevant security policies on a "need-to-know" basis. Alternatively, changes in the security policies for the network can be triggered by the invalidation of a trust model, perhaps by the detection of intrusion or other abnormal behavior.

In this paper we describe the integration of active security into a software system (Bowman and CANES [13, 14]) implementing the active network architecture [12] to showcase the above claimed advantages. Our active security system is composable and can be easily plugged into current active network systems. The integration demonstrates that the active security can provide users the ability to dynamically create and enforce highly customized and situational policies for their applications. It also shows that the active security can permit security systems to react to intrusion and can

aid the application of the "need-to-know" security principle to network software and application security.

The rest of the paper is organized as follows. Section 2 overviews our Seraphim active security architecture. Section 3 describes the integration of our architecture into a software system implementing the active network architecture. Section 4 presents an application example to show the flexibility of the active security based on the current implementation. Section 5 shows the preliminary performance measurement. Section 6 describes the future plan of the integration. The last section concludes this paper.

2 SERAPHIM: ACTIVE SECURITY ARCHITECTURE

Seraphim is a dynamic, flexible, and application specific security architecture that exploits the active, dynamic functionality provided by active networking using an active capability (AC) [6, 9]. Essentially an AC is an executable Java code, which concisely represents dynamic security policies and mechanisms. ACs are evaluated by a security guardian in a secure sandbox environment and the security guardian enforces the security requirements of AC evaluation results. We describe the architecture in more detail next.

2.1 ACTIVE CAPABILITY

Active capabilities are used to support flexible distributed dynamic security policies and services control employing the same active principles as active networks [6, 9]. Unlike a traditional capability, which is merely a static authorization credential that encodes the principal and the permissions associated with the principal, an active capability is a customized piece of code that encodes the type of access control policy and other constraints used in the access control decision making process. In our implementation, an AC is an executable Java code that concisely represents dynamic security policies and mechanisms. In addition, an active capability is protected by digital signatures, resides in user space and can be freely passed around.

An active capability can carry all the decisions policy information in its code. This way of implementation is not modular, elegant and efficient. A better way is to have a generic policy framework that supports different various policy types. ACs use the framework to implement specific policies. An application presents an active capability along with its regular data or protocol capsules to the active router's security guardian at execution time. The access control policy type and user credentials are extracted from the capability. The remote router's security guardian recreates the context of the policy type within its policy framework. If at any point during this process the policy framework discovers that it does not have an implementation for the type of the

policy, it downloads the code dynamically into the framework, using the underlying active network. It then instantiates the run-time parameters associated with the application in its sandbox-like environment and executes the active capability in this environment. Based on the result of the evaluation of this active capability, the access control decision is enforced.

The principal of the active capability, which can be a user, a role, or other principal, must be authenticated by a trusted authority. The trusted authority acts as the policy server in our system. The policy server is responsible for generating ACs, serving ACs to applications and keeping track of ACs. Usually one or more policy servers are associated with each protection domain. Application programs contact their nearest or least-loaded server and obtain the active capability dynamically.

2.2 POLICY FRAMEWORK

The policy framework is an object-oriented and coded in Java. This allows users and commercial organizations to specify policies tailored to their specific operational needs. The framework itself is a hierarchy of classes as shown in Figure 1.

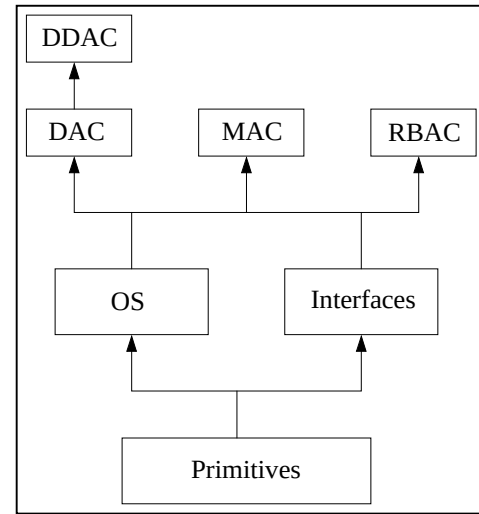


Figure 1: Component-level Map of the Policy Framework

The framework 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 incremented specialized classes representing concepts such as labels and access control lists. Finally, at the top of the framework are classes that can be used to represent a variety of generic policy forms.

The policy framework supports the following common types of access control: Mandatory (MAC),

Discretionary (DAC), Double Discretionary (DDAC), and Role-based (RBAC) [10]. More application specific access control policy systems can be easily extended from this object-oriented framework ([11] provides several good examples). In our model, we can specify not only the traditional $\langle \text{subject}, \text{object}, \text{operation} \rangle$ access control triple, but also include a resource limit on usage, situational decision rules, constraints and dependences, e.g., based on current time of the day or current role of the principal. The main policy type we use for active networks is RBAC because of its flexibility. We will describe its usage in more detail later.

2.3 SECURITY GUARDIAN

The security guardian in the architecture supports AC evaluation and enforcement. The security guardian's functionality is similar to a traditional reference monitor. All accesses to node resources must go through the security guardian. The security guardian uses the security library services to verify the signature on the active capability. To carry out the intended security operations specified by ACs, an evaluation engine and an enforcement engine are included in the security guardian. The evaluation engine evaluates ACs in a secure sandbox. The enforcement engine interacts with other NodeOS components to enforce faithfully the security operations, using the security library services. The enforcement engine can initiate security actions based on ACs requirements. So the security guardian may trigger or initiate security actions. The triggers can be intrusion detection alarms, or explicit requests by execution environments (EEs) or applications that use active networking features. For example, the security guardian can initiate installing firewalls dynamically [6].

3 INTEGRATION OF SECURITY INTO ACTIVE ARCHITECTURE

This section describes the integration of the above security system into a software system implementing the active network architecture [12]. The software system has two parts: the Bowman NodeOS and the CANEs execution environment [13, 14]. We first briefly overview the Bowman and CANEs systems, and then present the integration.

3.1 OVERVIEW OF BOWMAN AND CANES

The Bowman node operating system is built to support the CANEs EE. It is designed around three key abstractions: channel, a-flow, and state-store. A channel is the primary abstraction for communication and an a-flow is the primary abstraction for computation. The state-store provides a mechanism for a-flows to store and retrieve state that is indexed by a unique key. The Bowman is layered on top of a host operating system that

provides lower level services. To make the elementary Bowman channel, a-flow, and state-store abstractions more useful for users, Bowman provides an extension mechanism that is analogous to loadable modules in traditional operating systems. Using extensions, the Bowman NodeOS interface can be extended to provide support for additional abstractions such as queues, routing tables, user protocols and services ([15] provides a more complete NodeOS API).

The CANEs EE is built on the top of the Bowman NodeOS. It provides a composition framework for active services based on customizing a generic underlying program by injecting code to run in specific points called slots. The composition model basically has two parts. The first part, the underlying program, is a fixed part for uniform processing applied to every packet. The second part, the injected program, is a dynamic part that provides user-specific functionality for routing and processing the packets. The injected program is dynamically executed at the appropriate specific points (*slots*) in the underlying program. CANEs uses signaling messages to control the injected programs.

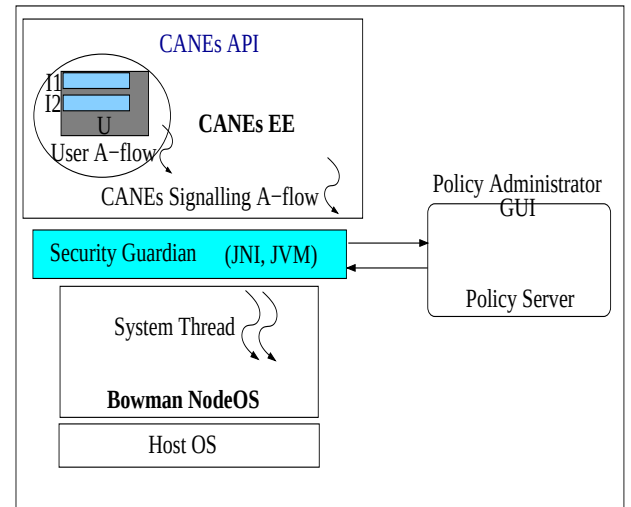


Figure 2: Integration of Active Security into Bowman and CANEs

3.2 INTEGRATION

The integration of active security, CANEs and Bowman is shown in Figure 2. The security guardian is a thin layer between the Bowman NodeOS and the CANEs EE. The Bowman NodeOS interfaces are replaced by the security interfaces. The security guardian does the following security checkings [16]:

- **Authentication:** It verifies the identification and the signature of the request messages. We use X.509 certificates [17] and a simple public key infrastructure (PKI) for authentication.

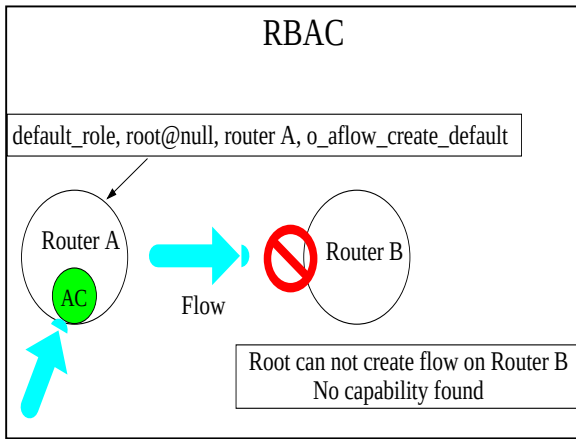


Figure 3: First Demo Scenario

- Authorization: If authentication succeeds, it then checks the access permission for the requests. This requires fetching and evaluating ACs.

Since Bowman and CANEs are written in C to obtain high performance and the Seraphim architecture is implemented in Java for interoperability and security purpose, we use JNI (Java Native Interface) in Bowman and CANEs to invoke the security guardian in Java. When Bowman starts, it starts the security guardian component that invokes Sun JVM. Each security check from CANEs to the Bowman NodeOS security interface is attached to the Sun JVM as a Java thread. After the checking, the Java thread is detached and destroyed.

The security guardian obtains ACs through a secure channel from the policy server. The policy administrator uses a GUI that allows users or system administrators to create and define policy specific attributes and to generate active capabilities. The GUI allows the administrator to create role definitions and associate users and permissions with the role, and supports other functionality (see [10] for more details).

4 APPLICATION EXAMPLE

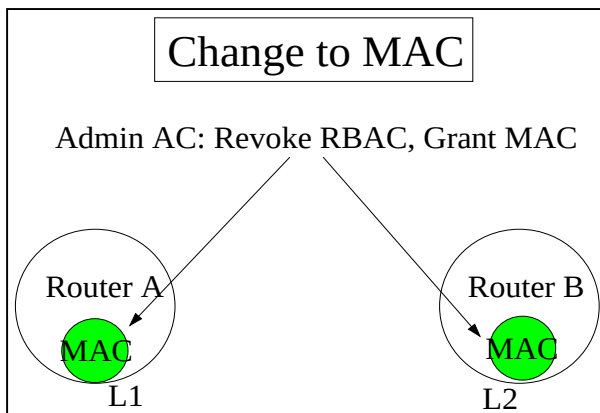


Figure 5: Third Demo Scenario

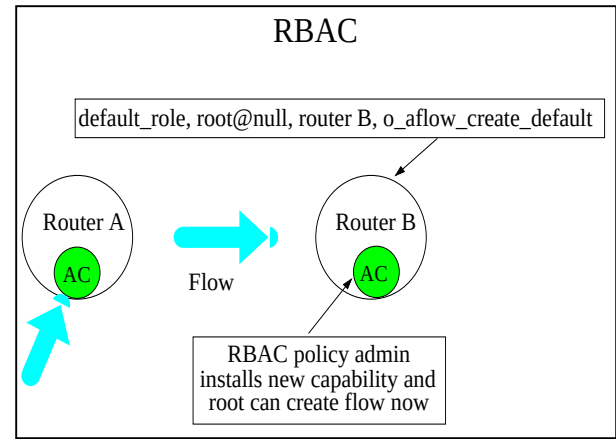


Figure 4: Second Demo Scenario

We have implemented a preliminary version of the authorization part of the integration. Based on the authorization part, we have developed an example application scenario, which is shown in Figure 3, 4, 5 and 6. Figure 3 shows that on behalf of user *root@null* of role *default_role*, the CANEs EE can create an a-flow on the router A, but not on the router B. In order to have a complete flow path from the user *root@null* to the router B, we can dynamically create a new AC through the policy GUI and install it at the router B. Now the CANEs EE can create an a-flow on both router A and B on behalf of the user *root@null* of role *default_role* (Figure 4). The policy type of Figure 3 and 4 is RBAC. If we want to have a stricter and less flexible policy we can dynamically change RBAC to MAC (Figure 5). In this case, the trigger for the policy type change may be an intrusion detection alarm. In MAC policy every entity is assigned a security level. A hierarchy is defined in terms of these levels. Subjects with lower levels cannot read from objects of higher levels and subjects with higher levels cannot write to objects of lower levels. We assume that the MAC level L1 is higher than MAC level L2. This means that the router B has lower security level than the router A. So if user *root@null* is also at security level L1, then user *root@null* can create an a-flow at only the router A since user *root@null* cannot write to router B (Figure 6).

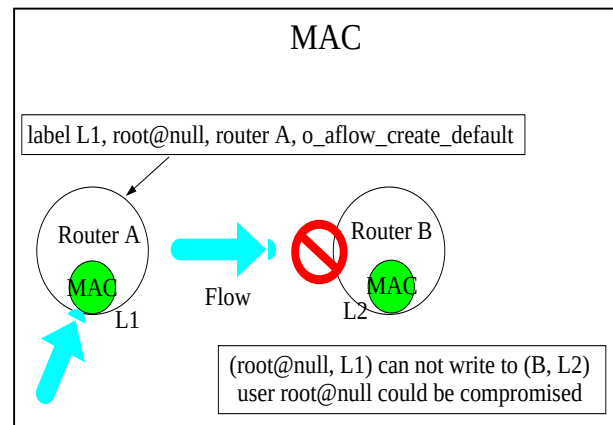


Figure 6: Fourth Demo Scenario

5 PERFORMANCE

The overhead that the integration introduces includes the JNI invocation overhead and the regular security overhead. The regular security overhead, which includes AC fetching and evaluation, is necessary for flexible access control and has been studied previously [6]. We used a simple active ping application (*atraceroute*) between two machines A and B to measure the JNI invocation overhead. Machine A is a Sun Ultra-5 machine, and machine B is a Sun Ultra-2 machine. Both A and B are on the same 100Mbps Ethernet LAN. Machine A sends an *atraceroute* to machine B that is running the Bowman NodeOS. We measure the round trip time (RTT) of the *atraceroute* command with and without JNI invocation (When with JNI invocation, we let security guardian simply return a *true* value in order not to include the regular security overhead). Without any optimization, the RTT is about 2400ms without JNI invocation and about 9100ms with JNI invocation.

In order to improve the performance, we plan to have a leaner JVM replace the current Sun JVM. A possible choice is Kaffe JVM [18]. A more dramatic improvement would be to use a simpler language than Java for ACs. The sandbox evaluation engine of the security guardian of the simpler language must be efficient.

6 FUTURE PLAN

We plan to extend the current integration implementation to provide security checking for all CANEs signaling messages. We plan to add authentication and dynamic revocation to the integration, using the security NodeOS API [16]. We also plan to integrate the Denial of Service prevention features into the system. Finally we will install an experimental setup for the flexible, secure, and composable demanded video distribution application [19] to demonstrate the secure composable services for active networks.

7 CONCLUSION

This paper describes the integration of the Seraphim active security into a software system implementing the active network architecture [12]. The active security architecture is dynamic, fully extensible, interoperable and is based on the underlying active network principles. The integration demonstrates that the active security architecture can be easily plugged into the active network architecture such as Bowman and is flexible and accommodates a wide variety of security policies and mechanisms. We show that active security can provide users the ability dynamically to create and enforce highly customized and situational policies for their applications.

We also show that the active security can permit security systems to react to intrusion and can aid the application of the "need-to-know" security principle to network software and application security. We believe that exploiting active security is a step in the direction of designing a comprehensive and flexible framework to integrate various security mechanisms and services into the active network architecture.

8 ACKNOWLEDGEMENTS

We would like to thank Matt Sanders, Ken Calvert and Ellen Zegura to help us understand the Bowman NodeOS and CANEs execution environment system.

This research is supported by DARPA F30602-98-1-0192.

REFERENCES

- [1] D. Wetherall, J. Gutttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *OPENARCH'98*, 1998.
- [2] D. Wetherall, U. Legedza, and J. Gutttag. Introducing New Internet Services: Why and How. In *IEEE Network Magazine*, July 1998.
- [3] S. Murphy, ed. Security Architecture Draft. AN Security Working Group. Draft.
- [4] Ravi Sandhu. Role-Based Access Control. In *Advances in Computers*, Vol. 46, Academic Press, 1998. Also at <http://www.list.gmu.edu/articles.htm>
- [5] The SwitchWare Project Homepage. <http://www.cis.upenn.edu/~switchware/>
- [6] Zhaoyu Liu, Prasad Naldurg, Seung Yi, Tin Qian, Roy H. Campbell, and M. Dennis Mickunas. An Agent-based Architecture for Supporting Application Level Security. In *the DARPA Information Survivability Conference and Exposition*, Hilton Head Island, SC, January 2000.
- [7] Roy H. Campbell, Zhaoyu Liu, M. Dennis Mickunas, Prasad Naldurg, and Seung Yi. Seraphim: An Active Security Architecture for Active Networks. Tech. Report 2137, Department of Computer Science, University of Illinois at Urbana-Champaign, November 1999.
- [8] Roy H. Campbell, Zhaoyu Liu, M. Dennis Mickunas, Prasad Naldurg, and Seung Yi. Seraphim: Dynamic Interoperable Security Architecture for Active Networks. In *IEEE OPENARCH 2000*, Tel-Aviv, Israel, March 2000.
- [9] Roy H. Campbell, M. Dennis 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,
Vienna, VA, May 1997.

[10] 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.

[11] 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.

[12] K. Calvert, ed. Architectural Framework for Active Networks. AN Architecture Working Group. Draft.

[13] S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert, and E. Zegura. Bowman and CANEs: Implementation of an Active Network. In *Proceedings of 37th Annual Allerton Conference*, Monticello, IL, September 1999.

[14] The CANEs Project Homepage.
<http://www.cc.gatech.edu/projects/canes/>

[15] L. Peterson, ed. NodeOS Interface Specifications. AN NodeOS Working Group. Draft.

[16] Zhaoyu Liu, Roy H. Campbell, and M. Dennis Mickunas. Securing the Node of an Active Network. In *Active Middleware Services*, Salim Hariri, Craig Lee, and Cauligi Raghavendra (editors), Kluwer Academic Publishers, Boston, MA, September 2000.

[17] C. Adams and S. Farrell. Internet X.509 Public Key Infrastructure Certificate Management Protocols. RFC 2510, March 1999.

[18] The Kaffe Homepage. <http://www.kaffe.org/>

[19] The PANAMA Project Homepage.
<http://www.tascnets.com/panama/>

Developing Dynamic Security Policies*

Prasad Naldurg, Roy H. Campbell, and M. Dennis Mickunas
 Department of Computer Science,
 University of Illinois at Urbana Champaign
 IL, 61801, USA
 {naldurg,roy,mickunas}@cs.uiuc.edu

Abstract

In this paper we define and provide a general construction for a class of policies we call dynamic policies. In most existing systems, policies are implemented and enforced by changing the operational parameters of shared system objects. These policies do not account for the behavior of the entire system, and enforcing these policies can have unexpected interactive or concurrent behavior. We present a policy specification, implementation, and enforcement methodology based on formal models of interactive behavior and satisfiability of system properties. We show that changing the operational parameters of our policy implementation entities does not affect the behavioral guarantees specified by the properties. We demonstrate the construction of dynamic access control policies based on safety property specifications and describe an implementation of these policies in the Seraphim active network architecture. We present examples of reactive security systems that demonstrate the power and dynamism of our policy implementations. We also describe other types of dynamic policies for information flow and availability based on safety, liveness, fairness, and other properties. We believe that dynamic policies are important building blocks of reactive security solutions for active networks.

1. Introduction

Policy Management tools provide administrators the ability to specify, implement, and enforce rules to exercise greater control over the behavior of entities in their systems. In this article, we describe the policy development life-cycle for a special class of policies we call *dynamic policies*. Without loss of generality, we model shared resources or entities in a distributed system as objects that export well-defined interfaces. The behavior of an object is controlled

by its interface. These interfaces allow other objects (including objects acting on behalf of users or administrators) to query, access, and modify the objects' operational parameters (or state variables) by calling the appropriate methods, thereby changing the behavior of the system.

Policy management tools automate the task of changing these parameters by providing a simplified cross-platform front-end to system implementations. Currently, most network policies are implemented by systems administrators using tools based on scripting applications [5, 26] that iterate through lists of low-level interfaces and change values of entity-specific system variables. The policy management software maintains an exhaustive database of corresponding device and resource interfaces. With the proliferation of heterogeneous device-specific and vendor-specific interfaces, these tools may need to be updated frequently to accommodate for new hardware or software, and the system typically becomes difficult to manage. As a result, general purpose low-level management tools are limited in their functionality, and are forced to implement only generic or coarse-grained policies [33].

Since most policy management tools deal with these low-level interfaces, administrators may not have a clear picture of the ramifications of their policy management actions. Dependencies among objects can lead to unexpected side effects and undesirable behavior [24]. In Seraphim [23, 7], we remedy this situation by focusing our attention on a special class of policies that are designed with explicit knowledge of system behavior and interactions between various system objects. Our policy development cycle begins with the formal specification of system properties of interest. These properties correspond to security guarantees we want to preserve in our system. Properties are represented as sets of desirable behaviors, described in terms of objects and methods, and are expressed using an appropriate formal notation.

Next, we specify a model of the system behavior with respect to the properties we want to guarantee. This model can be viewed as an abstraction of behavior of the system

*This research is supported by DARPA BAA 98-03 and AFRL Contract Number F30602-98-1-0192

implementation. This specification includes the behavior of all objects that correspond to identifiers in the property specifications, and the transitive closure of the objects that interact with them. We take advantage of model checking techniques [9] to verify that our system specification can satisfy the desirable behavior specified by the properties. Once this is verified, these properties correspond to behaviors that can be guaranteed within the framework of our model. If the property cannot be validated, model checking provides counter-examples that can be used to improve the system design and implementation.

The mapping between validated properties and policy-preserving security policies follows from the specifications. Once the specifications are validated against the system model, we identify specific objects, variables, and methods in the system implementation corresponding to the state variables and mechanisms in the property specifications. These objects and methods automatically form a part of the property-preserving policy implementation and enforcement mechanisms. For example, we allow access to system resource if and only if our system has a rule in its access control rule database that allows the action. This property has to be guaranteed by any access control policy implementation at all times. To change an access control policy, we need to change the corresponding entry in the access control rule database. In addition, we need to guarantee that access is only allowed to objects that can provide authorization proofs. These objects, rules and mechanisms therefore automatically form a part of our policy management infrastructure.

Based on the discussion above, we introduce the concept of a *dynamic* or executable property-preserving policy. A dynamic policy is a program consisting of a set of guards and actions, created by our policy administrator. It encodes not only the logic to modify the system implementation to change operational parameters, but also includes all the necessary guards to enforce good behavior and prevent its misuse. For example, in the access control policy example, the guard can include proofs of authorization, and the commands are programs to change the access control rules. In our Seraphim active network prototype, these programs map directly to active capsules, and can be viewed as in-line policies [27]. These policies are managed, updated and changed by executing the appropriate capsules in a suitable protected software context (NodeOS or EE) [6]. We describe examples of guards and commands for different types of policies in this paper. Active capabilities are special guarded commands for access control policies. These guards and commands allow us to change operational parameters in the policy implementation, without causing undesirable behavior. Policies implemented in this fashion can make strong and verifiable guarantees about system behavior.

We believe that the real application of such policies is in the design of reactive security systems. By including formal analysis, verification, and validation in our policy development life-cycle, we reason about the effectiveness of our policies, and can therefore change operational parameters of dynamic systems with greater confidence. Situational policies in response to attacks can be developed as the system evolves in response to threats and exposures. Once the framework for a new policy type is in place, new policies can be created on the fly, following the specification guidelines. These policies can also be enforced instantaneously by sending and executing guard capsules over networks, during an attack window, to successfully mitigate the impact of an attack. In the course of this article, we describe a framework for specification, enforcement and implementation of these dynamic policies within the active networking context. However, our techniques are general enough to augment any distributed system.

In Section 2 of this paper, we define the class of dynamic policies and present a general method for constructing these policies. We also discuss the threat model for this class of policies. In Section 3 we give a detailed construction of dynamic access control policies that are annotated with authorization proofs, based on preservation of safety properties. In section 4, we provide a brief description of the Seraphim architecture and the implementation of dynamic access control policies in the context of active networks. We also explain why active networks can be used as a framework to develop, disseminate, and enforce such policies, and how these policies enhance the usability of the active networking paradigm. We also briefly describe an example application developed by the Seraphim group that allows an administrator to change between two different access control policy strategies by creating policies on the fly. Section 5 gives examples of other dynamic policies for information flow and availability in terms of safety, liveness, fairness and other properties. We summarize related work in Section 6 and conclude this paper in Section 7.

2. Dynamic Policies

In this section we describe a general method for design, specification, enforcement and implementation for dynamic property-preserving policies. We also describe the threat model and discuss the attack resilience properties of this class of policies.

2.1. Policy Development Life-Cycle

We describe the construction of dynamic policies in a systematic manner. An example of this procedure is the design of Seraphim's dynamic access control policies that

is presented Section 3. Our policy development life-cycle can be broken down into the steps shown in Figure. 1.

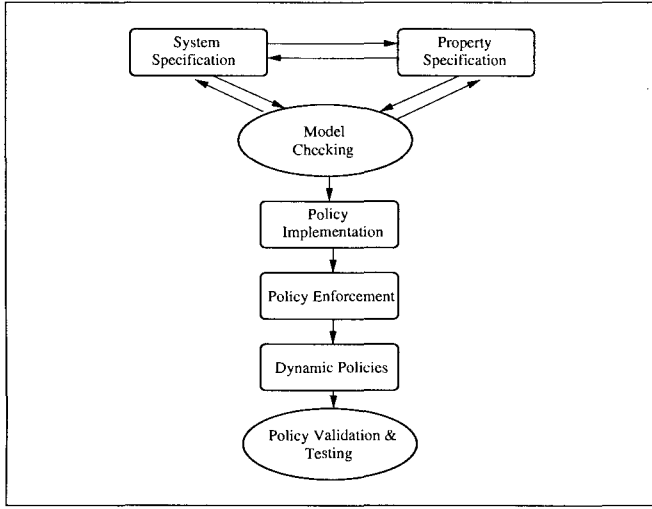


Figure 1. Policy Development Life-cycle

We explain each of these steps in detail below:

1. **Property Specification:** In the first step of our life-cycle, we specify the set of desirable behaviors or security properties that need to be guaranteed in our system, using a suitable language or formal notation. Traditionally, security policies are classified as *access control*, *information flow*, and *availability* policies depending on the type of behavior they control. Different types of properties include safety properties which assert that something bad never happens, liveness properties that assert that something (good) eventually happens, fairness properties that assert that everybody gets a chance to use a resource, etc. As described in the next section, access control policies can be specified as safety properties, and can be specified using simple temporal logic operators (LTL [9], PTL [32] etc.). We investigate the use of different types of logics and logic operators to represent properties for access control, information flow, and availability policies.
2. **System Specification:** This step models the environment of the entities of interest in the property specifications. Lamport [18] states that the behavior of every discrete system can be formally represented by a behavior. Dynamic systems can be modeled in different logics such as Rewriting Logic [2, 10] and TLA [17], by abstracting the behavior of objects and interfaces and mapping these as state variables and actions (e.g., as atomic propositions or predicates) in a formal notation. We develop a behavioral model of the environment of the property that includes not only all the

entities in the property specifications, but also any entities they interact with in the system. We also specify this model using an appropriate formal notation. For example, rewriting logic allows us to develop an executable specification that can be used directly for property checking. The level of abstraction of the model depends on the properties we are interested in, and Steps 1 and 2 are not strictly sequential. Modeling the behavior of the system helps us understand the different entities (or objects), their interfaces and interactions.

3. **Model Checking:** The next step is to check and verify that the properties can be satisfied by the behavior description of the model using model checking [9]. A Model M satisfies a property, expressed as a temporal logic formula f , if $M, s \models f$, where s is an initial state. This step helps us determine what security properties can be enforced in the system, and therefore allows us to identify *what dynamic policies can be implemented*, without sacrificing these guarantees. It is also a useful method to identify fundamental design flaws, to account for side effects and recognize subtle interactions that might induce undesirable behavior.
4. **Mapping and Identification of Policy Implementation Mechanisms:** In this step, we develop the mapping between logical variables in the temporal formulas of the property specification, and the system objects and interfaces of the system implementation. Policies can therefore be implemented by ensuring that the behavior of these objects do not violate the property specifications.
5. **Development of Policy Enforcement Mechanisms:** The policy implementation mechanisms identified in the previous step are augmented with *enforcers* to guarantee that they do not violate their behavior profiles. For example, enforcers for access control mechanisms can include reference monitors that intercept access requests and validate authorization proofs. An enforcer for a fairness policy can include a priority inversion mechanism that can be activated by an authorized user or administrator, to stop a misbehaving object from hogging resources. In general, the system may need to be augmented with enforcers to guarantee desirable behavior, and these form a part of the TCB (Trusted Computing Base). The system is engineered so that it is difficult to gain access or ownership to these enforcers or their interfaces.
6. **Creation of Dynamic Policies:** After the enforcers are in place, the next step is to specify the dynamic policies that can be implemented in the framework of our model and property specifications. These policies

are implemented as code capsules that encode the necessary guards and logic to change the operational parameters of the system objects, without sacrificing the properties.

7. **Policy Validation and Testing:** The final phase in the policy development of dynamic policies is the testing of the dynamic policy implementation. So far, the formal specification and verification of properties, system models and dynamic policies allows us to reason formally about security guarantees. The testing phase actually checks the implementations of these policies to make sure they match the specifications. Software testing may involve type-checking and information flow analysis. Rigorous tests are developed from the formal descriptions of the model and properties and validated by observing the traces.

To summarize this subsection, we present a general construction for a class of security policies called dynamic policies that are based on formal specification, analysis, and verification of the behavior of systems. Dynamic policies provide administrators the ability to change the operational parameters of important system entities, without sacrificing guarantees of good behavior. These policies are implemented as programs and can be created on the fly and implemented by executing them in a suitable context. Our life-cycle helps us identify vulnerabilities and reduce the threat of exposure with respect to policy specification, implementation and enforcement. A concrete example of dynamic policies is given in the next section.

In the next subsection we briefly explain the threat model associated with our policy class.

2.2. Threat Model

Dynamic policies are designed with careful regard to system behavior. However, designing a system resilient to all threats and exposures will require similar guarantees from all components of the system, hardware and software, and adequate social engineering practices. Our policies provide a mechanism to change operational parameters to implement situational policies during the running of a system. The system does not need to be restarted and bootstrapped to change a policy rule. What we provide is a guarantee that changing the policy rules does not leave the system in an undesirable state. With our policies and enforcers, we augment systems with the mechanisms to ensure that the desirable properties that can be satisfied by the underlying system are enforced and not violated.

However, we are also limited by the level of abstraction in our model. While fine-grained abstractions can increase the resilience of our system, the performance penalties may

be unacceptable. Threats to our system can include external factors (e.g., social engineering, insider attacks, compromise of authorization credentials, malicious administrators, etc.) that cannot be modeled as undesirable behavior, because we may not be able to distinguish it from the desirable case.

3. Dynamic Access Control Policies

In this section we describe the specification, implementation and enforcement of dynamic access control policies. In the first subsection, we develop an initial specification of these policies following the general outline described in Section 2. We augment this with authorizations and present the complete specification in Section 3.3. In Section 4, we describe an implementation of these policies in the Seraphim architecture, along with example applications that demonstrate the attack resilience properties of these policies.

3.1. Access Control

In this subsection, we develop a behavioral model of the access control problem in a distributed system. Access-rights to resources are traditionally modeled as access control lists (ACLs) or capability lists. The basic construct used to represent access control is the access-right tuple $\langle \text{subject}, \text{object}, \text{right} \rangle$ which represents a subject's access rights to object interface methods. We restrict our initial specification to this type of access right, though our methodology is general enough to develop specifications for negative access rights, group and role-based access rights etc. ACLs are object-centric and list all the subjects that can access a given object, along with specific access rights. Capability lists are subject-centric and contain a list of objects and rights that can be accessed by a given subject. In most systems these two lists are merged into an Access Control Matrix, which is a table whose rows are subjects and columns are objects. At the intersection of each $\langle \text{subject}, \text{object} \rangle$ pair is the list of allowed methods that can be accessed by the subject. ACLs and capability lists are equivalent with respect to the types of access rights they model [8]. ACLs allow easy revocation and capability lists are easier for delegation, and the use of one or the other is a matter of preference.

An access control operation is therefore a simple lookup operation on the access matrix to check if the subject (or the object acting on behalf of the subject) is allowed to call the method on the relevant object. An access is allowed if and only if the corresponding entry can be found in the access control matrix. A formal specification of the basic access control rule is given as:

$$\boxed{Allow(s, o, m) \Leftrightarrow \langle s, o, m \rangle \in A}$$

Here A is the access control matrix and s, o, m stand for subject, object and method respectively. The desirable behavior of this system or the safety property that is invariant at all times is:

$$\boxed{\square(Allow(s, o, m) \wedge \langle s, o, m \rangle \in A \longrightarrow skip)}$$

Here $\square$ is the *henceforth* or G operator in a suitable temporal logic (LTL or PTL).

From the modeling of the behavior of the system we observe that the safe behavior of the system relies on the proper enforcement of the property defined above. Access Control policy development consists of defining methods to control and modify the Access Control matrix A . Since the *Allow* method has three parameters (subjects, objects and methods), methods that manipulate these parameters form a part of the policy specification. Different access control policies can be enforced by adding or removing subjects, objects and methods. We use the guarded command language [12, 30] to specify set of executable policy management operations (or dynamic policies) in the system. A guarded command is represented as a (*guard* $\longrightarrow$ *command*) where a sequence of guards is followed by a sequence of actions. The guards specify the preconditions necessary to execute the commands. The system specification, with methods to add and remove users, objects and methods to A is given below (we call subjects users here) :

state vars

U : set of USERS initial ϕ

O : set of OBJECTS

A : set of $\langle u : USERS; o : OBJECTS; m : METHODS \rangle$

transitions

$Allowed(u, o, m) \wedge \langle u, o, m \rangle \in A \longrightarrow skip$

$AddUser(u, u') \longrightarrow U := U \cup \{u'\}$

$RemoveUser(u, u') \longrightarrow$
 $U := U - \{u'\};$
 $A := A - \{\langle u', o, m \rangle \mid o \in O, m \in METHODS\}$

$AddObject(u, o) \longrightarrow$
 $O := O \cup \{o\};$

$RemoveObject(u, o) \longrightarrow$
 $O := O - o;$
 $A := A - \{\langle u, o, m \rangle \mid u \in U, m \in METHODS\}$

$AddObjectMethodToUser(u, u', o, m) \longrightarrow$
 $A := A \cup \{\langle u', o, m \rangle\}$

$RemoveObjectMethodFromUser(u, u', o', m') \longrightarrow$
 $A := A - \{\langle u', o', m' \rangle\}$

From this specification we observe from the policy management interfaces that any user in the system is allowed to add or delete subjects, methods, and objects arbitrarily. Furthermore objects and users can be added to the system, without creating any entries to the Access Control Matrix. However removing the objects implies that we need to delete all the corresponding methods. This policy implementation is of little use in the absence of stronger enforcement mechanisms. What is missing in this specification is the notion of authorizations.

In practical access control systems, different sets of users are allowed (or authorized) to create and delete users, objects and methods. In a DAC (Discretionary Access Control) system, only the administrators can create and delete new users. However, users are allowed to create and own objects and add access rights to objects they own. For example, if *user1* owns *file_{user1}*, then *user1* can insert $\langle user2, file_{user1}, read \rangle$ into A . In an MLS system, users and objects can be added only by administrators.

We augment our specification, to make it more meaningful, with special proofs of authorization. In order to change an entry in A , the user is required to produce a proof attesting that he or she is allowed, by some trusted authority, to actually call the relevant method. In order to verify this proof we need to add adequate policy enforcement mechanisms to our design. Since our aim is to prevent unauthorized modification of the capability lists, we add additional guards to our specification to provide the necessary mechanism to guarantee that only authorized modifications are allowed. These guard and action pairs that specify how to change capability lists are called “active capabilities” in Seraphim [7].

3.2. Trust Management

One way of generating such proofs is by using an attestation from a trusted administrator that gives the holder of the attestation the capability to change an access-matrix entry, or the permission to call a method to change the entry. This type of capability (also called a license [34]) or credential is an attestation of trust. These capabilities can be passed around among users and processes that are not running in the Trusted Computing Base. For this reason, they should be protected against modification. An attestation can have the same format as an access matrix entry, and can be made unforgeable by the issuer by attaching a cryptographic digital signature. The signature should tie in the name of the issuer and the intended recipient to prevent modification. This not only prevents modification, but also provides non-repudiation of ownership.

However, using these signed capabilities as attestations to control modification of capability lists is not without problems. Consider the set U of users who can issue signed capabilities, the set O of shared objects in the system and the set M of methods corresponding to access rights. The set of all licenses that can be presented to the policy manager in this system is exponential in the size of these three sets and is given by $C \subseteq U \times \mathcal{P}(O, M)$. In the absence of rules to govern the creation and dissemination of these licenses, the system can quickly become unmanageable. A user can have many different licenses and may present any subset of these to the policy manager during an access control request. The policy manager needs to decide whether the decision is consistent with the trust management implications of these attestations and this may be non-trivial. For example, the monotonicity of the privileges available after revocation may have to be maintained [34] to prevent undesirable behavior.

In our policy management architecture, we do not use signed capabilities to create the authorization proofs. Instead we rely on two simple credentials that attest to the identity of the entities (primarily subjects) and the ownership of one entity by another. The credentials in our system cannot be delegated and are not available to any entity except the policy implementation logic. An example identity credential $typeof(Alice, administrator)$ asserts that the identifier *Alice* is an administrator. The credential $owns(object, method)$ or $owns(user, object)$ attests that the method is “owned” or exported by the object or the object is owned by the user, respectively. Unlike signed capabilities, the size of these credentials is linear in the size of the number of users, objects and methods. All commands that modify the capability lists in the policy logic can include these credentials.

In the next subsection, we augment our specification with these credentials for a specific policy type (DAC) and show how these authorization proofs guarantee only authorized behavior in our system.

3.3. Modified Seraphim DAC Policy

Central to a DAC policy is the notion of ownership. Users are allowed to own objects and set appropriate access control policies for these objects. We associate this notion of ownership with a method *control* that gives the users the ability to delegate rights to access its object methods to other users. If a user has the capability $\langle user, object, control \rangle$, then it can add methods for the object in other capability lists.

In addition to this type of capability, we also have credentials or attestations of trust. From the policy enforcement point of view, we observe that access is allowed only when the corresponding capability can be found in the ca-

pability list A , and when the subject of the access control produces an authorization proof. We present the complete specification of Seraphim’s DAC policy next. The set C contains the identity and ownership credentials attested by the administrator, as required.

state vars

U : set of USERS initial ϕ

O : set of OBJECTS

A : set of $\langle u : USERS; o : OBJECTS; m : METHODS \rangle$

C : set of identity and ownership credentials

transitions

$Allowed(u, o, m) \wedge \langle u, o, m \rangle \in A \rightarrow \text{skip}$

$AddUser(u, u') \wedge typeof(u, admin) \in C \rightarrow$
 $U := U \cup \{u'\};$
 $A := A \cup \{\langle u, u', control \rangle\}$

$RemoveUser(u, u') \wedge \langle u, u', control \rangle \in A \rightarrow$
 $U := U - \{u'\};$
 $A := A - \{\langle u, u', control \rangle\};$
 $A := A - \{\langle u', o, m \rangle \mid o \in O \wedge m \in METHODS\}$

$AddObject(u, o) \wedge owns(u, o) \in C \rightarrow$
 $O := O \cup \{o\};$
 $A := A \cup \{\langle u, o, control \rangle\}$

$RemoveObject(u, o) \wedge \langle u, o, control \rangle \in A \rightarrow$
 $O := O - o;$
 $A := A - \{\langle u, o, m \rangle \mid u \in U \wedge m \in METHODS\}$

$AddObjectMethodToUser(u, u', o, m) \wedge owns(u, o) \in C$
 $\wedge owns(o, m) \in C \rightarrow$
 $A := A \cup \{\langle u', o, m \rangle\}$

$RemoveObjectMethodFromUser(u, u', o', m')$
 $\wedge \langle u', o', m' \rangle \in A \wedge owns(u, o') \in C \rightarrow$
 $A := A - \{\langle u', o', m' \rangle\}$

From the specification, we observe that only administrators are authorized to add users to the system. The identity credential $(typeof(u, admin))$ is sufficient to guarantee this. In *RemoveUsers* we observe that the ability to remove a user depends on the ability to add a user, and by transitive closure, we can assert that only administrators can remove users. Any user can also add a capability for another user, as long as it owns the object and the object exports the required method. We observe from the transition functions *AddObject* and *RemoveObject* and *AddObjectMethodToUser* and *RemoveObjectMethodFromUser*, by transitive closure, that this ownership requirement is indeed enforced by the credentials $(owns(u, o))$ and $(owns(o, m))$.

To validate the model, for the given DAC specification,

we can reiterate that each transition that can change a capability list, along with the credentials and capabilities that are already in the system, under transitive closure, constitutes an authorization proof of why access should be allowed in our system. This is in accordance to the policy requirements. Any access control system built according to our specification is always in a safe state with respect to the manipulation of capability lists.

Other types of access control policies can also be specified and validated using a similar procedure. Seraphim's MLS (Multi-Level Security, which is a type of Mandatory Access Control) policy is based on the Domain and Type Enforcement (DTE) [1] policy. Users belong to domains and objects are classified into types. Policies are implemented using an access control matrix, where the access rights on object methods are stored at the intersection of a domain and type entry. We model this as a capability list indexed by the domain label, instead of the user identifier. The policy enforcer for MLS is responsible for pre-processing the $\langle \text{subject}, \text{object}, \text{method} \rangle$ tuple and converting it into a $\langle \text{domain}, \text{type}, \text{object}, \text{method} \rangle$ tuple required for DTE. Methods to add and remove users, domains, types, objects to types, and domains to users are restricted to the administrator by including a guard that verifies a $\text{typeof}(\text{user}, \text{admin})$ credential. Adding types, objects, and methods to a domain require an additional $\text{owns}(\text{object}, \text{method})$ credential. A partial specification of the DTE policy is given below:

state vars

U: set of USERS
O: set of OBJECTS
D: list of DOMAINS $\{d \mid d \subseteq \mathcal{P}(\text{USERS})\}$ *init default*
T: list of TYPES $\{t \mid t \subseteq \mathcal{P}(o : \text{OBJECTS})\}$
A: set of $\langle d : \text{DOMAINS}; (t : \text{TYPES}, o : \text{OBJECTS}); -m : \text{METHODS} \rangle$
DDT: set of $\langle d : \text{DOMAINS}; d : \text{DOMAINS}; m : \text{METHODS} \rangle$
L: set of credentials

transitions

$\text{Allowed}(d, t, o, m) \wedge \langle d, (t, o), m \rangle \in A \rightarrow \text{skip}$

...

$\text{AddTypeMethToDom}(d, d', t', m') \wedge \text{typeof}(d, \text{admin}) \in \mathcal{C} \wedge \text{owns}(o', m') \in \mathcal{C} \rightarrow$
 $A := A \cup \{\langle d', (t', o'), m' \rangle\}$

$\text{RemoveTypeMethFDom}(d, d', t', m') \wedge \langle d', (t', o'), m' \rangle \in A \wedge \text{typeof}(d, \text{admin}) \in \mathcal{C} \rightarrow$
 $A := A - \{\langle d', (t', o'), m' \rangle\}$

In Seraphim MLS, we tightly couple the types and

objects, to maintain fine-grained control over the addition and removal of types and methods to the access matrix. Note that unlike DAC we do not require the $(\text{owns}(\text{user}, \text{object}))$ credential here. For a complete specification of Seraphim's dynamic access control policies, including RBAC, please refer to [29].

3.4. Model Validation

From the specifications for Seraphim DAC and MLS, we claim that if the credentials are generated correctly and the administrators keys are not compromised, then the executable policies allowed in our system have the required authorization proofs necessary (by transitive closure) to guarantee that authorized access to a resource is synonymous to the possession of unforgeable credentials.

In the next section, we describe briefly the Seraphim active network implementation of dynamic access control policies.

4. Active Networks and Dynamic Policies

In this section we provide a brief overview of our Seraphim security architecture for active networks. For more details refer to [23, 7, 22, 19, 20]. The major components of our architecture and their interactions in the context of the active network architecture are shown in Figure. 2.

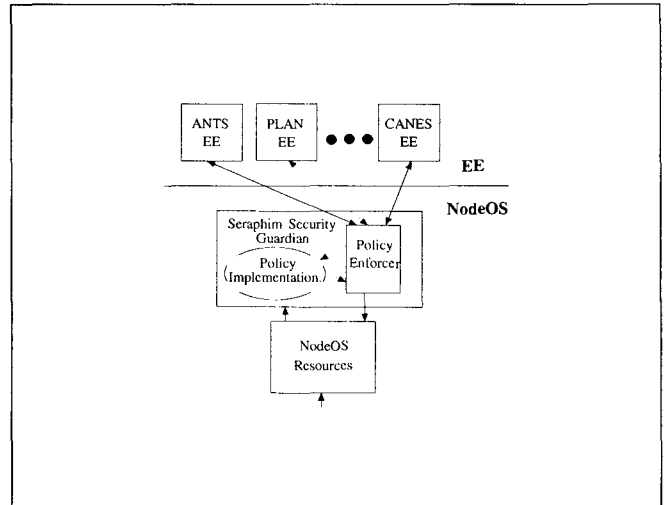


Figure 2. Seraphim Active Network Node

The key component of Seraphim Dynamic Policy Architecture is the Security Guardian. The guardian is the policy enforcement mechanism and is implemented as a collocated extension to the Node OS. Every node has a guardian that intercepts all accesses to node resources. The policy framework implementation is also a part of the guardian. The

implementation is componentized and reconfigurable, and can be downloaded dynamically when required.

To change operational parameters that change the policy implementation, administrators use the interface provided by the policy framework to create a customized piece of code that encodes the type of access control policy and the guarded commands from the policy specifications from the previous section. This code fragment is called the *active capability* (AC) [7, 23]. Unlike a traditional capability, which is merely a static authorization credential that encodes the principal and the permissions associated with the principal, an active capability is actually executable Java bytecode in our implementation¹. In addition, an active capability is protected by cryptographic digital signatures, resides in user space, and can be freely passed around.

An active capability relies on a policy framework for context. An application presents an active capability along with its regular data or protocol capsules to the active router's guardian at execution time. The enforcement mechanisms in the guardian recreates the context of the policy type within its policy framework. If at any point during this process, the policy framework discovers that it does not have an implementation for the type of the policy, it downloads the code dynamically into the framework, using the underlying active network. It then instantiates the run-time parameters associated with the active capability in its sandbox-like environment and executes the active capability in this environment. Based on the result of the evaluation of this active capability, the access control decision is enforced.

From this description, we observe that active networks not only help in realizing the concept of dynamic policies, but also allow the development of "what you need is what you get (WYNIWYG)" implementations. In addition, the paradigm is enriched by these policies because their behavior can be validated by formal methods. One of the main concerns about active networking is the proliferation of code capsules that can cause arbitrary and undesirable behavior. Dynamic policies are examples of capsules that preserve a verifiable and well defined behavior. By deploying dynamic access control policies on chosen network routers, administrators can have greater control over what active capsules are allowed to execute and enforce stricter access control policies on these routers under attack.

In the next subsection, we describe an example application scenario which demonstrates the expressiveness and usefulness of these policies.

¹Note that this definition of active capability is less general than the one described previously

4.1. Example Dynamic Security Scenarios

Our dynamic policy architecture allows many different access control strategies to exist in the same system, though only one type of strategy may be active at any time. Within each strategy, the policy implementation can be changed dynamically without affecting the safety properties, while protecting against unauthorized modifications of these policies at the same time.

We have built two applications of example scenarios to demonstrate our dynamic access control policies from the specifications. The first example is the dynamic firewall.² Initially all routers are bootstrapped with the same access control policy. Then an attack scenario is simulated, where we generate an unwanted ICMP ping capsule directed at a victim router, forcing the victim to reply with another ICMP capsule. An attack detection agent triggers an alarm that dynamically removes the ability of a ping capsule to access the routing table implementation on the victim router. The victim router drops these packets without replying, and propagates a "vaccine" to its upstream router, which deletes its ICMP ping access rule and so on. Eventually, the attacker's flood is stemmed at the source. Once the attack stops at a node, the access rule is reinstated. The dynamic firewall grows outwards from the victim, filtering packets closer and closer towards the attacker and remains in place only as long as the duration of the attack. The average overhead [7] for an application running on a Sparc 10 on the same 100Mbps average time to send and install a vaccine across the network is 34ms.

Our second example demonstrates how we change the DAC policy on a specific computer to a more restrictive MLS policy to protect the integrity of information flowing between sensitive objects on the system. This policy can be deployed in response to an email virus. In the new MLS policy, users are prevented from sending messages on the network by removing their ability to transfer to the network domain and send their messages. This example is straightforward and requires that the policy logic of both MAC and MLS already exist on the system. Policies are developed and installed by the administrator on the fly, and when the administrator is done implementing the policy in the new strategy, the appropriate enforcement mechanism is activated. The performance overheads to switch between two strategies for our simple example was $\approx 2s$. Other examples of dynamic policies in Seraphim and detailed performance numbers can be found in [21, 20].

5. Information Flow and Availability Policies

Security policies are classified as access control, information flow, and availability policies [30]. As we saw in

²This application was demonstrated in [7]

Section 3, access control policies can be specified as safety properties (also shown in [30]). In this section, we examine the other two classes of policies and explore the type of properties and logic needed to specify these policies. Property types include safety, liveness, fairness and denial of service. We also include example specifications of these properties and briefly describe some enforcers for such policies. Since this section describes work in progress, we do not give complete specifications of systems.

5.1. Information Flow Policies

Information flow policies specify controls over flow of information among different classes. Information flow policies can be specified as either safety or liveness properties [30, 32]. An example of a safety information flow policy is the MAC policy that regulates the directional flow (send or receive) of certain classes of information. In this sense, a well-behaving information flow policy is a safety property where no bad flows occur. To enforce this policy, flow filters are added to the points where information flows in and out of the system objects (or ports). A partial specification of the Bell-LaPadula [3] information flow policy as a safety property is shown below:

state vars U: set of USERS
 O: set of OBJECTS
 L: POSet of LABELS immutable
 UM: set of $\langle u : \text{USERS}; l : \text{LABELS} \rangle$
 OM: set of $\langle o : \text{OBJECTS}; l : \text{LABELS} \rangle$

transitions

$\text{Read}(\text{user}, \text{obj}) \wedge \langle \text{user}, \text{label}_{\text{user}} \rangle \in \text{UM}$
 $\wedge \langle \text{obj}, \text{label}_{\text{obj}} \rangle \in \text{OM} \wedge (\text{label}_{\text{user}} \geq \text{label}_{\text{obj}}) \longrightarrow \text{skip}$

$\text{Write}(\text{user}, \text{obj}) \wedge \langle \text{user}, \text{label}_{\text{user}} \rangle \in \text{UM}$
 $\wedge \langle \text{obj}, \text{label}_{\text{obj}} \rangle \in \text{OM} \wedge (\text{label}_{\text{user}} \leq \text{label}_{\text{obj}}) \longrightarrow \text{skip}$

To enforce this policy, immutable labels are added to all objects. User and Object labels can only be changed by the system owner. Before any read or write operation (e.g., before a send or receive), the labels are checked to see if they violate the two safety properties “no read up” and “no write down”, expressed as:

$$\boxed{\square(\text{Read}(\text{user}, \text{obj}) \Leftrightarrow (\text{label}_{\text{user}} \geq \text{label}_{\text{object}}))}$$

$$\boxed{\square(\text{Write}(\text{user}, \text{obj}) \Leftrightarrow (\text{label}_{\text{user}} \leq \text{label}_{\text{obj}}))}$$

To enforce this, all objects that send and receive information between each other have to be augmented with a guard that enforces this property. Dynamic policies in this

case will correspond to the policies that can add users and objects along with their labels.

However, other information flow policies can be expressed as liveness properties. Lamport [16] showed that the liveness property is dependent on the safety properties of sharing mechanisms. Liveness does not imply safety and vice-versa. A classical example is the specification of reliable streaming protocol. A property specification for such a protocol is given as (for all messages):

$$\boxed{\square(\text{send} \longrightarrow \diamond \text{receive})}$$

Here the $\diamond$ operator stands for the *eventually* operator or the F (in the future) operator in LTL. This property implies that all messages sent must be eventually received. A simple windowing protocol satisfies this specification. The enforcer in this case is the windowing implementation, and dynamic policies to change the window size and parameters can change the operational parameters of the protocol without changing its behavioral guarantees. Liveness properties can be specified using the $\square$ and $\diamond$ operators [32]. However, as we see in the next subsection, this property specification does not address fairness and denial of service issues.

5.2. Availability Policies

In the previous subsection, we showed example specifications of liveness properties. However these policies do not impose any bounds on the availability of resources. A sender may experience starvation and never be able to send, in which case channel liveness as specified by the property is vacuously true at all times (recall that an implication $p \rightarrow q$ is true when p is false).

Now consider the following property:

$$\boxed{(\square \diamond \text{send} \longrightarrow \square \diamond \text{receive})}$$

This represents the behavior of a system in which, for all users, if the trace of system behavior contains sends infinitely often, represented by $\square \diamond$, then it also contains receives infinitely often. This says that if the sender does not starve, the message will be eventually received. While this property is adequate to represent the notion of fairness, it does not express availability constraints. Specifically, it does not impose any bounds on the amount of time (finite or minimum) a sender should wait before a user can send in the first place. A policy implementation that ensures availability for sends (or prevents sender starvation) would therefore include some bounds on individual resource consumption, e.g., and enforce this using channel arbitration, priority queues or other fair queuing disciplines that multiplex the senders packets. This ensures that if users want to send, they do not wait forever.

We also observe that the example fairness specifications do not prevent denial of service. What is missing from the specification is the notion of “making progress” or resilience to denial of service. Yu and Gligor [35] develop a Finite Waiting Time (FWT) Policy, to prevent denial of service and show that in order to model the notion of denial of service resistance, in addition to fairness, simultaneity and user agreements are required to make progress. In the channel modeling example, in order to prevent a user from waiting forever, several things need to happen at the same time (simultaneity). When the resource becomes available, the user must have something to send. It is not enough if only one of these conditions hold, to make progress. Denial of service can also take place because another high priority user may decide to send, delaying the send of the lower priority user. Therefore, in order to guarantee a finite waiting time, user agreements to preempt this type of behavior are also required. Millen [25] extends the notion of denial of service resistance by defining a resource allocation model that satisfies MWT or maximum waiting time policies in addition to FWT policies.

We are extending this notion and modeling the Distributed Denial of Service (DDOS) problem, by identifying appropriate property specifications and enforcers, and developing appropriate dynamic policies to change operational parameters while guaranteeing DDOS resilient behavior properties. We have developed an enforcement protocol using credentials that authorizes the use of bandwidth (CABs), along with a dynamic filtering implementation that enforces the user agreements, simultaneity, liveness, and safety properties required to prevent denial of service. The complete details of these mechanisms will be published in a forthcoming paper.

6. Related Work

In this section we present a brief summary of related work. Policy specification, reasoning and trust management are mature areas of research, and to include all related research is beyond the scope of this paper. We only attempt to highlight relevant recent research and any omissions are inadvertent. We classify related research into the following categories: security issues in active networks, enforceable policies, security policy specification, specification of access control policies, and trust management.

The Active Networks Security Working group, which includes the PIs in the Seraphim project, has developed a Security Architecture draft that highlights the important security issues, and a comprehensive threat and trust model for the active networking paradigm. Murphy et al.’s [28] proposal for strong security in active networks discusses the issues and requirements for authentication and authorization mechanisms in the active networking paradigm. This

research deals with the trust assumptions and protection mechanisms required to prevent different active networking entities such as the end-user, NodeOS, EE and active capsules from behaving maliciously and compromising the network infrastructure as a result. Parallel research in the Seraphim group [19, 22, 20] with respect to the use of standard APIs to augment this required security, together with a flow analysis of the security protocols enabled by the APIs, complement our research in dynamic policies. A secure active network infrastructure, though orthogonal to the work presented here, is a prerequisite to the deployment of dynamic policies in active networks.

The PLAN project [14] has developed a “Packet Language for Active Networks” which is a resource-bounded functional programming language. The fundamental construct in the language is remote evaluation of delayed functional applications. By restricting the language, PLAN allows users to develop active capsules that have attack resilience properties, such as CPU and memory denial of service protection, and guaranteed termination. The choice of a programming language to encode dynamic policies is important. In addition to the behavioral guarantees, programming language safety (such as PLAN) needs to be an integral part of any dynamic policy implementation.

Schneider defines a class of policies called Enforceable Policies [30] that can be enforced by execution monitoring. This class of policies is specified using a special automata called Security Automata and is concerned with the preservation of safety properties. Our class of dynamic policies super-scribes this definition by providing a general method for building security properties with verifiable properties, based on the modeling of system behavior and validation of property satisfaction. As such, enforceable policies are an important subset of dynamic policies.

Different notations and languages for security policy specification have been proposed by various researchers. These include the policy specification from the IETF Policy Framework Working Group [33], the original SPKI project, and the SecPol project. The IETF Policy Framework Working Group [33] is working on a policy framework specification that focuses on the development and enforcement of policies for QoS and IPSec applications. A special language to specify policy rules, which consist of a set of conditions and a set of actions, is proposed by this working group.

The SPKI system was proposed to provide mechanisms to support security in a wide range of Internet applications, including IPSEC protocols, encrypted electronic mail, WWW documents and payment protocols etc [13]. The SecPol approach advocates the use of a role-based framework to manage security in large, multi-organizational distributed systems [31]. The SecPol project has developed Ponder [11], a declarative, object-oriented language to specify security policies, group them into roles and relation-

ships, and define management structures. However, none of these projects include a formal modeling of systems for which the policies are developed, or use model checking to verify that the model can actually enforce the properties and policies of interest.

Jajodia et al. [15] propose a language for expressing authorizations and enabling the enforcement of multiple access control policies. They show how programs written in this language effectively capture the abstractions necessary to define different access control models. The security properties of implementation programs are not modeled.

Weeks [34] provides a formal semantics for expressing trust management systems via a fixpoint lattice model for monotonic assertions. This model is useful to understand the trust management of capability-based assertions. Chander et al. [8] provide a state transition approach to model the interaction of trust management and access control. The interaction of access control and trust management including the use of unforgeable credentials to provide authorization proofs and the equivalence of ACLs and capability lists can be validated in their framework.

The capability-based KeyNote [4] system of Blaze et al., provides a single language for both policies and credentials, based on predicates that describe the trusted actions permitted by holders of specific public keys (or other cryptographic identifiers). Our model integrates access control policy management with a simple trust management mechanism. The main purpose of KeyNote is to express and evaluate policies and trust delegations that occur in PKI applications. KeyNote can be integrated into our framework for trust management for other types of dynamic policies that require more expressive credentials.

7. Conclusions

To summarize, in this paper, we describe a general method to construct a special class of security policies we call dynamic policies. Through the policy development life-cycle, we explore the specification, verification, and validation of these policies using suitable formal notations and methods. We also describe the mechanisms for policy enforcement based on the implementation of the specification. The policies created in this framework can be updated by administrators during the operation of the system without compromising the security properties of the implementation. As an example, we develop a formal specification of access control policies as safety properties in the behavioral model of our system, whose enforcement is augmented with proofs of authorization. We also describe our implementation of dynamic access control policies in Seraphim, in the context of active networks, and demonstrate the power of our policies with two examples. Construction of other types of dynamic policies based on liveness, fairness and denial of

service resistance are also briefly described.

At a higher level, our research explores the behavioral descriptions of programs that can be sent across networks to change a system's software state. We explore this in the context of security guarantees that can be made about the system state before, during, and after the execution of such programs. This research is crucial in the context of active networking and in other dynamic environments where operational parameters are constantly changing. Our dynamic policy development life-cycle enables the creation of customizable programs that can be deployed on-the-fly to enforce and implement strong security policies. We also strongly believe that security concerns need to be integrated into models of system behavior, and security properties have to form an integral part of system specifications. Our major contribution is that we present a powerful set of methods and mechanisms that can be used to create policies with strong security guarantees, eliminating guesswork in the design and deployment of reactive security systems.

At a more fundamental level, we also argue that dynamic environments require dynamic security solutions. Dynamic policies enable administrators to react to vulnerabilities detected by IDS and risk analyzers with greater confidence. By including temporal properties in our design of security policies, we can change our system implementations in a controlled manner, and turn on restrictive attack resilient policies at will, without sacrificing security guarantees. This dynamism also allows us to change back to default policies after the attack has been mitigated, allowing us to implement minimal security solutions on a need to protect basis, and amortize performance penalties. We believe that this is our unique contribution in the context of other important research in security for active networks that explore the mechanisms needed to secure the underlying active network infrastructure, language safety issues, and secure bootstrapping etc. One of the major concerns in the active networking paradigm is how to change software state on routers "actively" without sacrificing protection guarantees. We believe that dynamic policies will be important components of solutions to address these concerns.

References

- [1] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A domain and type enforcement UNIX prototype. In *Proceedings of the 5th Usenix UNIX Security Symposium*, Salt Lake City, Utah, June 1995.
- [2] D. Basin, M. Clavel, and J. Meseguer. "rewriting logic as a metalogical framework". In S. Kapoor and S. Prasad, editors, *Twentieth Conference on the Foundations of Software Technology and Theoretical Computer Science, New Delhi, India, December 13-15, 2000, Proceedings*, volume 1974, pages 55-80, 2000.

- [3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, Bedford MA, 1973.
- [4] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures. In *Security Protocols International Workshop, Cambridge, England*, 1998.
- [5] J. Boyle et al. The COPS protocol. Internet Draft, February 24, 1999.
- [6] K. Calvert et al. Architectural framework for active networks. AN Architecture Working Group, Draft, 1998.
- [7] R. H. Campbell, Z. Liu, M. D. Mickunas, P. Naldurg, and S. Yi. Seraphim: dynamic interoperable security architecture for active networks. In *OPENARCH 2000*, Tel-Aviv, Israel, March 26–27, 2000.
- [8] A. Chander, D. Dean, and J. Mitchell. A state-transition model of trust management and access control. In *14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, June 2001.
- [9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
- [10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001. To appear.
- [11] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. Ponder: A language for specifying security and management policies for distributed systems. *Imperial College Research Report*, July 2000.
- [12] E. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [13] C. Ellison, B. Frantz, B. Lampson, R. Rivest, et al. SPKI certificate theory. RFC 2693, September 1999.
- [14] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.
- [15] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, volume 26,2 of SIGMOD Record*, pages 474–485, 1997.
- [16] L. Lamport. A simple approach to specifying concurrent systems. In *System Research Center, DEC, Palo Alto, CA, Report No., 15*, 1986.
- [17] L. Lamport. Specifying concurrent systems with TLA+. In *Calculational System Design*. M. Broy and R. Steinbruggen, editors, 1999.
- [18] L. Lamport. A formal basis for the specification of concurrent systems. Notes for the NATO Advanced Study Institute, June 2000.
- [19] Z. Liu. *Securing the Node of an Active Network*. PhD thesis, University of Illinois, Department of Computer Science, Dec. 2001.
- [20] Z. Liu, R. H. Campbell, and M. D. Mickunas. Securing the node of an active network. In *Active Middleware Services*. Kluwer Academic Publishers, Boston, Massachusetts, September, 2000.
- [21] Z. Liu, R. H. Campbell, S. K. Varadarajan, P. Naldurg, S. Yi, and M. D. Mickunas. Flexible secure multicasting in active networks. In *ICDCS International Workshop on Group Computation and Communication, Taipei, Taiwan, April, 2000*.
- [22] Z. Liu, P. Naldurg, S. Yi, R. H. Campbell, and M. D. Mickunas. Pluggable active security for active networks. In *International Conference on Parallel and Distributed Computing and Systems (PDCS 2000)*, Las Vegas, Nevada, November 6–9, 2000.
- [23] Z. Liu, P. Naldurg, S. Yi, T. Qian, R. H. Campbell, and M. D. Mickunas. An agent based architecture for supporting application level security. In *DARPA Information Survivability Conference and Exposition*, Hilton Head Island, SC, January 25–27, 2000.
- [24] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference*.
- [25] J. K. Millen. A resource allocation model for denial of service. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 137–147, 1992.
- [26] R. Mundy, D. Partain, and B. Stewart. Introduction to SNMPv3. RFC 2570, April 1999.
- [27] S. Murphy et al. Security architecture for active nets. AN Security Working Group, July 15, 1998.
- [28] S. Murphy, E. Lewis, R. Puga, R. Watson, and R. Yee. Strong security for active networks. In *2001 IEEE Open Architectures and Network Programming Proceedings (OpenArch 2001)*, Anchorage, AL, Apr 27–28, 2001. pp 63–70.
- [29] P. Naldurg and R. Campbell. Dynamic access control policies in seraphim. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 2002.
- [30] F. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
- [31] SecPol. SecPol project homepage, 2000. URL: <http://www-dse.doc.ic.ac.uk/projects/secpol/SecPol-overview.html>.
- [32] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing* 6(5): 495–512 (1994), updated 1999.
- [33] M. Stevens et al. Policy framework. IETF draft, September 1999.
- [34] S. Weeks. Understanding trust management systems. In *2001 IEEE Symposium on Security and Privacy*, May 2001.
- [35] C.-F. Yu and V. D. Gligor. A formal specification and verification method for the prevention of denial of service. In *Proc. 1988 IEEE Symposium on Security and Privacy (Oakland '88)*, Oakland, CA, USA, Apr. 1988, pp.187–202.