UNIVERSITY OF CALIFORNIA
IRVINE

A Connector-Centric Approach to Architectural Access Control

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Jie Ren

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Debra J. Richardson
Professor David F. Redmiles

2006

The dissertation of Jie Ren
is approved and is acceptable in quality
and form for publication on microfilm:

____

____

____
Committee Chair

University of California, Irvine
2006

DEDICATION

To

my mother, Jingze Zhang

and

my father, Zugen Ren

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Professor Richard N. Taylor. His insights, encouragement, and feedback are invaluable. Thanks also to Professor Debra J. Richardson, Professor David F. Redmiles, and Professor Paul Dourish, whom I had the honor to work with and receive indispensable guidance and help.

Special thanks to Eric Dashofy, whose incredible work on xADL and ArchStudio form the foundations for this research, and whose insights and humor have made graduate school an enjoyable experience.

Thanks to my officemates, Yuzo Kanomata and Justin Erenkrantz. Not only have they provided priceless technical information, but they also have taught me numerous other lessons.

Other members of the C2 group, including Hazel Asuncion, Joe Feise, John Georgas, Michael Gorlick, Scott Hendrickson, Girish Suryanarayana and Debra Brodbeck, have provided valuable help in various forms.

Members of the Swirl group, including Rogerio DePaula, Xianghua Ding, Ben Pillet, Jennifer Rode, and Roberto Silva Filho, have greatly helped in developing the Impromptu system. Kari Nies, who is also the author of the original Coalition application, is worth special thanks.

Thor Larholm, Oliver Lavery, and Geoff Shively of PivX Solutions introduced me to the industry side of software security. That experience is irreplaceable.

I would also like to thank Professor Ninghui Li from Purdue University for allowing me to use the code for the RBTM framework, thank Professor Gregorio Martinez and Professor Antonio Skarmeta from University of Murcia for permission to integrate the XACML editor, and thank Seth Proctor and Anne Anderson from Sun for useful discussions on SunXACML and RBAC.

This work was supported in part by the National Science Foundation under grants 0133749, 0205724, and 0326105, a grant from the Intel Corporation, and a UC Irvine Dissertation Fellowship.

CURRICULUM VITAE

Jie Ren

Education

1999.9-2006.1	Ph.D.	Information and Computer Science, University of California, Irvine
1992.9-1995.7	M.Sc.	Department of Computer Science, Fudan University
1988.9-1992.7	B.Sc.	Department of Computer Science, Fudan University

Working Experience

2004.6-2004.9	PivX Solutions, Inc., Intern Security Researcher
2002.7-2002.9	Endeavors Technology Inc., Intern Quality Assurance Engineer
1998.9-1999.7	Department of Computer Science, Fudan University, Lecturer
1996.9-1998.8	Department of Computer Science, Fudan University, Assistant Lecturer
1995.7-1996.8	ZTE Corporation, China, Software Engineer

Teaching Experience

- Reader, ICS 142, Compilers and Interpreters, Winter 2004
- Reader, ICS 121, Software Tools and Methods, Fall 2003
- Teaching Assistant, ICS 52, Introduction to Software Engineering, Spring 2003
- Reader, ICS 125, Project in Software System Design, Winter 2003
- Teaching Assistant, ICS 123, Software Architectures, Distributed Systems, and Interoperability, Fall 2002
- Teaching Assistant, ICS 121, Software Tools and Methods, Spring 2000
- Teaching Assistant, ICS 125, Project in Software System Design, Winter 2000
- Teaching Assistant, ICS 125, Project in Software System Design, Fall 1999
- Co-Instructor, Advanced Software Engineering, Fudan University, Spring 1999
- Co-Instructor, Advanced Software Engineering, Fudan University, Spring 1998

Publications

Jie Ren, Richard Taylor, *A Secure Software Architecture Description Language,* Proceedings of the Workshop on Software Security Assurance Tools, Techniques, and Metrics, held in conjunction with the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, California, USA, November 7-11, 2005.

Jie Ren, Richard Taylor, *Automatic and Versatile Publications Ranking for Research Institutions and Scholars*, to appear in the Communications of the ACM.

Rogerio de Paula, Xianghua Ding, Paul Dourish, Kari Nies, Ben Pillet, David Redmiles, Jie Ren, Jennifer Rode, Roberto Silva Filho, *In the Eye of the Beholder: A Visualization-based Approach to Information System Security*, International Journal of Human-Computer Studies (IJHCS), Vol. 63, No. 1-2, pp. 5-24, July 2005.

Rogerio de Paula, Xianghua Ding, Paul Dourish, Kari Nies, Ben Pillet, David Redmiles, Jie Ren, Jennifer Rode, Roberto Silva Filho, *Two Experiences Designing for Effective Security*, Proceedings of the 2005 Symposium On Usable Privacy and Security, pp. 25-34, Pittsburgh, PA, July 6-8, 2005.

Jie Ren, Richard Taylor, Paul Dourish, David Redmiles, *Towards An Architectural Treatment of Software Security: A Connector-Centric Approach*, Proceedings of the Workshop on Software Engineering for Secure Systems, held in conjunction with the 27th International Conference on Software Engineering, St. Louis, Missouri, USA, May 15-16, 2005.

Jie Ren, Richard Taylor, *Utilizing Commercial Object Libraries within Loosely-Coupled, Event-Based Systems*, Proceedings of the 8th IASTED International Conference on Software Engineering and Applications , pp. 192-197, Cambridge, Massachusetts, USA, November 9-11, 2004.

Jie Ren, Richard Taylor, *An Automatic and Generic Framework for Ranking Research Institutions and Scholars based on Publications*, Technical Report UCI-ISR-04-5, June 2004.

Jie Ren, *Modular Security: Design and Analysis*, Technical Report UCI-ISR-04-4, June 2004.

Jie Ren, Richard Taylor, *Visualizing Software Architecture with Off-The-Shelf Components,* Proceedings of the 15th International Conference on Software Engineering & Knowledge Engineering, pp. 132-141, San Francisco, California, USA, July 1-3, 2003.

Jie Ren, Richard Taylor, *Incorporating Off-The-Shelf Components with Event-based Integration*, Proceedings of the ISCA 12th International Conference on Intelligent and Adaptive Systems and Software Engineering, pp. 188-191, San Francisco, California, USA, July 9-11, 2003.

Jie Ren, Richard Taylor, *Incorporating Off-The-Shelf Components with Event-based Integration*, Technical Report UCI-ISR-03-2, April 2003. (This is a longer version of the above paper.)

Jie Ren, *Internet-scale Event Notification: Architecture Alternatives*, position paper for Workshop on Evaluating Software Architectural Solutions, Irvine, California, USA, May 8-9, 2000.

Junfeng Wang, Xiaobin Qi, Kuanli Xia, Jie Ren, *Design Patterns and UML*, Application Research of Computers, vol. 16, no.5, pp. 27-30, May 1999. In Chinese.

Shengxin Zhang, Jie Ren, Leqiu Qian, *Investigation and Research on Components Matching Methods*, Computer Engineering, vol. 25, no. 3, pp. 8-10, March 1999. In Chinese.

Jie Ren, Wenyun Zhao, Yongxue Sun, Leqiu Qian, *Research on Domain-Specific Software Architecture*, Computer Engineering, vol. 23, Special Issue, pp. 222-224, December 1997. In Chinese.

Jie Ren, Leqiu Qian, *The Object-Oriented Development of C Coding Tool*, Proceedings of 5th Chinese National Conference on Software Engineering, Shanghai, China, December 1993. In Chinese.

ABSTRACT OF THE DISSERTATION

A Connector-Centric Approach to Architectural Access Control

By

Jie Ren

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2006

Professor Richard N. Taylor, Chair

An important problem is the architectural access control question: how can we describe and check access control issues at the software architecture level? We propose a connector-centric approach for software architectural access control. Our approach is based on a unified access control model incorporating the classic model, the role-based model, and the trust management model.

We design a secure software architecture description language, Secure xADL, that extends the xADL language with constructs necessary to describe access control issues. Secure xADL extends descriptions of components, connectors, their types, sub-architectures, and the global architecture with subject, principal, permission, resource, privilege, safeguard, and policy. We use the XACML language as the basis for architectural security policy modeling. Four types of contexts for architectural access control are also identified: 1) the nearby constituents of components and connectors, 2) the types of components and connectors, 3) the containing sub-architecture, and 4) the global architecture.

We present an algorithm to check architectural access control: given a secure software architecture description written in Secure xADL, if a component A wants to access another component B, should the access be allowed?

Tool support is provided as part of the ArchStudio architecture development environment, including an editor, a checker, the secure architecture controller, and a run-time framework enabling important architectural operations: instantiating components and connectors, connecting components to connectors, and message routing.

Connectors play a central role in our approach. They can propagate privileges within the architecture, decide whether architectural connections can be made, and route messages according to their security policies.

Our hypotheses are: an architectural connector may serve as a suitable construct to model architectural access control; the connector-centric approach can be applied to different types of componentized and networked software systems; the access control check algorithm can check the suitability of accessing interfaces; in an architecture style based on event routing connectors, our approach can route events in accordance with the secure delivery requirements.

To validate these hypotheses, we have performed an informal analysis of the algorithm, developed two applications, Secure Coalition and Impromptu, and modeled the security architecture of Firefox and DCOM.

1 Introduction

1.1 Problem Summary

The past decade has seen rapid penetration of information technology into every aspect of our society. More organizations and individuals have been transforming their work and lives with ever increasing computation power and communication capability. Such a trend will continue to change the way that our society operates.

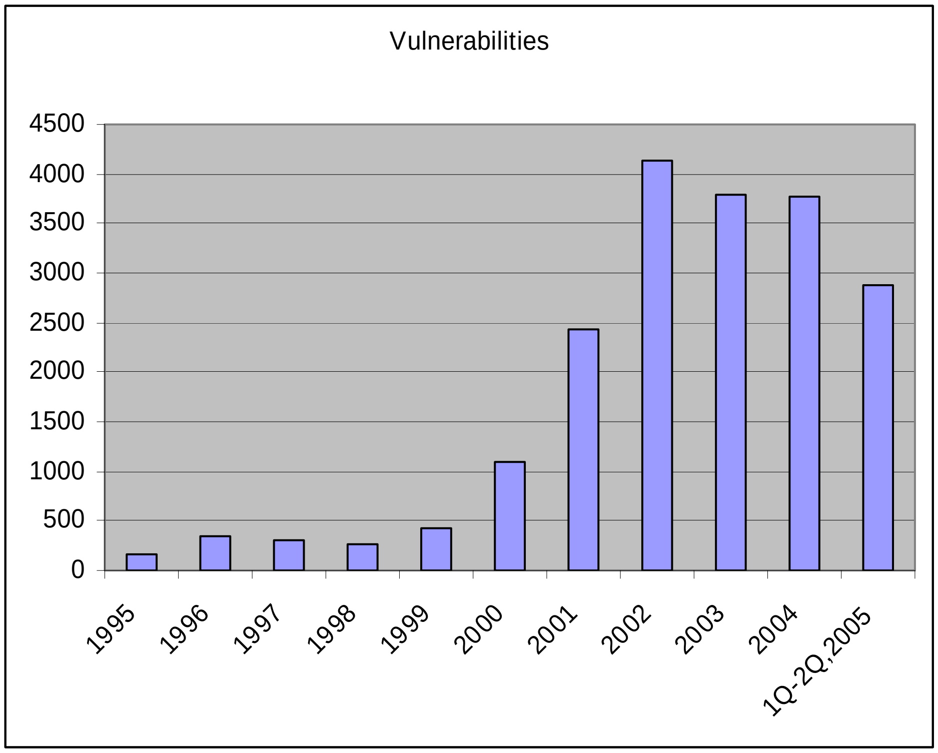


Figure 1-1, Vulnerabilities reported to CERT

Unfortunately, such transformation has certain undesirable side effects. Rampant security breaches are one of the most prominent examples of these unwelcome consequences. More than 3500 vulnerabilities were reported to the Computer Emergency Response Team Coordination Center (CERT/CC) each year during the past three years (see Figure 1-1). There were about 140,000 security incidents reported to CERT in 2003 (see Figure 1-2). Such incidents have become so commonplace that CERT has stopped publishing these statistics since 2004.

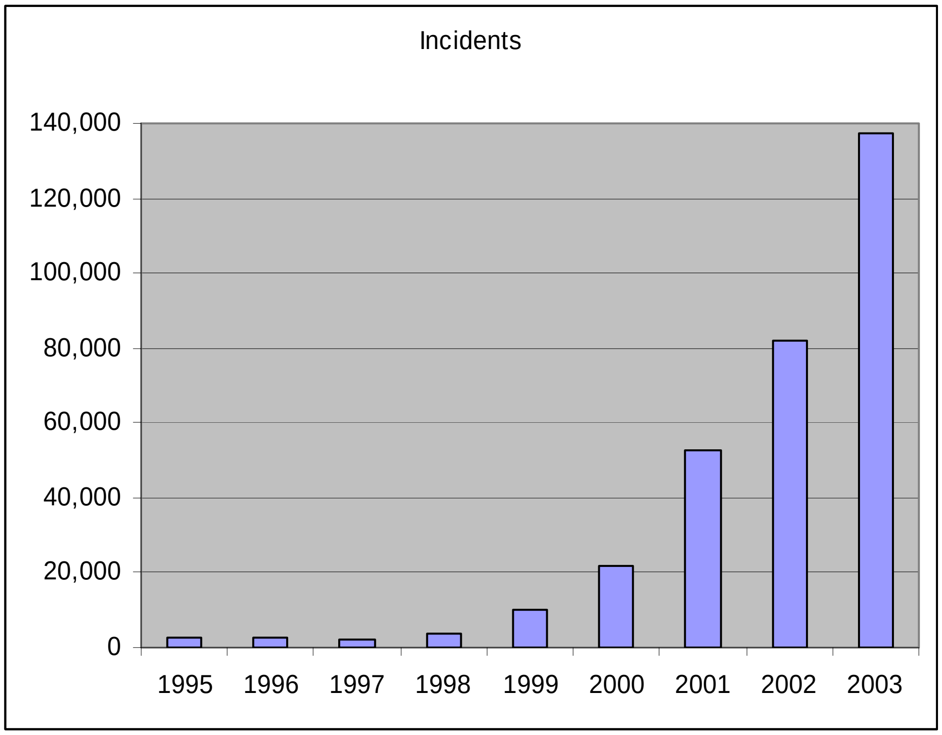


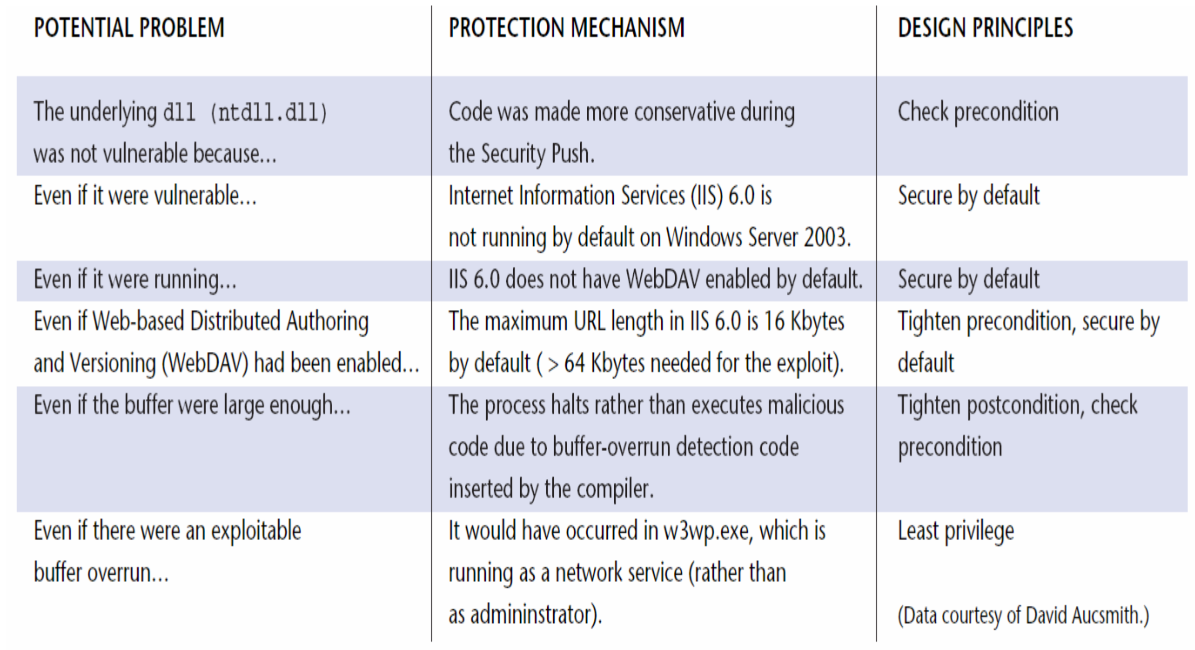
Figure 1-2, Incidents reported to CERT

The unsatisfactory situation of software security partly comes from the mechanism used to build the software systems and the environment in which these systems are deployed and operated. More and more software is built from existing components. These components may come from different sources. This complicates analysis and composition, even when a dominant decomposition mechanism is available. Additionally more and more software is running in a networked environment. The fast and permanent network connections open possibilities for malicious attacks that were not possible in the past. These situations raise new challenges on how we develop secure software.

Traditional security research has been focusing on how to provide assurance on confidentiality, integrity, and availability [14]. However, with the exception of mobile code protection mechanisms, the focus of past research is not how to develop secure software that is made of components from different sources. Previous research provides necessary infrastructures, but a higher level perspective on how to utilize them to describe and enforce security, especially for componentized software, has not received sufficient attention from research communities so far. Despite occasional cryptology-related attacks [25, 142], most security vulnerabilities result from poor software design and implementation, such as the ever-lasting buffer overrun bugs. Thus approaches to designing secure software, not just from a traditional cryptology viewpoint, but with a software engineering perspective, are needed to counter the current unsatisfactory situation.

Software architecture research, which has been an active field in the past decade, could provide an appropriate angle for guiding design of secure software. The potential of an architecture-guided secure software design approach can be illustrated by the rearchitecting of a major web server, Microsoft Internet Information Service, whose security has been significantly improved by a carefully designed software architecture, without introduction of major cryptography features [146]. The Microsoft Internet Information Server (IIS) web server was first introduced in 1995. It has gone through several version changes during the following years, reaching Version 5.1 in 2001. Along this course, it was the source of several vulnerabilities, some of which were high profile and have caused serious damages [9]. A major architectural change, as outlined in Figure

1-3, was introduced in 2003 for its Version 6.0. This version is much safer than previous versions, due to these architectural changes [146]. No major cryptology technologies were introduced with this version. Only existing technologies were rearchitected for better security. This rearchitecting effort suggests that more disciplined approaches to utilize existing technologies can significantly improve the security of a complex, componentized, and networked software system.



POTENTIAL PROBLEM	PROTECTION MECHANISM	DESIGN PRINCIPLES
The underlying `dll (ntdll.dll)` was not vulnerable because…	Code was made more conservative during the Security Push.	Check precondition
Even if it were vulnerable…	Internet Information Services (IIS) 6.0 is not running by default on Windows Server 2003.	Secure by default
Even if it were running…	IIS 6.0 does not have WebDAV enabled by default.	Secure by default
Even if Web-based Distributed Authoring and Versioning (WebDAV) had been enabled…	The maximum URL length in IIS 6.0 is 16 Kbytes by default (> 64 Kbytes needed for the exploit).	Tighten precondition, secure by default
Even if the buffer were large enough…	The process halts rather than executes malicious code due to buffer-overrun detection code inserted by the compiler.	Tighten postcondition, check precondition
Even if there were an exploitable buffer overrun…	It would have occurred in w3wp.exe, which is running as a network service (rather than as admininstrator).	Least privilege (Data courtesy of David Aucsmith.)

Figure 1-3, IIS Rearchitecting, from [146]

Component-based software engineering and software architecture provide the necessary higher-level perspective for system wide security. Security is an emergent property, so it is insufficient for a component to be secure. For the whole system to be secure, all relevant components must collaborate to ensure the security of the system. An architecture model guides the comprehensive development of security. Such high-level modeling enables designers to locate potential vulnerabilities and install appropriate countermeasures. It facilitates checking that security is not compromised by individual components and enables secure interactions between components. An architecture model also allows selecting the most secure alternatives based on existing components and supports continuous refinement for further development.

We present a software architecture-based approach that can help design and analyze secure software. Specifically, the approach extends traditional software architecture descriptions with the capability to describe and check access control policies. Access control, which controls how protected computational resource can be accessed, is arguably the most dominant security assurance mechanism. We are trying to answer the following **architectural access control** question: ***how can we describe and check access control issues at the software architecture level?***

Answering this question can bring deeper and more comprehensive modeling of architectural security, and help software architects detect architectural vulnerabilities and support correct access control at the architecture level. While such an answer cannot fully solve the general software security problem, it can complement and possibly guide other solutions that operate on the mathematical properties and low-level implementations to collectively provide the comprehensive solution that is necessary for a complex, componentized, and networked software system.

1.2 Approach

We propose a connector-centric approach for software architectural access control. The approach is based on a unified access control model. The model guides the design of a secure architecture description language that can describe architecturally significant operations. The validity of access within software architecture is checked by an algorithm that takes different types of architectural contexts into consideration. Connectors play a central role in establishing the security of these operations. Tools are provided to help software architects use this approach for designing and analyzing software security.

Our approach is based on a unified access control model that incorporates the classic access control model, the role-based access control model, and the trust management model. The classic model describes access control with a set of subjects that have permissions and a set of objects on which these permissions can be exercised. The role-based model introduces the concept of roles as an indirection to organize the permission assignments to subjects. The trust management model provides a decentralized approach to manage subjects and delegate permissions. The unified model uses subjects, objects and permissions to integrate the three models.

We design a secure software architecture description language, Secure xADL [115], that extends our extensible architecture description language, xADL [27], with constructs that are necessary to describe access control issues. Secure xADL extends descriptions of architectural constituents (components, connectors, their types, sub-architectures, and the global architecture) with the following constructs: **subject**, **principal**, **permission**, **resource**, **privilege**, **safeguard**, and **policy**. **Subject** is the user on whose behalf software constituents execute. A subject can take multiple principals. Each **principal** encapsulates a credential that the subject possesses to acquire permissions. A **permission** is an allowed operation on a resource. A **resource** is an entity whose access should be protected. A resource can be passive, like files, or it can be active, like components and connectors. A **privilege** describes permissions that components and connectors possess, depending on the executing subject. A **safeguard** describes permissions required to access the protected interfaces of components and connectors. A **policy** ties all these concepts together, and specifies what access is allowed and what access should be denied. We use the eXtensible Access Control Markup Language (XACML) [106] as the basis for our architectural security policy modeling.

In addition to the access control policies locally specified in components and connectors, we use **context** to designate those additional relationships involved in architectural access control for the components and connectors. Such contexts change how access control is regulated. There are four types of contexts: 1) the nearby constituents of components and connectors, 2) the types of components and connectors, 3) the explicitly modeled sub-architecture that contains components and connectors, and 4) the global architecture. These contexts are integrated in policy modeling through the XACML policy combination mechanism.

We present an algorithm to check architectural access control at design-time: ***given a secure software architecture description written in Secure xADL, if a component A wants to access another component B, should the access be allowed?*** Our algorithm is based on graph reachability analysis. The algorithm obtains necessary privileges, retrieves relevant policies from the established contexts, and makes a decision based on the privileges and the policies.

Tool support for the approach is also provided, as part of the ArchStudio architecture development environment [27]. An editor allows an architect to specify the architectural access control policies. A checker statically checks that the given component can access an interface of another component. A run-time framework enables applications to perform important architectural operations: instantiating components and connectors, connecting components to connectors, routing messages from one component to another connector, and forwarding messages between interfaces of a connector.

Connectors play a central role in our approach. Depending on their types, connectors can propagate privileges within the architecture, decide whether architectural connections can be made, and route messages according to its security policies.

1.3 Hypotheses and Validation

The hypotheses for this research are that:

Hypothesis 1: An architectural connector may serve as a suitable construct to model architectural access control.

Hypothesis 2: The connector-centric approach can be applied to different types of componentized and networked software systems.

Hypothesis 3: With a Secure xADL description, the access control check algorithm can check the suitability of accessing interfaces.

Hypothesis 4: In an architecture style based on event routing connectors, our approach can route events in accordance with the secure delivery requirements.

To validate these hypotheses, we have analyzed the validity of the algorithm by mapping it to a well known graph reachability problem, and we have also developed two applications and analyzed two third party software systems.

The first application developed was a secure coalition application. It is a C2-style application that uses events as the communication mechanism. Our approach enables architects to decide how the components are connected and how such connections can be used to deliver events according to the desired security requirements.

The second application developed was Impromptu, a peer-to-peer file sharing application that allows different users to share files freely and securely. The application visualizes security relevant events so users can make better informed decisions on security issues. This application uses the Jetty HTTP/Servlet server and the Apache Slide WebDAV server. A secure WebDAV proxy is developed to connect communicating peers. Users can use off-the-shelf standard-compliant applications to access and manipulate those shared files.

In addition to designing these two applications, we also have analyzed two existing systems. The first analyzed system was the Firefox open source web browser. Compared to another popular web browser, Internet Explorer, Firefox does not support ActiveX but makes extensive use of JavaScript. The script segments from a web page are executed by the script engine, subject to access control policies. We analyzed how this access control is enacted by various architectural connectors.

The second system analyzed was Microsoft's DCOM. DCOM provides network invocable services between clients and servers. The services can have different levels of authentication and authorization properties. An intermediate server can also act on behalf of a client to access a third server. We modeled DCOM's role as a secure connector between clients and servers using our approach.

Through these analysis, design and modeling activities, we have validated our four hypotheses, and illustrated the feasibility of our connector-centric approach.

Firstly, we have demonstrated that **an architectural connector may serve as a suitable construct to model architectural access control.** Connectors propagate privileges that are necessary for access control decisions. They regulate architectural connections between components. And they can also coordinate message routing securely.

Secondly, we have established that **the connector-centric approach can be applied to different types of componentized and networked software systems.** Impromptu demonstrates that our approach can be applied to develop a system composed of externally developed components connected through secure connectors. The modeling of Firefox shows the applicability of our approach in handling security of untrustworthy components. Modeling DCOM demonstrates that our approach can model a large and complex network application that has complex access control requirements.

These studies demonstrate that our approach is applicable to different types of software systems. The secure coalition application is based on an internally developed Java framework. Impromptu is a Java-based application that extensively reuses existing components. Firefox is a third party C/C++/JavaScript-based cross-platform application. DCOM is a third party C/C++-based Windows application. The diversity of these systems shows that our approach can support heterogeneous environments.

The studies demonstrate that our approach can be applied to both designing a new component-based application and analyzing an existing-component-based application. We have developed the secure coalition application with an internally developed framework. We have also developed Impromptu with many existing components. Both Firefox and DCOM are third party applications. We have modeled their component-based architectures and investigated the security implications of architectural choices.

The studies demonstrate that our approach can handle security properties of different types of software connectors. The secure coalition application uses a C2 style broadcast connector. Impromptu composes several connectors into a composite connector. Firefox is a host-based application whose operations are mostly limited to a client's machine. Different components interact with each other through traditional API connections, in the form of a cross-language connector. DCOM is an application based on a network protocol. The components of this application, the clients and the servers, interact through the protocol. The various types of connectors (event routing connector, local procedure call connector, network application protocol connector, and their composites) illustrate that our approach is capable of modeling access control properties of different software connection mechanism.

Thirdly, we have shown that **with a Secure xADL description, the access control check algorithm can check the suitability of accessing interfaces.** We transform a Secure xADL architecture description into a graph, where the nodes stand for privileges and safeguards, and the edges represent connections permitted by policies of connectors. We have shown that permitting an architectural access between a pair of interfaces roughly equals to finding a path in the constructed graph, and thus any standard solution to the reachability problem can be used.

Finally, we have illustrated that **in an architecture style based on event routing connectors, our approach can route events in accordance with the secure delivery requirements.** The secure coalition application, in the event-based C2 style, can use different types of message routing connectors to route messages that only one party deems secure to share with the other party. These connectors, being part of an application framework, can also be used in constructing other C2 style applications.

1.4 Contributions

Our research contributions include a design and analysis method for security in software architecture research, a formal language to model access control at the architecture level, an algorithm to check the correctness of such models, a technique to compose secure connectors from existing connectors, and a suite of support tools. This approach contributes to deeper and more comprehensive modeling of architectural security, and facilitates checking correct access control and detecting vulnerabilities at the architecture level.

More specifically, the contributions of this research are as follows:

A novel approach to the design and analysis of the access control property for software architectures. We address the access control problem from an architectural viewpoint and use an architecture model to guide the design and analysis of architectural access control in software systems. Access control is a very important security property at the architecture level. We provide a comprehensive treatment of this property. The treatment employs architectural contexts such as neighboring constituents, types, containing sub-architecture, and global architecture, to facilitate the design and analysis of access control. Our approach combines researches from software architecture and security, and it is the first approach that addresses the access control problem from an architectural level. The feasibility of our approach is illustrated through our design and analysis of four significant case studies.

A usable formalism for modeling and reasoning about architectural access control. We propose a secure architecture description language that can unify related security concepts, such as subject, principal, permission, resource, privilege, safeguard, and policy, at the architecture level. This language combines our base extensible architecture description language, xADL, with another extensible security policy language, XACML. This formalism is suitable for security design and analysis. We have illustrated this by designing two significant in-house applications, a secure coalition application and a secure file sharing application, and by analyzing two large-scale third party applications, Firefox and DCOM. These applications adopt different types of access control policies. Our validations demonstrate that our approach can be utilized for a wide range of security properties.

An algorithm for checking whether the architectural model maintains proper access control at design-time. Based on the modeling of access control properties at the architectural level, we have developed an algorithm that can check whether a componentized and networked software system violates specified security policies for the constituents and the architecture at design-time.

A novel approach to constructing secure connectors. Our treatment of access control at the architectural level is centered around connectors. We provide one type of secure connector that can securely route events to appropriate components when used in event-based software, like the C2 style. This connector has been used in one of our significant case studies, the secure coalition application. We also provide one type of composite connector that can achieve the conjunction of the access control properties of its constituent connectors. This type of connector has been used in another case study, the Impromptu file sharing application. Our treatment of connectors extends existing understanding and techniques of connectors and provides techniques and notations for handling richer connector semantics.

A suite of usable tools to design and analyze secure software. We supply a suite of usable tools to support our approach. We have extended our base architecture development environment, ArchStudio, with editors that allow architects to design access control properties and analysis tools that can check the proper access control. We have provided a framework that can be used to write secure software in the C2 architecture style. We have also provided run-time support tools for executing software written using this framework. Both the framework and the run-time support tools have been used in developing the secure coalition application.

1.5 Overview of Dissertation

The remaining part of the dissertation is organized as following.

Chapter 2 introduces the basic notions of security, discusses the classic security models, and surveys how previous software mechanisms have attempted to describe, analyze, and enforce security with a focus on component specifications and architectural approaches.

Chapter 3 defines the basic concepts of architectural access control, gives an overview of the proposed secure architecture description language, Secure xADL, discusses the central roles that connectors play in our approach, establishes the different architectural contexts that are involved in making access control decisions, and presents an algorithm that can check whether the intended architectural access should be granted within the given contexts.

Chapter 4 extends the previous chapter to handle large scale access through the role-based access control model, to handle heterogeneous access through trust management, and to handle content-based access among interfaces. It also outlines how dynamic architectural execution, including instantiation, connection, and message routing, can be controlled.

Chapter 5 illustrates the tool support. It first presents our implementations of the evaluation engines for the various access control models. After presenting an overview of the base architecture development environment, ArchStudio, the chapter elaborates on how design-time and run-time support is built into this environment.

Chapter 6 presents four case studies to illustrate and help validate the hypotheses: the development of the C2-based secure coalition application and the Impromptu file sharing application, as well as the analysis of Firefox and DCOM.

Finally, Chapter 7 summarizes the research and discusses future work.

2 Background and Related Work

In this chapter we introduce the basic notions of security, discuss the classic security models, and survey how previous software mechanisms has attempted to describe, analyze and enforce security, with a focus on component specifications and architectural approaches.

2.1 Security Overview

Because security is a very broad subject, this section only gives a brief overview of basic security concepts. For other security topics, Bishop provides a comprehensive and recent overview [14].

The main security properties are **confidentiality, integrity**, and **availability** [90]. Confidentiality refers to that there is no improper information disclosure. Integrity refers to that there is no improper information modification. Availability refers to that there is no improper denial of service to legitimate users. The focus of this research is integrity, which is usually enforced by controlling access to protected resources.

The terms of **security policy**, **security model**, and **security mechanism** are defined as follows. Security policies define what rules are to be enforced and what goals are to be achieved. A security model provides a formal representation of security policies. Security mechanisms are hardware devices and software functions used to implement security policies [122].

The most basic type of security mechanism to enforce secure access, solidly established ever since the seminal work of Anderson [4], is a **reference monitor**. The reference monitor is a **trusted computing base** (TCB) that is trusted to intercept every possible access from external subjects to the secured resources and assure that the access does not violate any established policy. Widely accepted practices require a reference monitor to be tamper-proof, non-bypassable, and small. A reference monitor should be tamper-proof so that no alteration of it is possible. It should be non-bypassable so that each access is mediated by the reference monitor. It should be small so that it can be thoroughly verified. A more comprehensive and deeper treatment of reference monitors can be found at [14].

Security policy composition, which occurs when multiple sub-policies coming from different sources are combined into an integral policy, has been extensively studied [19, 66]. The study has investigated questions such as what operations are available, how to decide whether to grant or reject an access request, and how to resolve conflicts between sub-policies.

2.2 Security Models

There are different types of security models. Two most common types are access control models and information flow models.

2.2.1 Access Control Models

Two dominant types of access control models are **discretionary access control (DAC)** models and **mandatory access control (MAC)** models. In a discretionary model, access is based on the identity of the requestor, the accessed resource, and the permission that the requestor has on the resource. The permission can be granted or revoked at will by the owner of the resource. However, in a mandatory model, the access decision is made according to a policy by a central authority.

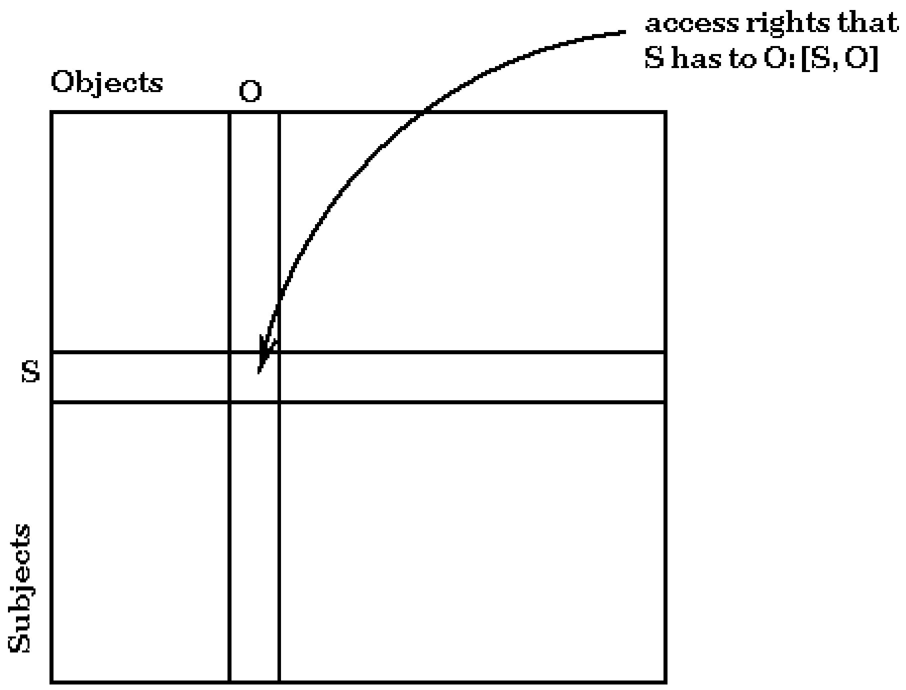


Figure 2-1, Access Control Matrix

The Access Matrix Model, depicted in Figure 2-1, is the most common discretionary access control model. It was first proposed by Lampson [79] and later formalized by Harrison, Ruzzo, and Ullmann [55]. In this model, a system contains a set of **subjects** that has **privileges** (also called permissions) and a set of **objects** on which these privileges can be exercised. A privilege is usually a permission to perform an action on an object. An access matrix specifies what privileges a subject has on a particular object. The rows of the matrix correspond to the subjects, the columns correspond to the objects, and each cell lists the allowed privileges that the subject has over the object. The access matrix can be implemented directly, resulting in an authorization table. More commonly, it is implemented as an **access control list (ACL)**, where the matrix is stored by column, and each object has one column that specifies privileges each subject possesses over the object. A less common implementation is a **capability**

system, where the access matrix is stored by rows, and each subject has a row that specifies the privileges (capabilities) that the subject has over all objects.

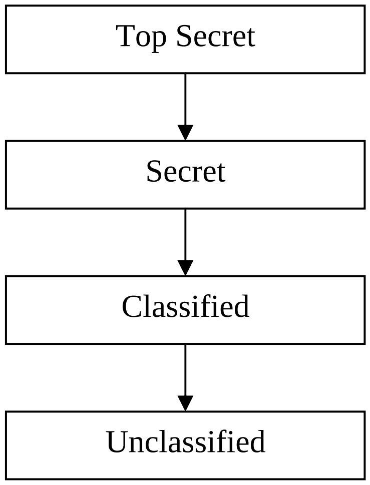


Figure 2-2, Dominance Lattice

Mandatory Access Control models are less common and more stringent than discretionary models. They can prevent both direct and indirect inappropriate access. The most common types of mandatory models work in a **multi-level security (MLS)** environment, which is typical in a military setting. In this environment, each subject (on behalf of a user) and each object is assigned a security label. The labels have dominance relationship between each other, forming a lattice [30]. For example, in Figure 2-2, the label “top secret” dominates the label “secret”, the label “secret” dominates the label “classified”, and the label “classified” dominates the label “unclassified”. The label on an object specifies the sensitiveness level of the information, and the label on the subject identifies the clearance and trustworthiness that the subject has. The subjects/objects with a dominating label are at a higher level, and the subjects/objects with a dominated label are at a lower level.

The most famous MLS MAC model, which is a model for confidentiality, is the **Bell-LaPadula model** [6]. The model specifies two rules that must be satisfied by each access to protect confidentiality: 1) no read up (originally called simple security): a subject is allowed reading an object only if its security clearance dominates the security level of the object. That is, the label of the subject is over the label of the object in the lattice. Thus, a low-level subject cannot read a high-level object. 2) no write down (originally called *-property) : a subject is allowed writing to an object only if its security clearance is dominated by that of the object, so a high-level subject cannot write to a low-level object (to leak more sensitive information intentionally or unintentionally). These rules prevent confidential information of sensitive objects from flowing to less trustworthy subjects.

Another important MLS MAC model is the **Biba model** [10]. This is a model for integrity, and can be considered as a mathematical dual of the Bell-LaPadula model. The model assigns an integrity label to each subject and object, as the confidentiality label of the Bell-LaPadula model. The Biba model has two principles. The first is “no read down”: a subject can only read an object whose integrity label dominates its own so it can trust the integrity of the object. The second is “no write up”: a subject can only write to an object whose integrity label is dominated by its own so it won’t violate the integrity of the object. These rules prevent information stored in lower level and less reliable objects from flowing to and affecting higher level and more reliable objects.

Both the Bell-LaPadula model and the Biba model work in a static environment, where the security labels of subjects and objects change little, if any. The **Chinese Wall model** [20] can be considered as a dynamic mandatory access control model. In this model, objects are assigned to different domains. Each domain represents its own interest, and its interest potentially conflicts with those of other domains. Initially, a subject can access any domain initially. However, once it is granted access to a domain, it is prohibited access of any other conflicting domains thereafter. It is essentially limited within the wall of its own domain. The Chinese Wall model is a model of dynamic separation of duty, and can be mapped to the Bell-LaPadula model if dynamic security labels are allowed in the Bell-LaPadula model [118].

Both the Bell-LaPadula model and the Biba model originate from a military setting. They do not fit well in a commercial environment. The **Clark-Wilson model** [23] summarizes many common security rules practiced in commercial activities. It defines four basic criteria that require authenticating all subjects, auditing all activities, allowing only well-formed transactions, and separating duties. The model has two types of data items and two types of procedures. Data items are either constrained data items or unconstrained data items. Procedures are either integrity verification procedures or transaction procedures. Constrained data items are the items whose validity is verified by Integrity Verification Procedures. These data items can only be changed by Transaction Procedures. The model also requires that administrators must certify all procedures and the system should enforce these procedures. This model is not as formal as other models, though. It is not easy to analyze and enforce.

2.2.2 Information Flow Models

Mandatory Access Control models can prevent overt channels that allow inappropriate information flows, but they are still vulnerable to covert channels where an information flow exists in a clandestine manner utilizing stealthy storage or timing facilities [78]. **Information Flow Models** are confidentiality models that are also called secrecy models. These models are interface models that specify how the information should or should not flow between subjects so that there are no covert channels. They do not suggest how this can be achieved [90].

There have been many proposals of different information flow security properties. Most of them adopt a trace-based viewpoint. In these models, subjects are usually called agents. Agents are classified into two categories: low level agents and high level agents. A trace is inputs received and outputs generated from these agents. The focus of an information flow security model is to prevent low level agents from receiving any secret information from high level agents.

The first information flow security property proposed is Non-Interference [49], which requires low level output should not be affected by high level input. This assures that a low level agent cannot get information about the high level inputs.

Other properties have also been proposed. Non-Deducibility on Input [136] utilizes information functions to require that low level agents cannot deduce information about high level agents. Restrictiveness [87] requires that low level agents cannot differentiate between possible states after certain state transitions. Correctability [68] requires that a trace after a perturbation (adding or removing an input) and a correction (adding or removing an output) is still a valid trace. Non-Deducibility on Strategy [147] specifies that a low level agent cannot tell a high level agent from a process formed from the composition of the high level agent and a strategy, where a strategy is a process that computes inputs to the high level agent based on previous histories.

These models can be applied differently, depending on whether the secrecy is intended for high-level inputs only or both inputs and outputs, whether synchrony is required, whether non-determinism is allowed, and whether probability, instead of possibility, is considered.

However, the programming language community looks at the problem of information flow security in a different manner [120]. Instead of focusing on prevention of any possible information flow, a less stringent but more realistic approach is taken to track the more explicit information flow. An example is given by Sewell and Vitek [129], where an intentional approach for information flow security is proposed, unlike the traditional extensional trace-based approach. This intentional approach assigns an agent "colors", which designate subjects that have causally affected the agent. The colors can be considered as the type of the agent, and a type theory calculus is used to check the validity for information flow security.

2.3 Formal Foundations for Composition

In this section we survey how the problem of describing and analyzing security property for composite software systems is handled by the formal methods community. The definitions and theorems form the theoretical foundation for further study.

2.3.1 Abadi-Lamport Composition in Alpern-Schneider Framework

In the formal method field, the theory of Abadi and Lamport serves as the foundation for composition. While the theory can deal with integrity adequately, it is insufficient for confidentiality.

Abadi and Lamport proposes a general composition principle and a proof rule that compose concurrent specifications in a modular manner [1]. The composition works within the safety/liveness framework first proposed by Alpern and Schneider [3].

In this composition framework, a state is represented by assignments to state variables. A trace is a set of state transitions caused by agents. A system specification describes all possible traces of the system. A property is a predicate that defines a set of traces. A property can also be viewed as the set of traces thus defined. There are two types of properties. A safety property defines the initial state and valid state transitions. A liveness property (also called progress property) specifies that the state transitions eventually occur. The specification of a system consists of the conjunction of various safety and liveness properties. Because systems, properties, and specifications can all be viewed as sets of traces, a system satisfies a property if the set of traces for the system is a subset of the traces for the property. The environment in which the system behaves can be specified in a similar manner, and a system's specification is valid only when the environment satisfies its constraints.

Reasoning about composite behaviors under this framework comprises two steps. The composition step uses the proof rule to establish under what conditions the properties of the subsystems can be connected together in the composite environment. The refinement step finds a mapping under which the conjunctions of subsystem properties will imply the composite property. Informally, a composition decides when subsystems can be composed together, and a refinement ensures the composed system implements the needed composite system.

The Abadi-Lamport composition/refinement rule provides a solid foundation for the general divide-and-conquer approach. However, because security properties are not functionalities, these properties are not preserved by standard notions of refinement or composition. This results in that assurance gained from formal proofs at one level of abstraction cannot necessarily be transferred to a more concrete level [92]. The reason, suggested in [91], is that general functional properties are sets of traces. Security properties, on the other hand, are sets of sets of traces, or power sets of traces. It is believed that luckily integrity, and hopefully availability, is mostly preserved under refinement and composition. However, confidentially is generally not preserved [125], because refinement into components can bring new chances of interaction and observation that are not possible in a monolithic system. This makes the security composition problem a hard problem.

2.3.2 Integrity

There have been many efforts that use Abadi-Lamport theory to directly verify security. Generally the security under consideration is integrity, and the problem will be reduced to prove the safety and liveness of the system. Some prominent examples found in literatures are summarized below.

Heckman and Levitt verifies the correct enforcement of access control policies by a set of distributed servers [56]. The verified system consists of two server processes, each implementing one system call. Both the safety property and the liveness property of the composite system are verified. A Higher Order Logic theorem prover is used to assist the proof. Of the 23000 lines of code for the proof, about 7% is about composition proof, 24% is for the refinement of safety, and 69% is for the refinement of liveness.

Hemenway and Fellows apply the composition theorem with the Formal Development Methodology tools [57]. A system consisting of a workstation, the IPC communication, and the network communication is modeled. The enforcement of a mandatory access control policy is verified.

Bieber uses a state machine to model the imperative properties and adopts temporal logic to describe declarative properties [13]. Even though he tries to handle information flow properties, the approach still mainly verifies safety.

Composability for Security Systems (CSS) [107, 108] is another logic-based method to reason about security of components and their composition. It uses PVS [109] to prove theorems, with a custom developed proof strategy. It mainly investigates integrity of composite systems.

The features of the CSS framework are: 1) it makes agents, which performs actions, explicit to support security analysis; 2) composing components will invoke environmental constraints automatically; 3) it does not support quantifiers, simplifying proofs at the cost of some expressiveness.

The CSS framework provides two lessons for using logic in security verification. The first is the elimination of state translators. Previously a translator between the states of components and the state of their composition was employed. This complicated the property proof. CSS instead uses a single common state that has a field for each component state. A theorem about the configuration of the system is also added. Both the common state and the configuration theorem simplify the proof. Secondly, they discover that a refinement proof is easier to perform than a property proof. To prove a lower level specification is secure, instead of following the more difficult route to prove the property on the specification itself directly, it is easier to first prove the security on a higher-level specification and then prove that refining from the higher level specification to the lower level specification preserves the security. This is a common theme in logic-based security design and analysis approaches [29, 48].

The CSS framework is used to prove that a file manager always returns a secure file handle to a process manager [110]. The components are developed and different approaches to compose them are investigated to compare the tradeoffs of different architectures. The effort confirms that first proving the properties on the components and then proving a refinement mapping between the system and the components is easier than directly proving the composite property on the system. The effort also argues that this route can reuse existing proofs in proving newer properties.

The techniques enumerated above demonstrate the effectiveness of the Abadi-Lamport theory. However, these examples also illustrate how labor intensive the verification activity can be, even for small problems. These approaches also require highly skilled professionals with special expertise.

2.3.3 Confidentiality: Information Flow Security

As discussed before, confidentiality cannot be sufficiently treated in the Abadi-Lamport composition. Researchers took a different path towards this property. They have proposed frameworks unifying information flow security properties and have studied composing these properties under the frameworks.

Unifying Framework. The various information flow security properties listed in Section 2.2.2 have been proposed with different intentions. These properties operate under different formalisms, making comparison among them difficult. There have been many efforts to unify these properties under a single formal framework so that the properties can be compared, deeper insights can be gained, and a consensus on which property is the most desirable might be reached. A unifying framework can also provide a more solid foundation to study the composition of these properties under different operations.

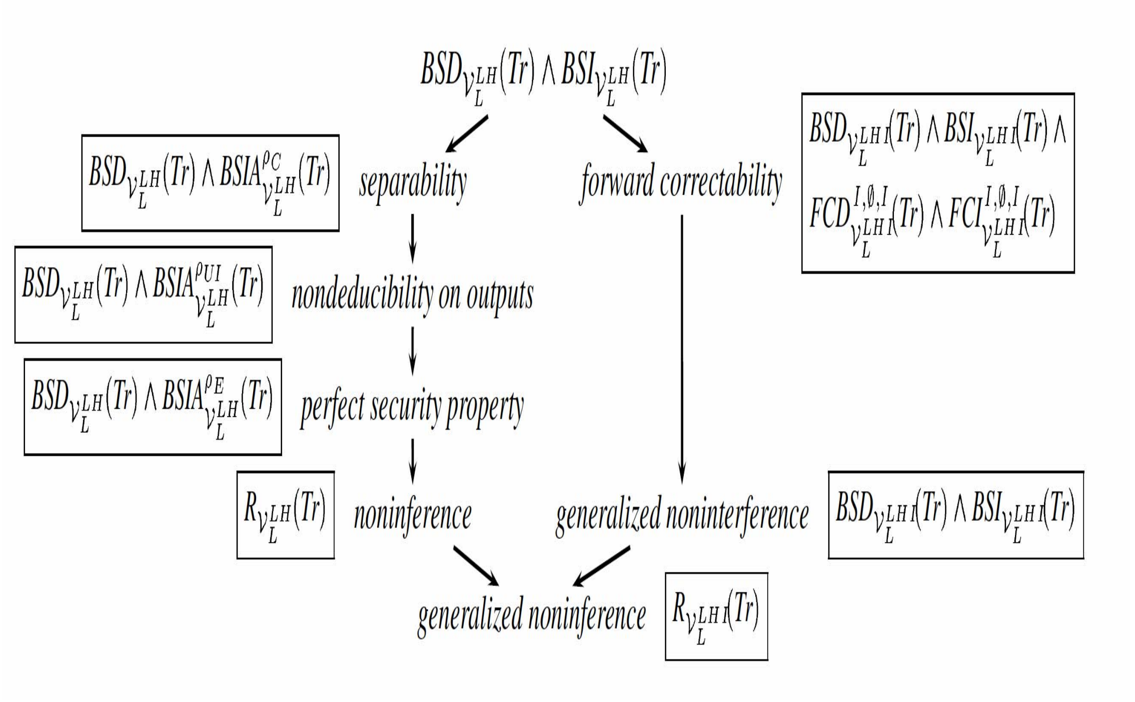
Naturally, most unifying frameworks are based on trace and logic because these are used for defining most of the properties originally. Four representative frameworks are outlined here. These efforts lay down the foundation to study securely composing abstract computations for confidentiality.

John McLean proposes the first such framework, Selective Interleaving Function (SIF) [89, 91]. It views each information flow security property as a function that takes two traces and interleaves fragments of these traces to generate a new trace. Different properties can be described using corresponding functions that takes related fragments and perform appropriate processing on the first and the second trace. A partial ordering among the proposed properties is established, based on the implication relationships between the equivalent functions of these properties.

Peri et al. suggest a simple unification framework based on the many-sorted logic [111]. They study a limited set of proposed properties with the logic and restate the properties using formulas of the logic.

MAKS is another concise unifying framework [85]. Its basic building blocks are Basic Security Predicates. A predicate can be Removal (R), Backward Strict Deletion (BSD), Backward Strict Insertion (BSI), Backward Strict Insertion of Admissible Events (BSIA), Forward Correctable Insertion (FCI), and Forward Correctable Deletion (FCD). These predicates describe operations available on traces. MAKS proves that existing properties can be constructed from these predicates. The implication relationship between the predicates can be used to order the corresponding security properties. The result is illustrated in Figure 2-3.

Halpern and O'Neill uses a modal logic of knowledge to unify the various properties [53]. Their framework models states of both the agents and the environment. The framework extends the notion of Non-Deducibility on Input [136] in several aspects. Firstly, its notion of secrecy allows asymmetric secrecy from one agent to the other, unlike the symmetry of the original definition. Since the secrecy is modeled as knowledge, it can be more specific on what is to be guarded, relieving the requirement that everything is a secret. Secondly, its notion of a trace (called Run in the framework) makes time more explicit. It introduces an allowability function based on time that can uniformly handle complete synchrony, complete asynchrony, and any middle points between the two extremes. Thirdly, it also introduces a probability measure to handle probabilistic secrecy. This measure can be either a global measure on all possible runs, or a locally defined one on partitions of runs. In addition to these extensions, using model logic of knowledge also enables the framework to model resource-bound adversaries where revealing of secrecy is computationally expensive.



$BSD_{\mathcal{V}_L^{LH}}(Tr) \wedge BSI_{\mathcal{V}_L^{LH}}(Tr)$
$BSD_{\mathcal{V}_L^{LH}}(Tr) \wedge BSIA^{\rho C}_{\mathcal{V}_L^{LH}}(Tr)$
separability
forward correctability
$BSD_{\mathcal{V}_L^{LHI}}(Tr) \wedge BSI_{\mathcal{V}_L^{LHI}}(Tr) \wedge FCD^{I,\emptyset,I}_{\mathcal{V}_L^{LHI}}(Tr) \wedge FCI^{I,\emptyset,I}_{\mathcal{V}_L^{LHI}}(Tr)$
$BSD_{\mathcal{V}_L^{LH}}(Tr) \wedge BSIA^{\rho UI}_{\mathcal{V}_L^{LH}}(Tr)$
nondeducibility on outputs
$BSD_{\mathcal{V}_L^{LH}}(Tr) \wedge BSIA^{\rho E}_{\mathcal{V}_L^{LH}}(Tr)$
perfect security property
$R_{\mathcal{V}_L^{LH}}(Tr)$
noninference
generalized noninterference
$BSD_{\mathcal{V}_L^{LHI}}(Tr) \wedge BSI_{\mathcal{V}_L^{LHI}}(Tr)$
generalized noninference
$R_{\mathcal{V}_L^{LHI}}(Tr)$

Figure 2-3, Information Flow Properties, from [85]

Some unifying frameworks based on process algebras are also suggested. Process algebras are compact, can express composition naturally, and can handle situations where traces on inputs and outputs are insufficient. For example, process algebras can specify that a low level agent should not get any information by observing a high level agent being deadlocked. This is a possibility that is not addressed in trace-based formalisms.

Security Process Algebra (SPA)[40, 41, 43] is a security extension to the process algebra Calculus of Communicating Systems (CCS) [100]. It views various definitions of information flow securities as requirements on the processes, and uses equivalence relations to classify those properties based on their implication relationships. It uses trace equivalence and test/failure equivalence to classify existing properties, and proposes behavior equivalence as a stronger definition of equivalence. The behavior equivalence is based on weak bisimulation of processes, where processes are equivalent if they can accept the same nondeterministic events. Based on this notion of equivalence and the definition of Non-Deducibility on Strategy [147], SPA proposes a new security property, Bisimulation Non-Deducibility on Composition, where a high level agent can compose with a general process.

Ryan and Schneider applies a different process algebra Communicating Sequential Process (CSP) [62] to unify information flow properties [119]. They eliminate the difference between inputs and outputs, viewing them as just events. They use power bisimulation to unify those properties. Power Bisimulation is a different equivalence than the weak bisimulation used in the Security Process Algebra.

Composition. The composition problem has received significant attention within the information flow security community. The general question to be answered is: given a component with one property and a component with potentially different properties, when they are composed using one composition construct, what property will the composite system satisfy [91]? A simplified version is: when two components with one property are composed using a particular composition construct, will the composite system also satisfy that property? If yes, then it can be said that the property is compositional (composable) under that composition construct.

The notion of composition depends on the formalism adopted. Selective Interleaving Function classifies composition into three different constructs [91]. In a product composition, two components are juxtaposed, without any interaction. In a cascade composition, one component's output is fed as another component's input. In a feedback composition, in addition to the input/output relationship established in cascade, the output of the second component is also the input of the first component, forming a loop between the two components.

Some representative results from studying composition under these constructs are summarized below. It is proved in [91] that the feedback composition retains less security properties than the product composition and the cascade composition, because it is too restrictive on what to accept and too generous on what to produce. MAKS only considers product composition and cascade composition [85]. It uses a powerful lemma to unify known composition results. MAKS reveals why certain properties cannot hold under composition and suggests what emergent behaviors (behaviors that only exist in a composite system) can emerge under composition. Zakinthinos proposes a simple bunch-theory based framework, where a bunch is the content of a set [148]. The framework studies both cascade and feedback and discovers that properties eliminating dependencies on inputs are preserved under feedback composition. Peri et al. [111] study the composition problem under the many-sorted logic and prove compositional properties in cascade and feedback composition using PVS [109].

Composition takes a different form in Security Process Algebra [41]. It is formed by the parallel execution of processes. These processes only synchronize on common complementary actions when one process's output is another's input. The algebra studies whether certain properties can still hold when the restriction operator and the hiding operator applies on the composition operator. The Bisimulation Non-Deducibility on Composition property has to be extended to its strong variant to be composable. A model checking tool, compositional security checker [42], is used to check the compositionality of security. The power bisimulation proposed in [119] is also composable.

Santen et al. views the compositional problem under the refinement/composition perspective [125]. They argue that traditional possibilitistic secrecy is too strong, requiring too many sufficient conditions and providing too few necessary conditions. They suggest that in a refinement setting, if a concrete specification preserves the same probability of discovering secrecy as an abstract specification, then it is a secrecy-preserving implementation of the abstract specification. Santen et al. discovers that failing to hold security under composition comes from the new window of observation opened up by decomposing a system into components.

Discussion. The information flow security property captures a natural notion of secrecy. Despite its general appeal and two decades of research for it, the topic remains mostly of an academic interest [117]. In real systems, high-level agents do interact with low-level agents. Even among researchers, there is no universally accepted consensus about what is the best definition and formalism to characterize the information flow security property. This can be seen from the many proposed properties and even more frameworks unifying them. These properties are too remote from a real system and few real policies care about information flow security. The composition mechanisms are very primitive and far from real connection facilities. Finally, information flow security models are very difficult to build. Their canonical definitions took a form of an inductive or a universally quantified format, which is not constructive at all. It may be necessary to retreat to building a traditional access control model first and performing covert channel analysis afterwards [90, 99]. As suggested in [117], "non-interference is little more than a rather intriguing topic of arcane debate, at best the source of compelling theoretical challenges on which learned but largely irrelevant papers can be written." In spite of its appeal and abundance of mathematically beautiful results, information flow security might not be very relevant and practical for real software.

2.4 Component Specifications of Software Security

After surveying the theoretical results on security, we will discuss two types of software techniques that describe, analyze, and enforce security for componentized and networked software systems in the remaining part of this chapter. This section focuses on how a component can be specified for its security requirements and provisions. The next section elevates the abstraction to the architectural level and surveys related approaches. For a more comprehensive discussion of software mechanisms utilized in complex systems to handle modular security, please see [113].

This section investigates techniques that support explicit component security specification. During composition, these specified components should be combined consistently, resolving potential conflicts and accomplishing system wide security.

2.4.1 Computer Security Contract

Computer Security Contract (CSC) addresses how to disclose the security property of a component to others [72, 73]. It tries to answer the following questions: how to characterize the security properties of a component, how to access these properties at runtime, how to characterize the composite security properties when a system is composed out of several components statically or dynamically, and whether the composite properties are also available at run-time.

Computer Security Contract explicitly specifies security properties of component interfaces. The interface specifies ensured and required security properties of a component using logic. When the components are composed together, a composite logical description is deduced to capture the ensured and required properties of the composite component. These properties can be accessed at run-time. An interface with reasoning capability and knowledge storage is named Active Interface.

The basic form of the logic is an atom describing three items: the security operation, the security credential used in the operation, and the data operated by the operation. For example, an encryption operation takes a key as the credential and a stream of data for encryption.

The CSC framework operates in an event-based environment. When a component needs a service, it broadcasts a request, and becomes the *focal* *component*. A *candidate component* is the component responding to this request. If the two components can successfully negotiate and find a way to satisfy the required security properties of each other, then a binding is established between the two components, forming a composition. The composite contract is the composition of the contracts of the two components, with the required property of the candidate component as the composite required property, and the ensured property of the focal component as the composite ensured property. After a successful negotiation, both the focal and candidate component reconfigure them to behave as specified by the contract.

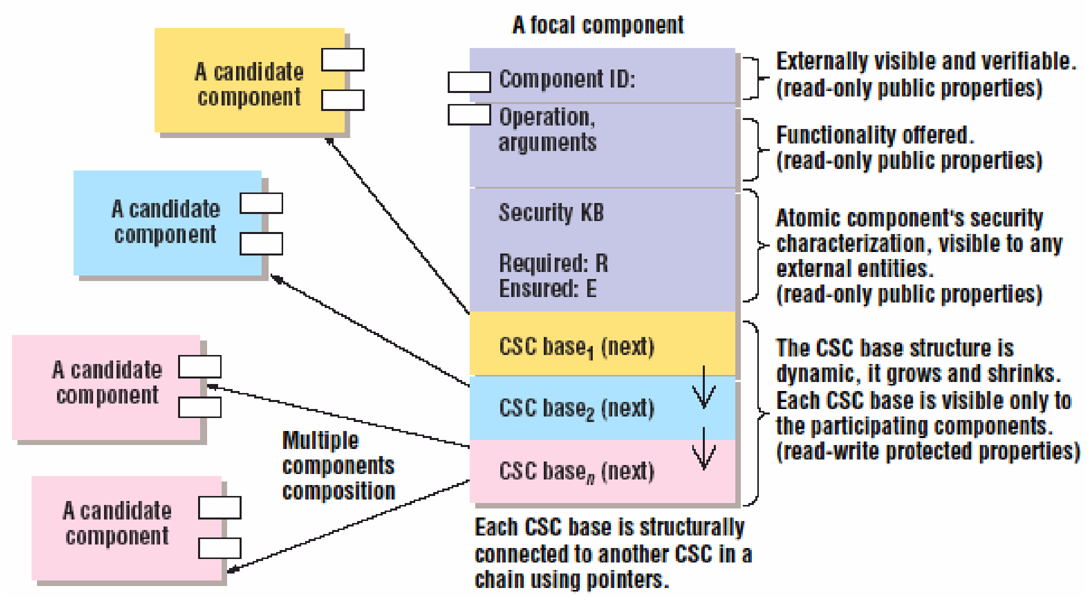

A focal component
A candidate component
A candidate component
A candidate component
A candidate component
Component ID:
Operation, arguments
Security KB
Required: R
Ensured: E
CSC base$_1$ (next)
CSC base$_2$ (next)
CSC base$_n$ (next)
Externally visible and verifiable. (read-only public properties)
Functionality offered. (read-only public properties)
Atomic component's security characterization, visible to any external entities. (read-only public properties)
The CSC base structure is dynamic, it grows and shrinks. Each CSC base is visible only to the participating components. (read-write protected properties)
Multiple components composition
Each CSC base is structurally connected to another CSC in a chain using pointers.

Figure 2-4, Active Interface, from [73]

To enable the run-time access of security properties described by composite contracts, each component has an interface called the Active Interface. The interface consists of an identifier verifiable through a digital certificate, a traditional functional interface describing the available functions, a read-only public security knowledge database providing the ensured and required security property of the component, and a read-write protected computer security contract base containing all the active contracts that the component is currently bound to as a focal component. The contract base will expand and shrink, as the component engages in different compositions. However, each candidate component bound to a focal component cannot see the contract of other candidate components, providing a protection among the components. The structure of the active interface is shown in Figure 2-4.

The logic-based contract is expressed with a Prolog-like form of logic programming [74]. A contract has a set of rules each of which has a header and a body. The header is a predicate that can be derived if all predicates in the body are satisfied. An ensured property is a rule containing only a header. A required property is a rule containing only a body. A compositional contract is the result derived from the rules of the components. Logic programming allows more powerful automation and reasoning. A rule can use predicates from the authentication logic proposed in [21]. The authentication logic reasons about the authentication and belief relationships among components, and provides a well-established foundation to from a compositional security property from component contracts.

In summary, the Computer Security Contract approach extends the traditional functional interface of a component with an extra-functional interface about required and ensured security properties. A logic approach is used to describe these properties. Logic reasoning is utilized in negotiating a composition of components and determining the composite security properties. A run-time structure provides storage and access of these properties.

While this approach is promising, some issues need to be resolved. Firstly, a more expressive and efficient expression mechanism is needed. The current basic atom describing security operations, credentials and data does not capture most entities involved in security design and analysis. How to improve its expressive power yet retain its computation efficiency is still an open research question. Secondly, the current composition mechanism is still rather simple, mirroring a function call between a caller and a callee. Existing logics on functional composition can be applied to this composition mechanism. Other composition mechanisms, possibly involving more than two entities, need to be incorporated [75]. Thirdly, the current contract base is stored at the focal component and requires modifying the component, so it depends highly on one party of the component. Whether this is the only or the best choice is arguable. When a general component container is used, it might be a better place to serve as the composite contract base. Other forms of composition might choose different places to store security contracts.

2.4.2 cTLA Contract

Hermann proposes a more elaborate component specification to describe and verify security properties of component-based systems [60]. Instead of the simple first order predicate logic used in the Computer Security Contract, a compositional extension to the Temporal Logic of Actions (TLA)[77], cTLA, is used to specify the behavior contract of components and their compositions.

The cTLA is a linear time temporal logic describing the safety and liveness properties of systems (see Section 2.3.1). The contract written in cTLA models each component as a process and delineates the state transitions of the process for the component, forming a state machine. The state machine can be used to enforce security properties, allowing valid state transitions and prohibiting invalid ones, as described in [127].

The composition feature of cTLA is based on concurrent execution of processes. cTLA enables composition from implementation-oriented processes, constraint-oriented processes, and processes combining both. The composition feature of cTLA supports the property of superposition, where a property of a process is also a property of the embedding system.

The superposition of composition greatly simplifies the verification of compositional systems. The verification can utilize a pre-developed framework containing theorems about shared global settings and the properties of constituent components. To prove a more concrete system holds the same property as a more abstract system, a correspondence between a process in the latter and a component in the former should be established, most probably in the form of a refinement mapping.

A Role-based Access Control policy is modeled as cTLA processes. The validity of the access control policy of an e-commerce procurement application is verified using the refinement mapping technology suggested above. That experience suggests that a refinement mapping is relatively easy to find, and much of the verification work can be automated with tools.

Compared to the Compositional Security Contract [73] (see Section 2.4.1), cTLA does not focus on what a compositional contract will be when composing components, and how a run-time system can support reasoning, storage, and utilization of this contract. Composition Security Contract is a bottom-up approach. cTLA is another instance of those top-down logic-based refinement verification methodologies [36, 134]. Despite the initial positive experience, the approach faces the same challenges, namely finding the suitable security properties for the methodology and effectively conducting the proof with more automation and less dependence on human experts.

2.4.3 Discussion

The techniques proposed in this section are only a sample of possible alternatives. They stand out by their explicit use of logic-based component specification. Using logic facilitates automatic reasoning and proving during composition and refinement. One issue beyond simple composition is the emergent property problem. Emergent properties are those properties that only come from composing components. Undesirable emergent properties might be the result of under-specification of the components or implicit assumptions made by the components. Specifications of components should be complete so no undesirable properties will emerge during composition [61, 149]. Desirable emergent properties are also challenging. An open research question is whether a set of secure components can be composed to achieve more security that what is available through a single component [32] and how this can be accomplished.

A problem with the component specification approach is how trustworthy the specification is, because there might be no proof that the real behavior of the component is the same as that specified in its specification. One possible mitigation is using certification [47]. Some trusted third party can certify the conformance between the specification of a component and its underlying behavior and issue a certificate difficult to forge to the component. The certificate can easily be verified during composition. This is not a complete solution, but it can be part of the foundations to support secure composition of components.

2.5 Architectural Approaches to Software Security

Software architecture has been proposed as an effective method to design and analyze large and complex software systems. Most of the previous work has focused on functionality. This section will examine its support for security. Some questions specific to an architectural approach are: Does the technique employ a formal architecture model? If there is a formal architecture model, are connections between components buried in an ad hoc manner, or are the connections abstracted as first class connectors? If connectors are used, how do they facilitate the expression and enforcement of security?

This section begins by examining security extensions of standard object-orientated techniques (Section 2.5.1 and Section 2.5.2). It then turns to approaches without an explicit notion of connectors (Section 2.5.3 and 2.5.4). The next discussion is about architectural models supporting explicit connectors (Section 2.5.5 and 2.5.6). The issue of security in architecture evolution is discussed in Section 2.5.7.

2.5.1 Object-Oriented Labeling

Like modeling software architecture with standard object-oriented notations [93], some design techniques extend object-orientated methodologies to support security. Herrmann introduces a methodology to analyze information flow security [59]. The theoretical foundation of the methodology is a decentralized labeling model. The meta-model used in the methodology is the

Common Criteria [22]. To facilitate the adoption of the methodology, a tool based on graph rewrite system is also developed.

A label in the decentralized labeling model [105] identifies a set of principals. One of them is the owner; the others are readers who are granted reading access by the owner. An “act for” relationship can be defined between principals so one principal can have the same reading privilege as the other principal. Operators are defined over labels to generate more restrictive or less restrictive labels. Each component, interface, method, and field of an object-oriented design model is assigned a label. A label serves as an access control policy to define what kind of access is granted to which principal. The decentralized labeling model facilitates static analysis of information flow security for a model so labeled.

The Common Criteria [22] defines a set of classes for concepts utilized in a security evaluation process. An *asset* is a resource needing protection. It has *vulnerabilities*, so it is exposed to *threats*. *Risks* are associated with these threats. *Countermeasures* can be deployed to fight the threats. However, countermeasures may contain vulnerabilities themselves, so more countermeasures are needed. For each asset, vulnerability, risk, threat, and countermeasure, a number is assigned to reflect its relative *value*, *severity*, or *effectiveness*.

A graph rewrite system is a set of rules used in transforming graphs. Each rule specifies a pre-pattern that identifies the graph before transformation, a post-pattern that specifies the graph after transformation, an application function that must be met by the attributes of the original graph, and an effect function that the attributes in the transformed graph will exhibit.

Guided by the meta model of the Common Criteria, the object-oriented labeling methodology assigns a numeric value to each data item described in the object-oriented model. It also labels each component, interface, method, and field to reflect the current access control policy. Using graph rewrite rules, the full access control relationship is computed, so is the asset value of each data structure and data storage component. If some of the more precious assets might be exposed to malicious principals, a threat is identified, and the corresponding risks are assessed. If the risks are within the acceptable range, then the object-oriented model is satisfactorily secure. Otherwise, either the label needs relabeling, or countermeasures should be deployed to attack the threats. The effectiveness of the new countermeasure needs to be reevaluated. Since countermeasures might bring in new vulnerabilities, this process will iterate until the risks fall into a range acceptable to the security assessor.

This methodology integrates formal information flow analysis into mainstream object-oriented design techniques, resulting in a usable approach that can enhance the security of design. Its use of a graph rewrite system can easily integrate more knowledge about security analysis into the design process, if the knowledge can be embodied in a graph rewriting rule.

The assessment on security is reached through a subjective evaluation process, thus the assurance provided by the methodology is at best qualified. Currently the methodology can only utilize one kind of formalism (object structure) and evaluate designs statically. Integrating multiple kinds of formalism (object behaviors) and expanding the evaluation into a dynamic environment is worth pursing.

A similar approach is MOMT [86], a methodology that adds multilevel security to the original Object Modeling Technique. The basic extension is to add a security label to attributes and operations of objects and classes in the static model, and add a security label to the events produced in the dynamic model. The MOMT methodology is not widely used, possibly due to its incompleteness.

2.5.2 UML-based Security Modeling

UML is a standard design modeling language. There have been several UML-based approaches for modeling security. UMLsec [70] and SecureUML [82] are two UML profiles for developing secure software. They use standard UML extension mechanisms (constraints, tagged values, and stereotypes) to describe security properties.

Aspect-Oriented Modeling [112] models access control as an aspect. The modeling technique uses template UML static and collaboration diagrams to describe the aspect. The template is instantiated when the security aspect is combined with the primary functional model. This process is similar to the weaving process of aspect-oriented programming. The work described in [71] uses concern diagram as a vehicle to support general architectural aspects. It collects relevant UML modeling elements into UML package diagrams.

2.5.3 ASTER

Bidan and Issarny proposes one of the first techniques to address security issues using an architecture description language supporting connectors [12]. Based on security requirements of components to be composed, the approach uses the specification matching technique [150] and composes a customized connector out of base connectors and system-provided connectors to connect the components and meet those requirements. These connectors are implicit, though.

In canonical software architecture paradigm, a connector handles communication issues between components. The quality of service of communication, such as security, can be handled via newly formed connectors composed of existing application-level connectors and system connectors [132]. This connector composition approach has the following benefits: 1) separation of concerns: computation, communication, and QoS of communication are handled by different constituent parts of the architecture; 2) limited impact on the existing architecture; 3) assurance of enforceability by the underlying system.

The proposed approach addresses three types of security properties: encryption, authentication, and access control. An encryption specification of a component specifies the parameters of the encryption, such as the algorithm being used, the key size, and the session length. A component might use a set of encryption algorithms and have different levels of trust for each algorithm, with the highest trust on the most secure encryption. Based on the specifications, if two components can each find an algorithm sufficiently trusted and the algorithms are compatible (probably using the same algorithm and accepting keys of the same size), the components are bound together, and the connector will be the most secure connector that can be established between the two components.

A similar process is applied to match the authentication requirements of the components. Each component specifies the authentication protocols that it can use and the level of trust of each protocol. The most trusted protocol that can be mutually applied will authenticate the components.

A different specification is used to specify access control policies [11]. For each component, the specification stipulates the types of subjects (classifications) and the types of access these subjects will be granted (access rules). When composing two components together, the composite classifications can be the union, intersection, or product from the classifications of the components. The composite access rules can be the logical conjunction or logical disjunction of the access rules of the components. Two types of match are defined to compare access control policies: a plug-in match if one policy subsumes the other and an exact match if they are equal.

The ASTER configuration-based environment is extended to compose components having security specifications. The environment is based on a module interconnection language, and it can be used for run-time composition of components.

This approach is among the first to specify security requirements for components and form composition based on the requirements (see also Section 2.4). The approach is supported by a configuration-based design environment. The approach still has the following limitations: 1) The security specification is not very expressive. It is limited to certain aspects of certain properties, such as algorithms of encryption and protocols of authentication. 2) The match of the specifications is primitive. It is mostly a selection process based on parameters of the specifications. 3) Even though the approach argues for composition of connectors, it is still oriented towards module interconnection, lacking an explicit notion of connector that stores and enforces the composite security property. 4) The approach does not directly address how composition can be applied to composite systems.

2.5.4 System Architecture Model

System Architecture Model (SAM) is a methodology that can be used to model and analyze security of system architectures [29]. The methodology models security as a global constraint on the system architecture. It then propagates the constraint down to the components, and verifies that the components satisfy the constraint collectively. The methodology then applies the same process to model and analyze each component individually.

The System Architecture Model (SAM) integrates a model-oriented formalism, Petri net, and a property-oriented formalism, Temporal Logic. Its lower level (proposition level) utilizes Place-Transition nets and Real-Time Computation Tree Logic, so the model can be automatically analyzed. At the higher level (first order level), it adopts Predicate/Transition nets and First Order Temporal Logic, because they are more expressive. The security modeling and analysis is based on the higher level notions. Petri nets describe components and connectors, and Temporal Logic specifies architectural constraints.

The methodology consists of the following steps [29]:

1) Construct a top-level secure system architecture model.

2) Specify system wide architectural security constraint patterns. These patterns are expressed in temporal logic, and they involve only ports of the components.

3) Decompose the system wide security constraint patterns into individual constraint patterns on components.

4) Verify the consistency between the system wide constraint patterns and the component-level constraint patterns. The verification generally is not decidable. However, since the component constraints are derived from the system wide constraints and the architecture connects components together, a smaller Petri net can be designed to replace each component, using conversion guidelines delineated by the methodology. The resultant larger and executable Petri net can be used to verify the consistency between constraint patterns from two different levels.

5) Incrementally design and verify components. Apply the about four steps for each component.

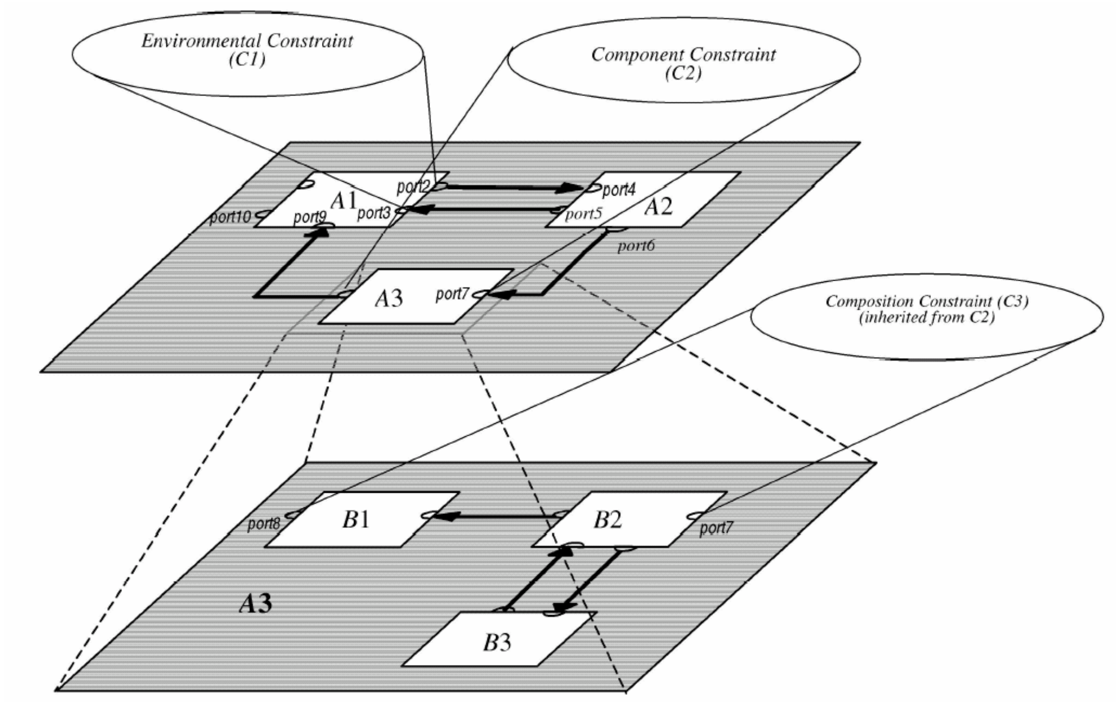

Environmental Constraint (C1)
Component Constraint (C2)
A1
port2
port3
port4
port5
A2
port10
port9
port6
A3
port7
Composition Constraint (C3) (inherited from C2)
port8
B1
B2
port7
A3
B3

Figure 2-5, System Architecture Model, from [29]

The overall methodology is illustrated in Figure 2-5, which shows the environmental constraints and component constraints at the high level, and how constraints on one component are inherited as the composition constraint in the low level.

The SAM methodology is applied to model the Resource Access Decision Facility of CORBA. It is verified that the architecture satisfies the security constraints: the access control decision is always in accordance with the current policy.

This methodology can model the security of a system architecture in a systematic and formal manner. It can assure that a system composed from components satisfies the security requirements. It claims to be one of the first such efforts that model architectural security in a composable and verifiable fashion.

The methodology achieves scalability through the classical divide-and-conquer mechanism. Once the constraints on each component are verified to preserve the architectural constraints, each component can be designed and analyzed separately. As long as a component conforms to its part of the full contract, the global property will not be affected.

The SAM methodology is a top-down approach. It starts with the security requirements of a system, and assigns responsibility to each component, so their composition can be verified for satisfying the requirements. The methodology could not be applied in a bottom-up manner, where the composite security from composing components needs to be reasoned from the security of those components.

The methodology also models security as a form of correctness. It treats security as a property that can be expressed by first order temporal logic. While this can cover a large set of problems, the approach cannot address problems in the covert channel domain. This methodology is an architectural level integrity verification methodology for safety composition and refinement (see Section 2.3.1 and 2.3.2).

In step 3 of the methodology, how to decompose the global constraints into each component is not always straightforward. With a given architecture, there can be several alternatives to allocate constraints. How to decide the tradeoffs of the alternatives is worth exploration. More challengingly, when the architecture is still under design and it can still be changed to accommodate different security property, performing such an allocation and tradeoff analysis becomes even more difficult.

Since the System Architecture Model is based on Petri nets, it does not have the notion of explicit connectors. The "connector" is actually the transitions between places, not the usual notion of communications between computations. Therefore, the methodology does not have a step to incrementally design and verify "connectors". While the temporal logic-based formalism is applicable to other software architecture description languages, extrapolating the Petri net-specific mechanism might not be very straightforward.

2.5.5 Connector Transformation

Given the importance of connectors in architectural development [95], constructing them effectively is of great importance. Handcrafting each connector can be expensive. Existing connectors do not always provide all required qualities.

Like composing general application using existing components, connector composition is aiming at reducing development efforts. Spitznagel and Garlan proposes a set of operators that can be used to transform an existing connector into a new connector that provides required security property [131].

The motivating problem of the approach is to add security property to a generic communication mechanism. In the example given in [131], it is to add Kerberos authentication support to Java Remote Method Invocation. One possible solution is to ask the developer to modify the original application that uses the communication mechanism. This solution is rather expensive, and the result is not maintainable. A second possibility is to modify the generator generating stubs for the communication mechanism so the mechanism provides the security capability at appropriate locations. This method requires expertise of the communication and security mechanisms, and it cannot scale to other properties because a new property will require further modification of the modified mechanism.

The authors propose a solution employing a set of transformations on the original connector to produce a new connector that can meet both the communication and security requirements. A tool can be developed to automate the process. This transformational method lowers requirements on the knowledge about the original communication mechanism. The general transformational method could be applied again on the resultant connector when the mechanism needs to provide other qualities.

The transformation method is outlined in Figure 2-6, where l designates communication libraries, generated stubs, etc., below the application level, s represents low level infrastructure services, t stores data and tables for information like locations of communicating parties, p is a policy specifying the proper use of these parties, and w collects the formal specification describing the connector's proper behavior.

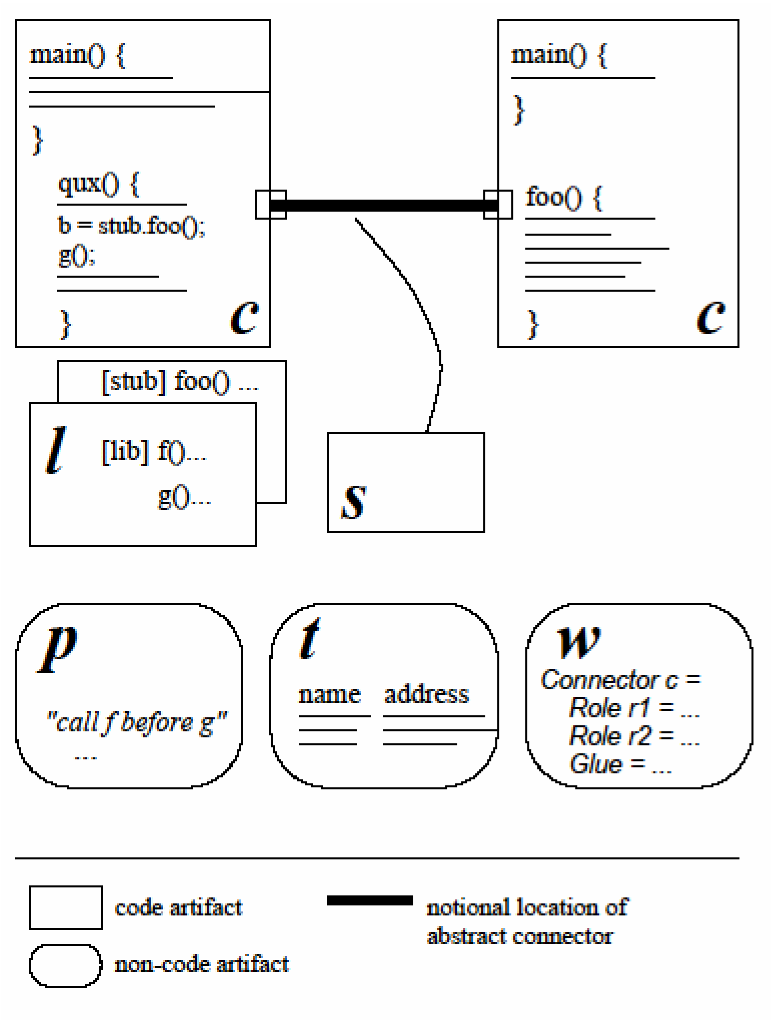


Figure 2-6, Connector Transformation, from [131]

The authors argue that the transformations on connectors should balance between formalism and practice, and the transformations should be useful, general and analyzable. They propose the following transformations for secure communication: *data transformation* that changes the format of data exchanged, *splice* that combines two binary connectors into one new binary connector, *adding a role* that enables adding a new party to the interaction, *session* that makes a stateful connection stateless or vice versa, and *aggregate* that puts a set of connectors under the control of one controller.

The Kerberos support is successfully added to Java RMI after these transformations. The engineering effort involved is reasonable, but the advantages gained are significant.

They admit that their current technology only handles different types of transformations applied on a single type of connector, because a transformation requires knowledge of the specific connector. Finding a set of general transformations applicable to many types of connectors is a great challenge. The current formalism used in describing the transformations is still limited to the specific connector type.

Transformational construction of connectors can be an effective way of providing extra functionality in connectors. However, finding a set of transformations useful, general, and analyzable remains a big challenge.

Connector transformation can be considered as one method to introduce more aspects onto the base communication capability. The aspect methodologies provide a general framework that can handle many different aspects, but not much support specific for security is provided [113]. The connector transformation methodology utilizes a set of transformations useful in supporting security. Which methodology is more powerful and more secure, and whether a combination of both is possible, remain open research issues.

2.5.6 SADL

Architecture Proof. Secure Software Architecture [104] is one of the few approaches that directly deal with security at the architectural level. Based on the correct refinement approach presented in [103], the Secure Software Architecture approach presents three unique features: it supports not only horizontal decomposition of architectures but also vertical decomposition between different layers of abstractions, it maintains a correctness retaining mapping between different layers, and it utilizes a canonical architecture description language that supports property refinement. The approach is illustrated in Figure 2-7.

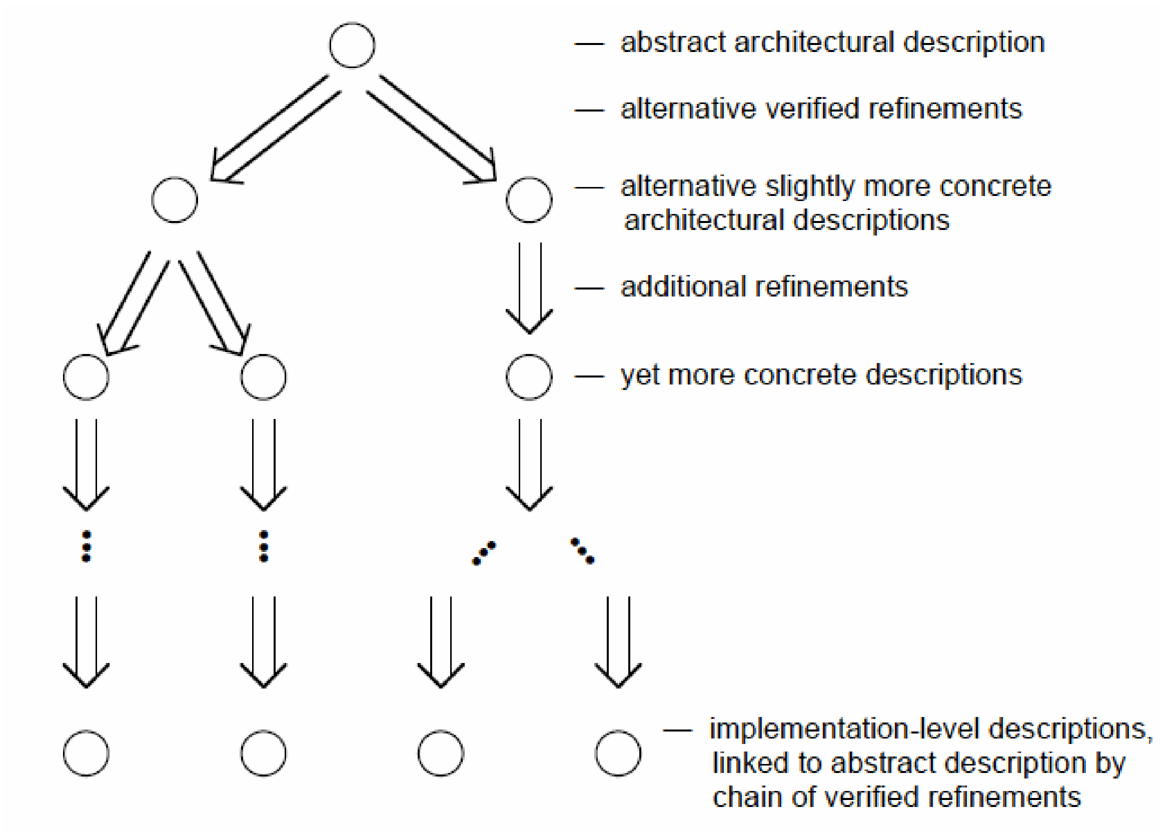

— abstract architectural description
— alternative verified refinements
— alternative slightly more concrete architectural descriptions
— additional refinements
— yet more concrete descriptions
— implementation-level descriptions, linked to abstract description by chain of verified refinements

Figure 2-7, Secure Software Architecture, from [48]

The authors use the approach to prove the Bell-LaPadula [6] security of a secure extension to the X/Open Distributed Transaction Processing standard (SDTP). They argue that proving the security property at an architectural level on a standard has the advantage that any compliant products will possess the same security assurance without further proof. They develop different security extensions to the original architecture and prove that each extension preserves the required security.

In the SDTP proof, the DTP standard partitions a distributed transaction processing system into three components: the application component that is the initiator of the transaction, the resource manager that manages resources of the transaction, and the transaction manager that coordinates the transaction. Three possible architectures that enforce Bell-LaPadula security are: 1) Place all three components into a single security level. 2) Put the application and the resource manager at different levels, connect them through a MLS (Multi Level Security) filter that enforces security, and use a full MLS transaction manager. 3) Use a full MLS application component, a full MLS resource manager, and a full MLS transaction manager. They prove that each architectural variation can preserve the required security.

The reasoning power of the architecture definition language SADL is based on logic. During the refinement process, the mapping established between the higher level abstraction and the lower level abstraction must be both a theory interpretation and a faithful interpretation. That is, a true property at the higher level abstraction is also true at the lower level, and a false property at the higher level is also false at the lower level. In other words, the lower level architecture implements the higher level architecture exactly. This is based on a completeness assumption on that all true statements at each level of abstraction can be derived from the specifications of that level. As will be clear later, this is a rather stringent requirement.

After establishing the mappings between the proposed secure architectures, they manually prove that these mappings actually preserve the security properties.

Implementation. The effectiveness of the architectural refinement methodology is demonstrated by implementing the secure distributed transaction processing (SDTP) architecture proposed above [48]. The demonstration reveals important properties of the methodology.

The most important objective of the implementation case study is to determine whether applying transformations using only faithful interpretations is sufficient to derive the implementation level description from the most abstract descriptions. The non-definitive conclusion from the case study is that it is very difficult or even impossible. A less stringent kind of transformations always preserving security is showed to suffice for the derivation, but it requires very strong preconditions, which severely affects the applicability of such transformations. Eventually the authors have to introduce transformations that do not always preserve security, and they will check to assure that such transformations still retain security in each case. To prove that the transformations still preserve security, they utilize the same transformations used in architectural descriptions to prove the security perseverance of these transformations. They call this notion as “proof-carrying architecture” because of the carrying along of transformations from architecture descriptions. Combining transformations that always preserve security and transformations that can be checked to preserve security together, they accomplish the goal of deriving a low-level secure architecture from an abstract description.

The study also demonstrates that rearchitecting can be an effective method to introduce security. Security is not an inherent property of the original architecture standard. It is an add-on feature after the architecture has been established. The methodology shows how to introduce and verify security on a legacy architecture.

Transformations are a common software production technique. While they cannot achieve everything through a limited set of transformations, they verify the validity of transformations that they believe are generally useful.

Also, the authors can derive the final implementation from the lowest “implementation-level” descriptions straightforwardly, due to the formality of facilities from the selected programming language. The argument for the programming language dependence is that this is necessary to assure no significant gap exists between the lowest level description and the code, and the confidence gained in the transformations and checking is not lost in the final step of software construction.

Discussion. This experience suggests that employing mathematically sound transformations only, such as faithful interpretations or security preserving transformations, is too difficult for practical applications of the methodology. However, loosening the stringent requirements on transformations and checking security after transformations with the connection embodied in architectural descriptions is more effective in verifying the security of the architecture. This is also demonstrated in [29], where verifying the consistency between architectural constraints and component constraints is facilitated by the fact that the latter is derived from the former.

A common obstacle against a transformation and proof-based approach is that it requires significant expertise and is highly labor intensive (see also Section 2.3.2). An automated tool simplifying the application of the methodology is possible, with the insights gained from the effectiveness of rearchitecting, the available stock of general and verified transformations, and the easiness of producing code from low level descriptions,.

The authors plan to use light weight formal approach, design a lot, specify some, and prove just a little [133]. This approach would be more practical than a formal method that requires great efforts from methodology experts.

2.5.7 Law-Governed Architecture

Law-Governed Architecture [101] is a methodology arguing for not only the description of an architecture model but also its enforcement. The benefits of an enforced architecture model are two folds. Firstly, it can bridge the gap between a descriptive architecture and the system, enabling reliable reasoning about the system. Secondly, due to its carefully circumscribed flexibility, developers can enforce invariants of evolution when the system evolves during its lifetime.

The focus of the Law-Governed Architecture approach is the evolution of a system in its operational context. An evolving system models three aspects of the system. The first is the system itself. The second is the explicit rules (called *laws*) that govern the structure of the system, the evolution of the structure, and the evolution of the laws. The third is the environment in which a system lives and the laws are enforced.

The laws can be classified into two categories. The system sub-laws govern the structure and behavior of the system. The evolution sub-laws regulate the development and evolution of the system and the laws themselves. Based on a set of initial laws, a system can evolve into other forms. During the evolution, certain rules are enforced, and these rules are called evolution invariants. Strong invariants are those invariants that not even the developer or the manager can change.

Different types of systems, different kinds of laws, and different enforcement techniques can be used in Law-Governed Architecture. The laws can be enforced statically and centrally, through a persistent object base describing all program modules, rules of evolution, meta rules about rules creation and modification, and builders who conduct development and evolution. The laws can also be enforced dynamically and distributedly, by intercepting message exchanges between architectural components.

The Law-Governed Architecture can be applied to enforce secure operation and evolution of a system. For example, a set of rules can be defined to require that one component cannot access data in another component. Rules can be refined into more detailed rules. They can also be relaxed to allow more permissive accesses. However, the strong invariants should never be violated.

In sum, Law-Governed Architecture not only models the architecture of a system but also specifies and enforces its evolution, through a set of reflexive rules. The rules can specify the security properties of the system.

The limitation of the Law-Governed Architecture methodology lies in the expressiveness and enforcement of the laws. The laws must be enforceable, and the enforcement should be reasonably efficient. This limits laws that can be imposed. The methodology suggests that there still are many useful laws within the limit. Finding these laws remains an open research problem.

2.5.8 Discussion

This section has discussed several software architecture-related solutions proposed for handling security of componentized software systems.

The simple extension of standard object-oriented notions with security information (Section 2.5.1) can be rather useful, when such a model comes into existence at a later stage of design. They can serve as a prelude to the secure program partition method [105], whose information flow security requirements on programs can derive from the secure object-orientated design models.

UML is now widely accepted as the standard detailed design notations. Previous research shows it had some major shortcomings when used to describe software architecture [93]. Thus the techniques proposed in Section 2.5.2 might not be well suited for architectural security. With the recent introduction of UML 2.0, this issue might need to be revisited.

Security should be addressed as early as possible. This naturally leads to an architecture-based approach. Simple extensions to module interconnection models (Section 2.5.3) do not provide a formalism rich enough to express and reason about architectural security concerns. Even models with a formal underpinning (Section 2.5.4) can mix the artificial requirements of the formalism and the underlying semantics of the real communication, and hinder the ability to reason about security in certain cases.

An architecture model that features connectors (Section 2.5.5 and 2.5.6) can facilitate the analysis and design of security, because the security issue can be expressed clearly at an early stage, and reasoning about, composing and implementing security can be allocated into relevant connectors.

An architecture model can also guide the proper evolution of a system (Section 2.5.7). The model can serve as a basis to prevent the system from degenerating into insecure variants. This still remains a big challenge for researchers.

Compared to previous methods using architectural connectors (Section 2.5.5 and 2.5.6), which only handles simple encryption and description functions but do not address other security requirements such as authentication and authorization, our approach supports the dominant security enforcement mechanism (namely access control), adopts an extensible architecture description language to express security modeling, uses connectors as a central vehicle for expressing and enforcing access control decisions, and provides a suite of support tools to realize these concepts.

3 Basic Modeling Concepts and an Analysis Algorithm

This chapter elaborates on our connector-centric approach to solve the architectural access control problem: ***how can we describe and check access control issues at the software architecture level?*** We first define the basic concepts in architectural access control, and then we give an overview of the proposed secure Architecture Description Language, Secure xADL. After that we discuss the central roles that connectors play in our approach. Then we establish the different architectural contexts that are involved in making access control decisions. Finally we present an algorithm that can check whether the intended architectural access should be granted within the given contexts.

3.1 Architectural Access Control

We choose the discretionary access control model discussed in Section 2.2.1 as the base security model, because it is the dominant model deployed and utilized by the majority of the componentized and networked software systems. We introduce the following core concepts that are necessary to model access control at the architecture level: ***subject***, ***principal, resource, privilege, safeguard,*** and ***policy***.

3.1.1 Subject

A ***subject*** is the user on whose behalf software executes. Subject is a key concept in security, but it is missing from traditional software architectures. Traditional software architecture generally assumes that a) all of its components and connectors execute under the same subject, b) this subject can be determined at design-time, c) it generally will not change during runtime, either inadvertently or intentionally, and d) even if there is a change, it has no impact on the software architecture. As a result, there is no modeling facility to capture allowed subjects of architectural components and connectors. Also, the allowed subjects cannot be checked against actual subjects at execution time to enforce security conformance. We extend the basic component and connector constructs with the subject for which they perform, thus enabling architectural design and analysis based on different security subjects defined by software architects.

3.1.2 Principal

A subject can take multiple ***principals***. Essentially, principals encapsulate the credentials that a subject possesses to acquire permissions. There are different types of credentials. In the classic access control model, the principal is synonymous with subject, directly designating the identity of the subject. But other types of principals provide indirection and abstraction necessary for more advanced access control models, as we will see in Chapter 4. The results for accessing resources will vary depending on the different principals a subject possesses.

Principals encapsulate credentials, yet in one sense a subject is also one type of credential, from which an access control decision based on that subject can be made. Both subject and principals are summary credentials that collect and abstract more concrete types of credentials for further use. Such more concrete credentials can be something the accessing entity is (the identity), something the accessing entity owns, or something the accessing entity knows (such as the password). The authentication process associates these more concrete credentials to the more abstract subject and principals.

3.1.3 Resource

A ***resource*** is an entity whose access should be protected. For example, a read-only file should not be modified, the password database can only be changed by administrators, and a privileged port can only be opened by the root user. Traditionally such resources are *passive*, and they are accessed by active software components operating for different subjects. In a software architecture model, resources can also be *active*. That is, the software components and connectors themselves are resources whose access should be protected. Such an active view is lacking in traditional architectural modeling. We feel that explicitly enabling this view can give architects more analysis and design power to improve security assurance.

3.1.4 Permission, Privilege and Safeguard

Permissions describe a possible operation on an object. Another important security concept that is missing from traditional ADLs is ***privilege***, which describe what permissions a component possesses depending on the executing subject. Most current modeling approaches take a maximum privilege route, where a component's interfaces list all privileges that a component possibly needs. This could become a source for privilege escalation vulnerabilities, where a less privileged component is given more privileges than what it should be properly granted. A more disciplined modeling of privileges is thus needed to reduce such vulnerabilities. We model two types of privileges, corresponding to the two types of resources. The first type handles passive resources, such as which subject has read/write access to which files. This has been extensively studied in traditional resource access control literatures. The second type deals with active resources. These privileges include architecturally important privileges, such as instantiation and destruction of architectural constituents, connection of components with connectors, execution through message routing or procedure invocation, and reading and writing architecturally critical information. Little attention has been paid to these privileges, and the limited treatment so far has neglected the creation and destruction of software components and connectors [145].

A notion corresponding to privilege is ***safeguard***, which describe permissions that are required to access the interfaces of the protected components and connectors. A safeguard attached to a component or a connector specifies what privileges other components and connectors should possess before they can access the protected component or connector.

3.1.5 Policy

A ***policy*** ties all concepts defined above together. It specifies what privileges a subject, with a given set of principals, could have to access resources that are protected by safeguards. It is the foundation for architectural constituents to make access control decisions. Components and connectors consult the policy to decide whether an architectural access should be granted or denied.

There have been numerous studies on security policies [54, 102, 144]. Since our focus is on a more practical and extensible modeling of software security at the architecture level, our priorities in modeling policy are not theoretical foundations, expressive power, or computational complexity. Instead, we focus on the applicability of such policy modeling within a software environment.

3.2 A Secure Software Architecture Description Language

To solve the architectural access control problem, we need a language to express the security requirements of software architecture. We extend our existing Architecture Description Language (ADL), xADL 2.0 [27], with these concepts introduced in the last section, to get a new language, Secure xADL. We adopt the eXtensible Access Control Markup Language (XACML) [106] as the basis for architectural security policy modeling. This is the first effort to model these security concepts directly in an architectural description language.

This section first gives overview of xADL and XACML, then describes the syntax constructs of Secure xADL, and finally discusses the rationales for following the extension route in language design.

3.2.1 Overview of xADL

xADL is an XML-based extensible ADL. It has a set of core constructs, and it supports modular extensions.

The core constructs of xADL support modeling both the design-time and run-time architecture of software systems. The most basic concepts of architectural modeling are components and connectors. Components are loci of computation, and connectors are loci of communication. xADL adopts these two concepts, and extends them into design-time types and run-time instances. Namely, in the design-time, each component or connector has a corresponding type, a componentType or a connectorType. At run-time, each component or connector is instantiated into one or more instances, componentInstances or connectorInstances. This run-time instance/design-time structure/design-time type relationship is very similar to the corresponding relationship between the run-time objects, the program objects, and the program class hierarchy.

Each component type or connector type can define its signatures. The signatures define what components and connectors provide and require. The signatures become interfaces for individual components. Note that xADL itself does not define the semantics of such signatures and interfaces. It only provides the basic syntactic support to designate the locations of such semantics.

xADL also supports sub-architecture. A component type or a connector type can have an internal sub-architecture that describes how the component type or the connector type can be refined and implemented, with a set of components and connectors that exist at a lower abstraction level. xADL allows specifying the mapping between the signatures of the outer type and the interfaces of the inner constituents. The sub-architecture support enables composing more complex components or connectors from more basic ones.

xADL has been designed to be extensible. It provides an infrastructure to introduce new modeling concepts, and has been extended successfully to model software configuration management and provide a mapping facility that links component types and connector types to their implementations.

3.2.2 Overview of XACML

The eXtensible Access Control Markup Language (XACML) [106] is an open standard from OASIS to describe access control policies for different types of applications. It is utilized in an environment where a policy enforcement point (PEP) asks a policy decision point (PDP) whether a request, expressed in XACML, should be permitted. The PDP consults its policy, also expressed in XACML, and make a decision. The decision can be `permit`, `deny`, `not applicable` (when the PDP cannot find a policy that clear gives a permit or a deny answer), and `indeterminate` (when the PDP encounters other errors).

The core XACML is based on the classic discretionary access control model, where a request for performing an action on an object by a subject is permitted or denied. In XACML an object is termed a resource. Syntactically, a PDP has a `PolicySet`, which consists of a set of `Policy`. Each Policy in turn consists of a set of `Rule`. Each `Rule` decides whether a request from a subject for performing an action on a resource should be permitted or denied. When a PDP receives a request that contains attributes of the requesting subject, action, and resource, it tries to find a *matching* `Rule`, whose attributes match those of the request, from the `Policy` and `PolicySet`, and uses the matching rule to make a decision about permitting or denying.

XACML has the following characteristics that make it a suitable choice to meet our policy modeling needs:

Firstly, the language is based on XML, which makes it a syntactically natural fit for our own XML-based ADL, xADL.

Secondly, the language is extensible. The language core supports expressing policies within the classic access control model. Several extensions, named profiles in the XACML standard, are developed to suit more specific needs. Each profile defines new concepts and algorithms that are applicable to a new domain. This modular approach, similar to our own xADL modular approach, makes the language evolvable. The extensibility allows us to adopt it for architectural modeling without loss of future expressiveness.

Thirdly, XACML provides a clean conceptual framework. The core concepts of subject, action, and object are directly adopted from the classic access control model. The basic matching procedure, based on the attributes of rules and requests, is based on first order logic and set theory, which makes XACML both reasonably expressive (many policies can be expressed using this formalism), and plausibly practical (most practical software architects and developers can be expected to master the language, maybe after some initial training). One notable feature of XACML is that it supports a variety of combing algorithms that allows flexibility in combing rules and policies. More specifically, it provides both a deny override algorithm and a permit override algorithm. The former, when combined with a “permit all” rule, supports an “open policy” [116] (where any requests that are not explicitly denied will be permitted), and the latter algorithm, when combined with a “deny all” rule, supports a “close policy” (where any requests that are not explicitly permitted will be denied).

Fourthly, the language has been equipped with a formal semantics [64]. While this semantics is an add-on artifact of the language, it illustrates the possibility to analyze the language more formally, and opens possibilities for applying relevant theoretical results about expressiveness, safety, and computational complexity to the language.

Last, but not least, even though XACML is still a rather new language, some tool support has been available. The first version came out in February 2003, and the second version is recently approved in February 2005. Some early tool support has been provided, including an open source evaluation engine implemented in Java [135] that can evaluate whether a request should be permitted by a policy, and a syntax-directed editor [44] for constructing and modifying XACML documents. These tools facilitate our research on architectural security policy modeling.

3.2.3 Constructs of Secure xADL

Combing the xADL language, the XACML language, and the architectural access control concepts defined in Section 3.1, we define a secure software architecture description language, Secure xADL, to describe security properties of software architecture.

From the viewpoint of XACML, Secure xADL can be considered as a profile for the software architecture domain. The profile defines new subjects (such as components and connectors), actions (such as instantiating connectors and connecting components), and resources (such as connectors connected to and interfaces being accessed).

From the viewpoint of xADL, Secure xADL defines a new schema that supplies a set of new elements types. Such types can be utilized, along with other base and extension xADL schemas, in defining a complete software architecture.

Figure 3-1 depicts the core syntax of Secure xADL. The central construct is SecurityPropertyType. It collects the subject, the principals, the privileges, and the policies of an architectural constituent. The policies are written in XACML syntax, and embedded in the xADL syntax. The SecurityPropertyType can be attached to types of components and connectors. Figure 3-1 illustrates that it is attached to a connector type to make a secure connector type. The

SecurityPropertyType can also be attached to components and connectors, making them secure components and connectors. Finally, the SecurityPropertyType can also be attached to the specifications of sub-architectures and the description of the global software architecture.

```
<complexType name="SecurityPropertyType">
  <sequence>
    <element name="subject"
           type="Subject"/>
    <element name="principals"
           type="Principals"/>
    <element name="privileges"
           type="Privileges"/>
    <element name="policies"
           type="Policies"/>
  </sequence>
</complexType>
<complexType name="SecureConnectorType">
  <complexContent>
    <extension base="ConnectorType">
      <sequence>
        <element mame="security"
           type="SecurityPropertyType"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="SecureSignature">
  <complexContent>
    <extension base="Signature">
      <sequence>
        <element name="safeguards"
           type="Safeguards"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
<!-- similar constructs for component, structure, and
instance -->
```

Figure 3-1, Secure xADL schema

Another construct illustrated in Figure 3-1 is Secure Signature. It has a set of associated safeguards to protect its access. The Secure Signature can be used to define signatures of component types and connector types. Safeguards can also be used on interfaces of components and connectors.

The xADL fragment in Figure 3-2 (In this dissertation, the XML syntax is greatly abbreviated, and indentation is used to signify the markup structure) specifies a secure connector. This secure connector is of type `BridgeConnector_type`, which is a secure connector type. The connector has a subject of US. Its associated policy, written in XACML, specifies that it will only allow the creation of the connector when the associated subject is US. Different types of policies will be further discussed in Section 4.4.

```
<connector id="UStoFranceConnector"
                xsi:type="SecureConnector">
  <type href="#BridgeConnector_type" />
  <security>
    <subject>US<subject/>
    <policies>
      <PolicySet PolicyCombiningAlgId="deny-overrides">
        <Policy RuleCombiningAlgId="deny-overrides">
          <Rule Effect="Deny">
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>SecureManagedSystem
              <AttributeDesignator>subject-id
            <ResourceMatch MatchId="string-equal">
              <AttributeValue>UStoFranceConnector
              <AttributeDesignator>resource-id
            <ActionMatch MatchId="string-equal">
              <AttributeValue>urn:xadl:action:AddBrick
              <AttributeDesignator>action-id
            <Condition FunctionId="not">
              <Apply FunctionId="string-is-in">
                <AttributeValue>US</AttributeValue>
                <AttributeDesignator>subject
          <Rule Effect="Permit" />
        </Policy>
      </PolicySet>
    </policies>
  </security>
</connector>
```

Figure 3-2, A Secure Connector with Subject and Policy

Note that both the policy combination algorithm and the rule combination algorithm are `deny-overrides`, and the last rule, which has no match clauses and thus matches any request, is a `permit` rule. The net effect is that if a request matches the first rule, the request will be denied; any other requests will be permitted. This is an “open policy”.

3.2.4 Rationales for Language Design

We choose not to design a brand new language. Instead, we extend and combine two base languages, xADL and XACML. Adopting this choice in language design is based on the following criteria.

Firstly, our base language, xADL, is extensible, and it has been extended to address different architectural concerns, such as types and configuration management. When it comes to modeling software security, it is rather natural that we keep this choice and focus more on the core issues in architectural access control. We have incorporated a set of core security concepts, and also allow further security extensions in the future. These extensions will eventually be subject to the extent that is made possible by both theoretical expressiveness and practical applicability.

Secondly, XACML is also extensible. Its core specification describes the basic concepts of subjects, actions, and resources and specifies how these concepts can be integrated into rules, policies, and policy sets, and how these policies can be combined. Several profiles have been developed for new domains. Software architecture can be viewed as a new domain for XACML.

Thirdly, XACML provides a flexible framework for combining policies, which allows us to design and combine suitable policies from different architectural contexts, as will be discussed in Section 3.4.

Fourthly, both languages have ample tool support that further facilitates new language designs. Our base language has been supported by a data binding library and an architectural development environment that makes a developer of extensions focus more on the development of the features per se. The availability of the evaluation engine and the editor for XACML allows us to devote more attentions to utilize XACML in an architectural context.

Finally, the major advantage of a new language is that it can be easily experimented and changed by the designer to suit special needs. However, we feel the core concepts that a language tries to express are more important than their syntactic expressions. If such expressions do not bring major advantages for conceptual understanding or tool support, we feel mere syntactic convenience does not fully compensate for the associated cost. From a language design viewpoint, reusing existing language facilities provides many benefits that a newly developed language must exhibit before such a new language can be even tested for its really innovative features. The costs of developing a brand new language are not always easy to justify. We feel this is especially true for modern software engineering research. Many times a language itself is a mere syntactic convenience, because the core concepts that the language embodies can take many different forms. The usability of the language is determined more by the concepts and associated tools.

3.3 The Central Role of Architectural Connectors

Architecture Description Languages (ADLs) provide the foundation for architectural description and reasoning [94]. Most existing ADLs support descriptions of structural issues, such as components, connectors, and configurations. Several ADLs also support descriptions of behaviors [2, 84]. The description of behaviors is either centered around components, extending the standard "providing" and "requiring" interfaces [137], or is attached to connectors, if the language supports connectors as first class citizens [2]. These descriptions enable reasoning about behaviors, such as deadlock avoidance and deadlock detection [65].

Among the numerous ADLs proposed, some do not support connectors as first class citizens [29, 84]. Interactions between components are modeled through component specifications in these modeling formalisms. This choice is in accordance with the principles of component-based software engineering, where every entity is a component and interactions between components are captured in component interfaces. A component has a "provided" interface that lists the functionality this component provides. It also has a "required" interface that enumerates the functionalities it needs in providing its functionality. Interactions between components are modeled by matching a component's "required" interface to other components' "provided" interfaces.

Embedding interaction semantics within components has its appeal for component-based software engineering, where components are the central units for assembly and deployment. However, such a lack of first class connectors does not give the important communication issue the status it deserves. This lack blurs and complicates component descriptions, which makes components less reusable in contexts that require different interaction paradigms [28]. It also hinders capturing design rationales and reusing implementations of communication mechanisms, which is made possible by standalone connectors [33]. We believe a first class connector that explicitly captures communication mechanisms provides a necessary design abstraction.

Several research efforts are focused on understanding and developing connectors in the context of ADLs. A taxonomy of connectors is proposed in [95], where connectors are classified by services (communication, coordination, conversion, facilitation) and types (procedure call, event, data access, linkage, stream, arbitrator, adaptor, and distributor). Techniques to transform an existing connector to a new connector [131] and to compose high-order connectors from existing connectors [83] are also proposed and experimented in handling encryption and decryption issues.

Our approach for architectural access control is centered on connectors. Connectors propagate privileges that are necessary for access control decisions. They regulate architectural connections between components. And they can also coordinate message routing securely. In the remaining part of this section we discuss the central role of connectors. The importance of connectors becomes more evident in Section 3.4 and Section 4.4, when we elaborate access control check and architectural execution.

3.3.1 Components: Supply Security Contract

A security *contract* specifies permissions an architectural constituent possesses to access other constituents and the permissions other constituents should possess to access the constituent. A contract is expressed through the privileges and safeguards of an architectural constituent.

For component types, the above modeling constructs are modeled as extensions to the base xADL types. The extended security modeling constructs describe the subject the component type acts for, the principals this component type can take, and the privileges the component type possesses.

The base xADL component type supplies interface signatures, which describe the basic functionality of components of this type. These signatures become the active resources that should be protected. Thus, each interface signature is augmented with safeguards that specify the necessary privileges an accessing component should possess before the interfaces can be accessed.

3.3.2 Connectors: Regulate and Enforce Contract

Connectors play a key role in our approach. They regulate and enforce the security contract specified by components.

Connectors can decide what subjects the connected components are executing for. For example, in a normal SSL connector, the server authenticates itself to the client, thus the client knows the executing subject of the server. A stronger SSL connector can also require client authentication, thus both the server component and the client component know the executing subjects of each other. DCOM over Internet (Section 6.4.4) could utilize such a strong connector.

Connectors also regulate whether components have sufficient privileges to communicate through the connectors. For example, a connector can use the privileges information of connected components to decide whether a component executing under a certain subject can deliver a request to the serving component.

This regulation is subject to the policy specification of the connector. DCOM for SP2 (Section 6.4.2) introduces such regulation on local and remote connections.

Connectors also have potentials to provide secure interaction between insecure components. Since many components in component-based software engineering can only be used “as is” and many of them do not have corresponding security descriptions, a connector is a suitable place to assure appropriate security. A connector decides what communications are secure and thus allowed, what communications are dangerous and thus rejected, and what communications are potentially insecure thus require close monitoring. XPConnect of Firefox (Section 6.3.10) can play such a role in securing possibly insecure extensions.

Using connectors to regulate and enforce a security contract and leveraging advanced connector capabilities will facilitate supporting multiple security models [139]. These advanced connector capabilities include the reflective architectural derivation of connectors from component specifications, composing connectors from existing connectors [114], and replacing one connector with another connector.

Connectors can be composite connectors. A composite connector combines several connectors together into a large connector to achieve a composite security policy. The combination can be conjunctive, when the composite connector permits a request only if each sub-connector permits. Or the combination can be disjunctive, when the composite connector permits if any sub-connector permits. Each sub-connector operates independently, yet they collaborate together to accomplish the accumulative effect.

3.4 Context for Architectural Access Control

Access control decisions are generally based on attributes of subjects, resources, and actions. Factors other than the subject-operation-object tuple can also contribute to access control decision. The most prominent example is time, which has been extensively used to express temporal access control constraints [69]. Such factors are called *context* or *environment* of access control decisions.

Likewise, from an architectural modeling viewpoint, when components and connectors are making security decisions, the decisions might be based on entities other than the decision maker and the protected resource. For example, a component might need to adhere to the policy of its type, in addition to following its own policy. We use *context* to designate those relationships involved in architectural access control. These relationships affect how the accessing constituent acquires its privileges and how the accessed constituent constructs its policy.

There are different types of relationships that can affect access control decision. Here we discuss four of them: the nearby components and connectors, the type of components and connectors, the sub-architecture containing components and connectors, and the global architecture. The reason we choose these four is because that we believe they are probably the most common types of contexts, and xADL provides native support for them.

3.4.1 Nearby Components and Connectors

A common source of context in software architecture is the neighboring components and connectors. For a component, the context is the connectors that it connects to. For a connector (which could be of many categories, such as a simple procedure call connector or a more complex event routing connector), the context derives from the components and other connectors that are connected to it. In the most common case where a connector is connected to two components, these two components form the immediate context for the connector. The neighboring relationship can be expanded to components and connectors that are not immediate neighbors.

An example of taking nearby components and connectors into consideration during access control decisions is the access control check algorithm of the Java security architecture [50]. In this architecture, components are methods of different classes, and connectors are the procedure calls among these methods. When a method tries to access a resource, the system will check and decide whether such an access should be granted. The system does not only inspect the permissions that the immediate calling method has. In addition, the system uses a mechanism called Stack Inspection to walk up the call stack frames and check all methods that reside above the method in the call stack (i.e., the caller of the method, the caller of the caller, etc.) also have the requisite permission. The requested access can be granted only if all methods involved have the necessary permissions. For example, in Figure 3-3, where method A calls method B that calls method C, if C needs to access a protected resource, then C and B and A should all have the necessary permissions. This example illustrates that when an access control decision is to be made, not only the immediate component (the method that tries to access the protected resource) is considered, but also the neighboring components that are connected through connectors (in this case, all methods that are above the calling method in the call stack) should also be accounted for.

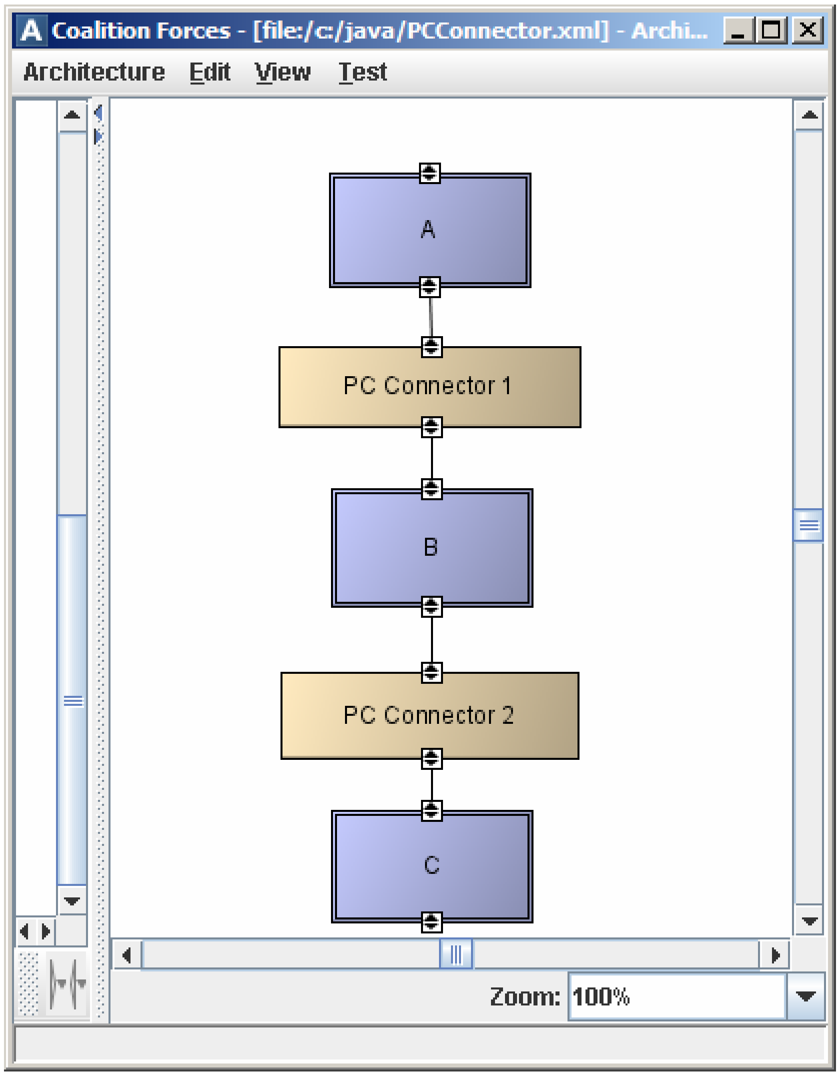


Figure 3-3, Privilege Propagation Connectors

In this stack inspection scheme, a method with enough permission can choose to take full responsibility of the access control with a `doPrivileged` call. Then, when the system walks up the call stack, if it encounters such a method, it stops walking the stack, and grants the permission of access to the original calling method. Essentially the privileged method grants any method it calls the same permissions as itself.

In Figure 3-3, if B is such a method, then its privileges can propagate through the PC Connector 2 to method C, and C can acquire the permissions of B even if the outermost caller A probably does not have the permission. This can be viewed as an instance where a more privileged component uses special connectors to grant such privileges to components of its choice.

In the general case, there exists a privilege propagation relationship among components connected by connectors. Each component carries a set of privileges. Such privileges can be propagated to the nearby components by using different types of connectors. A more specialized connector can propagate more privileges than other regular connectors. When a component makes a request, it is its own permissions and all permissions propagated to it that are considered for deciding whether the access should be granted or not.

Our approach supports describing this type of architectural context for access control. Each component can have a description of its possessed privileges. The connectors connecting them can be equipped with descriptions of how these privileges should be propagated. When an architectural constituent tries to access a protected resource, the support tools use the accumulated privileges of the constituents to check against the safeguards of the protected resource.

3.4.2 Types

An important construct of xADL is its modeling capability of component types and connector types. This construct is inspired by types in traditional programming languages, where types provide a unit for abstraction and reuse. More recently, type safety has been suggested as a foundation for security [128].

Types of components and connectors provide another context for access control decisions of their instances.

In programming languages, the general rule is that an instance possesses all properties of its type. In a security context, the relationship between an instance and its type does not always follow this pattern. When both the type and the instance specify a security policy, if there is any conflict, either the type or the instance could be given a higher priority. A policy administrator may often desire a “most-specific-override” policy, where the policy that has the most specific applicability range should take precedence. This means that the policy of the instance should be given the priority and the instance policy overrides the type policy. In other situations, the administrator might want to assure that all instances of the type obey the same policy, thus the management and enforcement of the policy can be simplified, since only one policy, the type policy, would need to be changed. This requires giving priority to the type policy.

Our approach treats the type of a component and a connector as a context, and provides the required flexibility in choosing the more authoritative policy between the instance policy and the type policy.

3.4.3 Containing Sub-architecture

xADL supports sub-architecture, which describes the internal architectural structure of a component type and a connector type. This construct allows such a type to be implemented as a collaborative set of components and connectors, and provides a facility to map the interfaces of the sub-architecture to the interfaces of the internal components and connectors. Sub-architecture enables abstraction and composition of software architectural constituents.

Substructure is a powerful modeling mechanism. A large software system can be composed from components and connectors, using sub-architecture hierarchy. Such a hierarchy has implications not only for functionality but also for security. In the most common case, each component and connector contained within the sub-architecture possesses the same security properties as the container, such as subject, principals, and privileges. In other cases, a component or a connector has different security properties than its container. A representative scenario for this case is mobile code, such as Java or JavaScript. The mobile code, downloaded from external sites, executes within the container, yet it is restricted to a sand box and subject to a set of access restrictions that do not apply to the sub-architecture container and components outside of the container. This containing sub-architecture should be utilized to make access control decisions.

These cases illustrate the concept of trust domain and trust boundary. A trust boundary delineates the trust relationship among components and connectors. All components and connectors within the boundary trust each other and form a trust domain. No security checks are needed for access between constituents within the boundary. However, crossing boundary must be very carefully checked and monitored.

Software architecture provides an appropriate means to model such boundaries. The traditional software architecture should be augmented with trust information to clearly define where trust starts and ends within a software system. Viewing from this angle, the common case of sub-architecture has only one trust domain, within which the components and connectors inherit security properties from the container. The mobile code case, on the other hand, contains multiple trust domains. The downloaded mobile code executes in a trust domain that is different than that of the containing sub-architecture.

Our approach supports treating sub-architecture as another type of context in which access control decisions are made. Our approach allows specifying the internal components and connectors to inherit the same security properties from the containing sub-architecture, thus forming the same trust domain. It also allows specifying different security properties for components and connectors within the sub-architecture and creating new trust domains.

3.4.4 Complete System

The complete software system forms the last type of context within which security decisions are made. However, the boundary of a complete software system is not always easy to identify. For example, in component-based software engineering, a complete architecture might be just embedded into another architecture and serve as a component within that larger architecture. Or, two seemingly complete systems can interact with each other to accomplish a collaborative task, thus forming a federation which might be viewed as the real complete system. Our approach defines a complete system as the highest level of architecture considered by an architect during architecture design and analysis.

In most traditional access control systems, the decisions are only made within this context. Our approach allows combining this context with three other types of contexts mentioned earlier: the nearby constituents, the type of components and connectors, and the containing sub-architecture. We feel these architectural contexts not only form the functional architecture of a software system but also have significant security relevance.

3.5 An algorithm to Check Architectural Access Control

Based on the concepts outlined in Section 3.2.3 and the contexts established in Section 3.4, in this section we present an algorithm that can check whether an architectural access in a software architecture description should be granted or denied. More formally, the algorithm finds the answer to this question: ***given a secure software architecture description written in Secure xADL, if a component A wants to access another component B, should the access be allowed?*** Finding the answer to this question can help an architect design secure software from two different perspectives. Firstly, the answer helps the architect decide whether the given architecture satisfies intended access control. If there is some access that is intended by the architect yet is not allowed by the description, the description should be changed to accommodate the access. Secondly, the answer can also help the architect decide whether there are architectural vulnerabilities that introduce undesired access. If some undesired access is allowed, then the architect must modify the architecture to eliminate such vulnerabilities.

We first present an algorithm that decides whether a single architectural access should be granted. Then, the algorithm is extended to check a complete architecture description. Finally, we discuss the applicability of the algorithm.

3.5.1 Algorithm for Single Architectural Access

In xADL, each component and connector has a set of interfaces that designate externally accessible functionalities. An interface can be either an incoming interface, designating functionality the constituent provides, or an outgoing interface, designating functionality that the constituent requires. Each incoming interface can be protected by a set of safeguards, which specify the permissions any components or connectors should possess before they can access the interface. Each outgoing interface can also possess a set of privileges, which are generally the same as those of the owner constituent.

The interfaces are connected together to form a complete architecture topology. A pair of connected interfaces has one outgoing interface and one incoming interface. Such a connection defines that the constituent with the outgoing interface accesses the constituent at the incoming interface. Each such connection defines an *architectural access*. For example, in the C2 architecture style [138], a component sends a notification at its bottom interface to a top interface of a connector. The algorithm can be used to decide whether the outgoing interface (the bottom interface of the component in the above example) carries sufficient privileges to satisfy the safeguards of the incoming interface (the top interface of the connector).

Architectural access is not limited to direct connections between interfaces. In xADL, a component cannot directly be connected to another component. Two components should be connected through a connector. Thus, a meaningful architectural access might involve the two components that are only indirectly connected through a connector. For example, if a client is connected to a server through a remote procedure call connector, then the meaning for access control is the client's access of the server's methods, though it is through a standalone connector. In the most general case, architectural access involves two interfaces that are indirectly connected by components and connectors.

The architectural access check is made more complex by the contexts discussed in Section 3.4. The accessing component can acquire privileges from multiple sources. The component can possess privileges itself. It can also get privileges from its type (Section 3.4.2). It can obtain privileges from the containing sub-architecture (Section 3.4.3) and the complete architecture (Section 3.4.4).

More complexly, privileges can also propagate to the accessing component through connected components and connectors, probably subject to the privilege propagation capability of the connectors (Section 3.4.1). When a privilege needs to propagate from one interface of one constituent to another directly connected interface of another constituent, we assume this propagation is always successful, since such connection between the interfaces, named *link* in xADL, does not possess semantics beyond pure connection.

However, when a privilege tries to propagate from an incoming interface of a constituent to an outgoing interface of the same constituent, the constituent can decide how the privilege can traverse through the constituent. In the most common case, where the constituent does not provide any special specifications, we assume that the incoming interface and the outgoing interface are connected, and we also assume that the privilege can propagate between them unmodified. This covers the majority of the specifications.

The constituent can change this default behavior. It can decide that there is no connection between the incoming interface and the outgoing interface, thus the privilege cannot be propagated. It can also decide that the privilege should not be propagated, and thus remove it from the privileges at the outgoing interface. Figure 3-4 illustrates how a procedure call connector can stop propagating the WritePasswordFile privilege from the Caller interface to the Callee interface. Finally, the connector can decide that the privilege should be replaced by another privilege. All these choices can be expressed using a policy written in XACML. The privilege propagation policies can be acquired from the constituent, the type, the container, and the complete architecture, just like other security policies.

```
<connector id="PC Connector 2"
                xsi:type="SecureConnector">
  <type href="#ProcedureCallConnector_type" />
  <security>
    <subject>System<subject/>
    <policies>
      <PolicySet PolicyCombiningAlgId="deny-overrides">
        <Policy RuleCombiningAlgId="deny-overrides">
          <Rule Effect="Deny">
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>Caller
              <AttributeDesignator>subject-id
            <ResourceMatch MatchId="string-equal">
              <AttributeValue>Callee
              <AttributeDesignator>resource-id
            <ActionMatch MatchId="string-equal">
              <AttributeValue>WritePasswordFile
              <AttributeDesignator>action-id
          <Rule Effect="Permit" />
        </Policy>
      </PolicySet>
    </policies>
  </security>
</connector>
```

Figure 3-4, Policy for Privilege Propagation

The accessed components can acquire safeguards from similar sources. One notable difference in acquiring safeguards is that this process does not involve the connected constituent context, and thus does not go through a propagation process.

```
Input: an outgoing interface, Accessing,
    and an incoming interface, Accessed

Output: grant if the Accessing can access
    the Accessed, deny if the Accessing
    cannot access the Accessed

Begin
  if (there is no path between Accessing
    and Accessed)
    return deny;
  if (Accessing and Accessed are connected
      directly)
    DirectAccessing = Accessing;
  else
    DirectAccessing = the constituent
       nearest to Accessed in the path;
  Get AccumulatedPrivileges for
    DirectAccessing from the owning
    constituent, the type, the containing
    sub-architecture, the complete architecture, and the
    connected constituents;
  Get AccumulatedSafeguards for Accessed
    from the owning constituent, the type,
    the containing sub-architecture, and the
    complete architecture;
  Get AccumulatedPolicy for Accessed from
    similar sources;
  if (AccumulatedPolicy exists)
    if (AccumulatedPolicy grants access)
      return grant;
    else
      return deny;
  else
    if (AccumulatedPrivileges contains
      AccumulatedSafeguards)
      return grant;
    else
      return deny;
End;
```

Figure 3-5, Algorithm 1: Access Control Check

To make a decision whether to allow such access, the simplest decision is to check whether the accumulated privileges of the accessing constituent covers the accumulated permissions of the accessed constituent. However, the accessed constituent can choose to use a different policy, and the sources of the policy can be from the accessed constituent, the type of the constituent, the sub-architecture containing it, and the complete architecture.

The architectural access control check algorithm is described in Figure 3-5. It first checks whether the accessing interface and the accessed interface is connected in the architecture topology. If not, the algorithm then denies the architectural access. If they are connected, the algorithm proceeds to find the interface in the path that is nearest to the accessed interface. If the accessing interface and the accessed interface are directly connected, this direct accessing interface is the same as the accessing interface. Then, the privileges of the direct accessing interface are accumulated, using various contexts, so are the safeguards and policies of the accessed interface. If a policy is explicitly specified by the architect, then the policy is consulted to decide whether the accumulated privileges are sufficient for the access. If there is no explicit policy, then the access is granted if the accumulated privileges contain the accumulated safeguards as a subset.

3.5.2 Extend to Complete Architecture

The proposed algorithm to check architectural access control is applied to a pair of interfaces. Extending it to the complete architecture description is straightforward. We can just enumerate each pair of interfaces and apply the algorithm to each pair.

A useful optimization is to calculate a topological order between the interfaces first, so the first constituent in this order has no other constituents that it can obtain privileges from. Then the algorithm uses this order to compute the privileges that each architectural constituent can obtain from its connected constituents. During the computation a constituent can get its accumulated privileges by simply applying the connector's privilege propagation capability to the accumulated privileges of other constituents connected to the connector. There is no need to do more expensive non-local computation to obtain propagated privileges.

After getting the accumulated privileges from the connected constituents context, the algorithm computes the accumulated privileges, safeguards, and policies using various contexts. Finally the algorithm can check each pair of interfaces. Depending on the completeness of the specifications, there probably does not exist non-trivial checks between certain pairs of interfaces. However, the algorithm can be used incrementally during the process of developing the full specification for the complete system.

When sub-architecture is involved, the algorithm gets more complex. In a xADL specification, there is a set of types that can be used to describe the architecture, and there is also a set of possibly independent architecture structures (the `archStructure` xADL element). A sub-architecture is implemented by connecting a component type or a connector type to one architecture structure, and map the signatures of the component type or the connector type to interfaces of the components and connectors of the architecture structure. Thus, one sub-architecture can be contained within another sub-architecture, if one type using the former sub-architecture is used to instantiate a component or a connector in the latter sub-architecture. The sub-architecture that is not contained by any other sub-architecture becomes the top level architecture. However, the top level architecture can be later used in a newly added architecture structure, and thus becomes a sub-architecture.

To apply the algorithm of checking access control between two interfaces in a complex architectural description that contains layers of architecture structures, the algorithm first checks whether the two interfaces belong to the same architecture structure. If so, then the algorithm uses that architecture structure to check access control. If the two interfaces do not belong to a common architecture structure, then the common architecture structure that contains both of them or the top level architecture can be used as the architecture structure to check access control, depending on the choice of the architect.

Once the suitable architecture structure is found, it needs to be translated into a plain graph that the Algorithm 1 in Figure 3-5 can be applied. Since the architecture structure might contain internal architecture structures, it needs to be flattened such that only primitive interfaces of the primitive architecture constituents are connected with each other. Two types of processing are necessary during this step. Firstly, if a sub-architectured type, `S`, is used to instantiate multiple instances, such as `C1` and `C2`, in a containing architecture structure, then multiple copies of the sub-architecture of `S` should be generated and properly named with prefixes of the containers. The renaming is essential to avoid name conflicts when multiple instances of interface `I` can only be differentiated as `C1.I` and `C2.I`. Secondly, the signature of a sub-architectured type, O, that maps to an interface, I, within the sub-architecture is used to propagate privileges unmodified between them, similar to the links between architecture constituents in architecture structures. This algorithm is described in Figure 3-6.

```
Input: an outgoing interface, Accessing,
    and an incoming interface, Accessed

Output: grant if the Accessing can access
    the Accessed, deny if the Accessing
    cannot access the Accessed

Begin
  if (Accessing and Accessed belong to the same
        architecture structure)
    container = the architecture structure
  else if (use top level architecture)
    container = top level architecture
  else
    container = least common container
  if (container contains other architecture structures)
    {
      replace constituents of sub-architectured types
        with the sub-architecture;
      rename the constituents of the sub-architectures
        if there are multiple instances of them;
      connect the outer signatures and the
        inner interfaces as privilege preserving
    }
  calculate the reachability closure of the expanded
      container interface graph
  return Algorithm1(Accessing, Accessed)
End;
```

Figure 3-6, Algorithm 2: Sub-architecture Access Control Check

3.5.3 Validity of the Algorithm

To validate our third hypothesis, that **with a Secure xADL description, the access control check algorithm can check the suitability of accessing interfaces**, we give an informal proof to show that the algorithm can be mapped to a well known graph reachability problem, and thus solutions for that problem can be used to decide the result of the algorithm.

Informal Proof. The validity of the algorithm can be established in a constructed privilege graph. For each privilege of an outgoing interface a node is drawn. Similarly, for each safeguard of an incoming interface a node is also drawn. Edges connecting these nodes are derived from the architectural topology and different types of contexts. For example, if a connector can propagate a privilege form one of its interfaces to another interface, then an edge is drawn between these two nodes. Viewing from a graph theory viewpoint, the algorithm executes reachability analysis, deciding whether necessary privileges can reach where they are needed. If there exists a path between the safeguard nodes and privilege nodes, then the architectural access is granted, otherwise the access is denied. Thus our algorithm can utilize any standard solution to the path finding problem to make decisions on granting or denying access. *End of Proof.*

What separates our algorithm from a normal reachability analysis is that the privileges can come from different contexts. It can come not only from the connected components and connectors, but also from types, containing sub-architectures, and the global architecture. In addition to the regular architectural topology graph, where constituents are nodes and their connections are edges, there are overlay graphs such as a type graph (a constituent node connected to its type node) and a containment graph (a constituent node connected to its container node). The policy against which these privileges are evaluated can also come from different sources. It can be either an explicitly specified policy or the accumulated safeguards of the accessed interface. Both of them can come from either the architectural topology graph or the overlay graphs.

The algorithm assumes an acyclic graph. With an arbitrary graph, standard algorithms for loop detection can be applied, and the architect needs to decide whether such loops are allowed. One possible solution to handle a loop is to partition the graph into architecturally meaningful acyclic sub graphs. Such partition could change the security implications for the original architecture, though. Determining such implications and extending the algorithm to handling arbitrary loops remains a future research question.

There might exist multiple paths from one interface to another interface. Under such cases, the algorithm depends on the architect to pick one path, and it allows the architect to enumerate all paths to inspect whether there exists one permissive path, and whether all paths are permissive. An architect has the flexibility in making the ultimate architectural choice.

The algorithm depends on degree of the completeness of the secure architecture description. In the architectural design phase, an architect can incrementally change the access control specifications of privileges and policies, and investigate their effects. If an intended access is not satisfied by the current specification, the architect can change the specification to meet the need.

Both the privileges and the policy can involve elements that can only be decided at run-time. For example, in the role-based access control model [124], a user can selectively activate different roles, thus acquiring different privileges. One future research problem is to employ our algorithm in these dynamic situations, probably with the help of the necessary enforcement infrastructure (Section 7.2.4).

4 Advanced Modeling Concepts

The previous chapter lays the foundation of architectural access control by defining the basic concepts, establishing relevant contexts, and outlining checking algorithms. This chapter builds on these foundations to cover advanced concepts necessary in modeling architectural access control in more complex situations. We first introduce the Role-based Access Control model to support large number of subjects. Then we incorporate a trust management model that enables access control across organizational boundaries. After that, we discuss the need to inspect beyond interfaces for finer degree of access control. We will then elaborate how run-time architectural access control can be modeled. This chapter concludes with a summary of the modeling concepts and language constructs of Secure xADL.

4.1 Handling Large Scale Access through Roles

4.1.1 Basic Role-based Access Control

The **Role-based Access Control Model (RBAC)** [124] is a more recent development to address two problems that are not well handled by the classic access control model. Firstly, a user needs to have different permissions under different capacities, even though the identity of the user remains the same. Secondly, in a large organization where there are tens of thousands of users, managing their access control permissions could be a daunting task. When some permission should be added or removed from users with the same capacity, the operation would have to be repeatedly performed for each user.

An extra level of indirection, role, is introduced to solve these problems. Roles become the entities that are authorized with permissions. Instead of authorizing a user's access to an object directly, the authorization is expressed as a role's permissions to an object, and the user can be assigned to the corresponding role. A user can take different roles under different situations to acquire different permissions, no longer being limited to a single set of permissions. A user can even posses several roles simultaneously to perform operations that demand those roles jointly. Also, when some permission needs to be added or removed from thousands of users, if all involved users can all take one role, then the permission can be added or removed from that role, and the system will assure that each user obtains or loses that permission. The RBAC model thus eases management of access control in large-scale organizations. Figure 4-1 depicts this indirection.

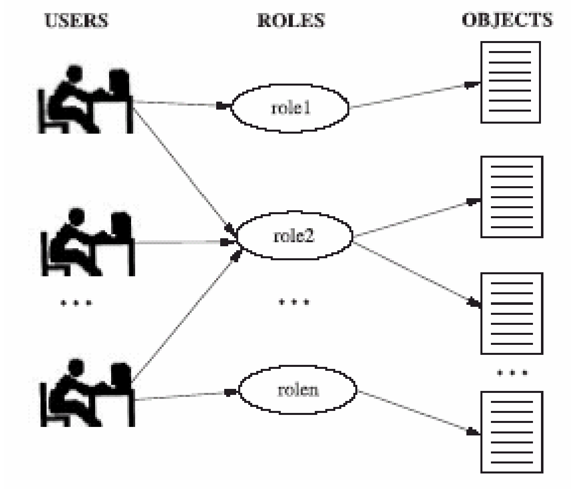


Figure 4-1, Role-based Access Control, from [122]

4.1.2 Hierarchical Roles and Separation of Duty

RBAC allows roles to form a hierarchy. In such a hierarchical RBAC model, a senior role can inherit from a junior role. Every user that takes the senior role can also take the junior role, thus obtaining all the permission associated with the junior role. A senior role can actually inherit from multiple junior roles, thus forming a lattice among the roles. In a lattice formation the inheritance relationship is more properly termed as “dominance”, where a senior role dominates a junior role. The hierarchical RBAC model resembles the single and multiple inheritance relationships in programming languages.

Another important concept in Role-based Access Control is the notion of separation of duty. The notion specifies that several sensitive tasks should not be performed by the same user. More formally, it requires that the user cannot perform those roles simultaneously. For example, a person cannot act as both the treasurer and the cashier at the same time so that embezzlement would at least require collusion of two people.

4.1.3 RBAC Support in XACML

XACML supports Core and Hierarchical RBAC through a profile. This support of RBAC through defining a profile over the core framework, without introducing unnecessary overheads, contributes to our choice of adopting XACML as the base policy language.

The RBAC profile defines roles as additional attributes of subjects and resources. To support making `permit` or `deny` decisions on requests involving roles, the profile utilizes two types of policy sets. Policy sets are the top level container in XACML, and generally only one set is needed. The two sets for RBAC are a role policy set (RPS) and a permission policy set (PPS). The role policy set restricts that it only matches the subject with the intended roles. It does not restrict on resources and actions. The permission policy set does not restrict the subject. It only limits what resources and actions will result in a permit decision. The role policy set references the permission policy set. To evaluate whether a request should be permitted or denied, the PDP should not use the permission policy set directly, since it does not limit subjects and any subjects, no matter whether they have the correct roles, will be granted permissions. Instead, the PDP should use the role policy set to exclude those subjects without the correct roles, and then use the permission policy set to decide whether the requested action on the resource should be allowed.

To support Hierarchical RBAC, the XACML RBAC profile lets the permission policy set of a senior role reference the permission policy set of a junior role. Thus, when a request reaches the senior set, it also includes permission from the junior set.

The XACML RBAC profile adopts a close policy. Each request that is not explicitly permitted in existing policy sets should be denied. Generally a PDP will return a result of `Not Applicable` if it cannot find a matching rule for the request. This constraints how Secure xADL policies should be expressed to achieve the desired result.

4.1.4 Roles as Principals in Secure xADL

RBAC has recently been standardized [5]. The standard contains four components: Core RBAC, Hierarchical RBAC, Constrained RBAC with static separation of duty, and Constrained RBAC with dynamic separation of duty. We support the RBAC model for its advanced capability in handling large scale access control. At this stage Secure xADL only supports both Core RBAC and Hierarchical RBAC.

Figure 4-2 depicts the conceptual framework of the Hierarchical RBAC model. There are four sets of entities: USERS (each element is a user), SESSIONS (each element is a session that a user is participating), ROLES (each element is a role), and PERMS (the set of permissions, whose elements are permissions about operations (members of the OPS set) on objects (members of the OBS set)). There are three important relationships: PA (the permission assignment between a role and its associated permissions), UA (the user assignment where a user is assigned associated roles and thus acquires related permissions through the PA relation), and RH (the role hierarchy where a senior role inherits from a junior role and obtains its permissions)

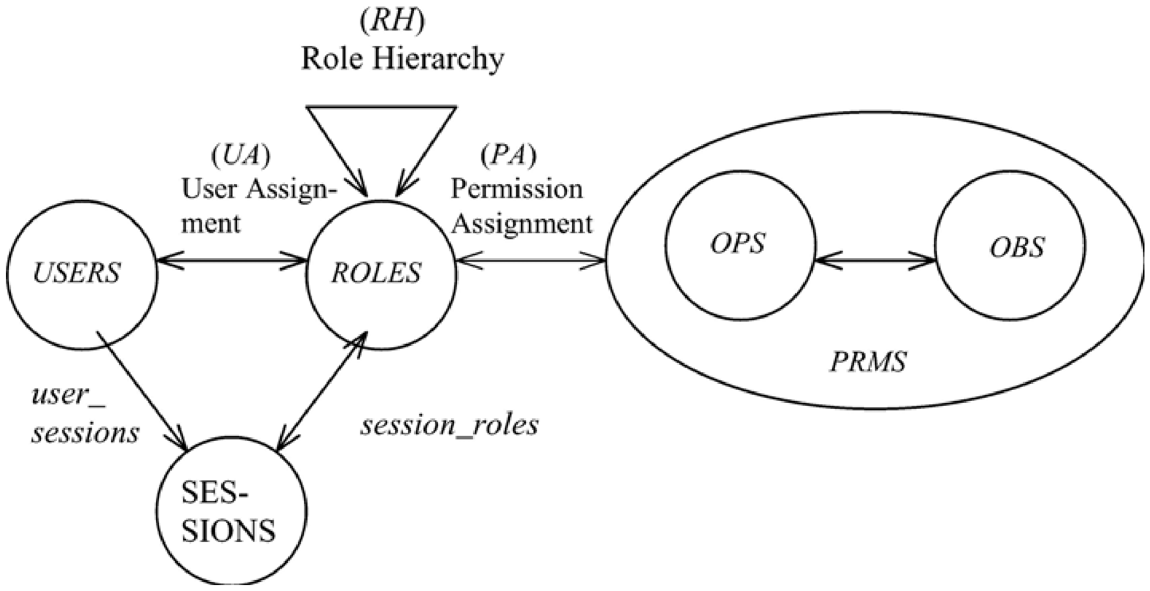


Figure 4-2, Hierarchical RBAC

Secure xADL uses principals to represent roles. As discussed in Section 3.1.2, principals are summary credentials used for access control. Since a role is the entity that is granted permissions in the RBAC model, we choose to use a principal to specify a role for a component or a connector.

In Secure xADL, each component or connector has one subject that designates the user the component or connector executes for, and it uses principals to designate the roles the user can take. Since a user might take multiple roles, there can be multiple principals associated with a component or a connector. These principals can be selectively activated or deactivated during system execution.

In the access control check algorithm (Section 3.5.1), principals are obtained and propagated like privileges, following the same contexts: the access path, the type of the component or the connector, the container, and the complete system architecture.

Figure 4-3 specifies a RBAC policy that uses only the Core RBAC model. The connector executes as the US subject and takes a NATO role (expressed as a NATO principal). Note the policy set with a `PolicySetId` "`RPS:NATO`". The specially formatted `PolicySetId` is the Secure xADL notation to signify a role policy set for a role. In this case the policy set is the role policy set for the role NATO. Similarly, the "`PPS:NATO`" policy set is the permission policy set for the NATO role, and is referenced by the role policy set through the `PolicySetIdReference`. Also note that the policy set with a `PolicySetId` "UA". This is the Secure xADL notation to specify the user assignment relation for the RBAC model. The example assignment specifies that a user US can take the role NATO, adopting the XACML RBAC Profile action with the id of `urn:oasis:names:tc:xacml:2.0:actions:enableRole`.

```
<connector id="UStoFranceConnector"
                xsi:type="SecureConnector">
  <type href="#BridgeConnector_type" />
  <security>
    <subject>US<subject/>
    <principals>
      <principal>NATO</principal>
    <policies>
      <PolicySet PolicySetId="RPS:NATO">
        <PolicySetIdReference>PPS:NATO
      </PolicySet>
      <PolicySet PolicySetId="PPS:NATO">
      </PolicySet>
      <PolicySet PolicySetId="UA">
        <Policy RuleCombiningAlgId="permit-overrides">
          <Rule Effect="Permit">
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>US
              <AttributeDesignator>subject-id
            <ResourceMatch MatchId="string-equal">
              <AttributeValue>NATO
              <AttributeDesignator>resource-id
            <ActionMatch MatchId="anyURI-equal">
              <AttributeValue>
          urn:oasis:names:tc:xacml:2.0:actions:enableRole
              <AttributeDesignator>action-id
      </PolicySet>
    </policies>
  </security>
</connector>
```

Figure 4-3, A Core RBAC Policy

4.2 Handling Heterogeneous Access through Trust Management

4.2.1 Trust and Delegation in Decentralized Systems

In a decentralized software system, where components and connectors execute for different owners, they have their own autonomous security administrative domains, and each of these domains decides who can access their services independently. Decentralization makes using a centrally managed subject and role hierarchy difficult, if not entirely impossible. The classic access control model and the role-based access control model are insufficient in these situations. Trust management schemes [130, 143] have been developed to provide a decentralized approach to address these issues.

PolicyMaker [16] is the first system that uses trust management, which combines authentication and authorization in their policy definitions, to implement decentralized access control. A local decision maker uses credentials presented to it by a remote party to make the access control decision. The credential is generally a certificate signed by the local decision maker, signifying the trust of the local party on the remote party. It unifies local and remote access control by treating a local policy also as a credential signed by the local decision control maker. Several later systems, such as KeyNote [15, 17] and SD3 [67], adopts a similar approach that uses logic and signed certificates as the basis for making access control decisions [143].

A concept related to trust is delegation. Entity A can make entity B as its delegate, so if an entity C trusts entity A then it will also allow the entity B to act on behalf of entity A [80, 152]. The delegation can propagate further and form a delegation chain, thus multiple entities are involved in a trust-based access control decision. The granted delegation can also be revoked, if entity A decides entity B should no longer act as entity A [51].

From a trust management perspective, the standard Java access control algorithm for stack walk [50] (discussed in Section 3.4.1) treats the different protection domains on a stack as a chain of trust and delegation. The system libraries, which are callees at the bottom of the stack, virtually grant their trust on the callers at the top of the stack, when the libraries invoke the `doAsPrivileged` method to perform operations requested by the callers. This trust chain is on the opposite direction of the call chain. When a less privileged A entity calls a more privileged entity B, entity B should trust entity B before it honors the call. Such trust and delegation, exhibited in the form of the `doAsPrivileged` method call, should be exercised carefully.

4.2.2 Role-based Trust Management in Secure xADL

So far we have discussed the classic access control model (Section 2.2.1), the role-based access control model (Section 4.1), and the trust management model (Section 4.2.1). Several efforts have been made to provide a more unified view of these models [123, 140]. Such a unified view provides the theoretical foundation for our architectural treatment of access control models. As we have discussed in Section 3.1.1, the Subject concept captures the user on whose behalf software executes. Section 3.1.2 suggests that principals provide indirection and abstraction necessary for more advanced access control models. In the classic model, the indirection is unnecessary, and a principal becomes synonymous to the subject. In the role-based model, the principal expresses the different roles a user can perform, and is essential to the access control decision. In the trust management model, a remote subject's principal can be public key credentials or certificates signed by the local subject, so that a local subject can use these principals to decide whether a request should be permitted or denied.

Secure xADL adopts the role-based trust management (RBTM) framework [81] as its base for support of trust management. The framework has a theoretical semantics based on logic and set theory, which makes it a natural fit since other parts of Secure xADL is also based on similar theoretical foundations. The framework uses roles as the basis for granting trust, so it integrates with the RBAC support of Secure xADL easily.

In a decentralized environment, there are different autonomous administrative domains. The most basic rule of the role-based trust management framework specifies that a role R_1 defined in a domain D_1 grant its trust on the role R_2 defined in the domain D_2, so that each user from the domain D_2 who can perform the R_2 role can acquire permissions in the domain D1 that are granted to the role R_1. The authors of the RBTM framework formally express this as $R_1.D_1 \leftarrow R_2.D_2$, signifying a trust relationship from D_1 to D_2. (The framework is backed by logic programming, so the arrow in the formal rule points in a direction that implies logic derivation.)

From a Role-based Access Control perspective, the trust grant rule is similar to the relationship between a senior role and a junior role in the Hierarchical RBAC model. There a senior role obtains all permissions of the junior role. Here a role from a remote domain acquires the permissions granted on a local role in the local domain. The RBTM framework views the trust management relationship as the set containment relationship between independently defined roles.

From a trust management perspective, the trust grant rule is a credential that a remote can present to the local entity for access control. If the remote entity can convincingly deliver the credential (for example, the credential is signed by the local entity), then the local entity will grant the permissions associated with the role.

Architectural constituents of Secure xADL use the trust grant rule to define what trust locally defined roles will grant on remotely defined roles. Syntactically this is expressed along with other Role-based Access Control policies.

Figure 4-4 is an example of a Secure xADL trust management policy. Note the “`TM:deault`” policy set, which is the Secure xADL notation to signify a trust management policy. The rule specifies that the US role from the US domain (specified by the subject attribute with an id of `urn:xadl:domain:name`) trusts (specified by the `urn:xadl:action:Trust` action) the France role from the France domain (specified by the `urn:xadl:domain:name` resource attribute). Compared to the RBAC policy in Figure 4-3, each role in this policy is explicitly specified with the autonomous domain under which it is defined. In the previous RBAC policy, the same default domain is assumed for each role defined.

4.2.3 Trust Boundary and Architectural Connector

While the term trust management was recently coined to handle decentralized access control, trust has long been a central concept in security research [121]. From a software architecture viewpoint, a trust boundary delineates the trust relationship among components and connectors. All components and connectors within a boundary trust each other, and thus no security checks would be necessary. However, any access crossing a boundary should be very carefully monitored and checked. Software architecture descriptions could provide an appropriate means to describe and analyze these trust boundaries.

```
<connector id="UStoFranceConnector"
                xsi:type="SecureConnector">
  <type href="#BridgeConnector_type" />
  <security>
    <subject>US<subject/>
    <policies>
      <PolicySet PolicySetId="TM:default">
        <Policy RuleCombiningAlgId="permit-overrides">
          <Rule Effect="Permit">
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>US
              <AttributeDesignator>subject-id
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>default
              <AttributeDesignator>urn:xadl:domain:name
            <ResourceMatch MatchId="string-equal">
              <AttributeValue>France
              <AttributeDesignator>resource-id
            <ResourceMatch MatchId="string-equal">
              <AttributeValue>France
              <AttributeDesignator>urn:xadl:domain:name
            <ActionMatch MatchId="string-equal">
              <AttributeValue>urn:xadl:action:Trust
              <AttributeDesignator>action-id
        </Policy>
      </PolicySet>
    </policies>
  </security>
</connector>
```

Figure 4-4, A Trust Management Policy

Secure xADL arguments traditional software architecture descriptions with trust information to clearly delineate where trust starts and ends within a complex software system. Traditional boundaries are laid around components, connectors, and the containing sub-architectures. Secure xADL overlays these boundaries with trust information. Thus, when a component executing for one subject needs to access another connector executing as a different subject, a cross trust domain access control check should be performed. The two subjects generally are defined within a single administrative domain, so the cross-subject trust boundaries in this case still lie within that administrative domain. When a component from one autonomous administrative domain tries to access another connector in another autonomous domain, the trust management policy should be consulted to check whether the cross-domain trust boundaries should be allowed to cross.

Connectors can play an important role in checking accesses crossing trust boundaries. When a connector connects two trust boundaries in the architectural topology, the connector can propagate and delegate trust in accordance with the trust policy. The necessary trust credentials, such as the subject and principals of the accessing component, can propagate along the connectors, subject to whether the delegation is allowed and whether the accessing role is trusted. Thus at the other end of the connector the accessed component can utilize such trust information to decide whether the access request is allowed.

4.3 Handling Content-based Access

Previously, we have established interfaces of components and connectors as a basic unit for access control protection. The interfaces are protected by safeguards. The accessing components and connectors need to have sufficient privileges for these safeguards before they can access the protected interfaces. This interface-based access control scheme is based on the dominant design in component-based software engineering and software architecture, where interfaces are the key encapsulations for provided and required services.

However, depending on how an interface is designed, the interface might not always provide all information necessary for making an access control decision. For example, a file system component usually provides interfaces for creating, reading, writing, and removing files. These interfaces remain the same for accessing both ordinary and sensitive files within the component, but the results would differ depending on not only the accessing component but also on the parameters passed to these interfaces.

In certain architectural styles, there are only limited types of interfaces. For example, in the classic Unix pipe-and-filter architecture style, each pipe and each filter has a standard input interface to receive byte stream inputs and a standard output interface to send byte stream outputs. The interface itself does not differentiate one input stream from another input stream, or differentiate one output stream from another output stream. The simplicity and uniformity of the interfaces provide great flexibility in composing larger architectures from smaller building blocks, but it does not provide enough information to control access. To provide access control for architecture styles that utilize generic interfaces, Secure xADL enables content-based access control that allows access control to be based on the content passing through these interfaces.

For example, in the message-based C2 style [138], each component and connector has a top interface and a bottom interface, and all top interfaces and all bottom interfaces within an architecture are of the same type. This generic interface type is used to send and receive all requests and notifications. We adopt the xADL message extension [58] to describe the content of a C2 message, including its type (whether it is a request or a notification), its source and destination component and connector, and the names and values of the parameters of the message. XACML provides a facility to allow inspecting the content of the resource as part of the generic rule matching process, in addition to matching attributes of a subject, an action, and a resource. Combining the message extension and the content inspection facility, Secure xADL allows architectural access control decisions based on not only the interfaces of components and connectors but also the content that passes through these interfaces. For example, an architect can limit the delivery of a notification from the bottom interface of a connector to the top interface of another connector only to notifications that has a certain value for a certain parameter.

```
<connector id="UStoFranceConnector"
                xsi:type="SecureConnector">
  <type href="#BridgeConnector_type" />
  <security>
    <subject>US<subject/>
    <policies>
      <PolicySet PolicyCombiningAlgId="deny-overrides">
        <Policy RuleCombiningAlgId="deny-overrides">
          <Rule Effect="Deny">
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>UStoFranceConnector
              <AttributeDesignator>subject-id
            <ResourceMatch MatchId="string-equal">
              <AttributeValue>RouteMessage
              <AttributeDesignator>resource-id
            <ActionMatch MatchId="string-equal">
             <AttributeValue>urn:xadl:action:RouteMessage
              <AttributeDesignator>action-id
            <Condition FunctionId="not">
              <Apply FunctionId="string-is-in">
                <AttributeValue>Secret</AttributeValue>
                <AttributeSelector
            RequestContextPath="//context:ResourceContent
          <Rule Effect="Permit" />
        </Policy>
      </PolicySet>
    </policies>
  </security>
</connector>
```

Figure 4-5, Content-based Access Control

Figure 4-5 depicts a sample policy for content-based access control. When the connector routes a message (an action with the id of `urn:xadl:action:RouteMessage`), it will deny the routing if it finds inappropriate messages when it inspects the content of the message. The content of the message to inspect is described with an XPath [141] expression beginning with "`//context:ResourceContent`". XACML uses this path as an `AttributeSelector` to retrieve the content of the resource from the request.

4.4 Handling Architectural Execution

To effectively enforce secure architectural access, we also need to provide necessary run-time support for architectural execution of an architecture described in Secure xADL. Two most basic types of operation for architectural execution are architectural instantiation and architectural connection. Most architecture styles need these two types of architectural operations. For message-based architecture styles like C2, providing message routing support is also essential to architectural execution.

4.4.1 Architectural Instantiation

The first architectural operation is instantiation, namely creating the components and connectors based on the architecture description. This has not been well studied by previous work [145]. A sample instantiation policy can specify that the components and connectors should not be created unless there are public key certificates present, since merely creating the component or the connector could be dangerous enough. This is the policy adopted by recent versions of Internet Explorer for ActiveX Controls.

Secure xADL uses `urn:xadl:action:AddBrick` to specify the architectural instantiation operation, as depicted in Figure 3-2.

4.4.2 Architectural Connection

The second architectural operation is connection, namely binding the interfaces of components and connectors together. When binding a component and a connector, the policies of both the component and the connector should be consulted. If either of them rejects such a connection, then the connection operation should not be allowed. A connection operation can involve more than just immediate connections. For example, when using a connector to connect a component with another component, the policies of the connector and the two components all should be consulted. This is an example of the nearby constituent context (Section 3.4.1).

Figure 4-6 specifies a policy for architectural connection. The operation for making an architectural connection is specified by the Secure xADL action `urn:xadl:action:AddWeld`. Note the resource match uses the XACML `regexp-string-match` method, which matches strings with regular expression. In this case the value to be matched is "`.*`", thus every connection request will be permitted.

4.4.3 Message Routing

For a message-based architecture styles like C2, when a system described in the style executes, each component and connector only communicates with each other through passing messages. Thus, providing support for message routing will complete the support of executing such software systems: first instantiating components and connectors, then connecting components with connectors, and finally messaging events among them. Secure xADL supports routing messages according to specified policies. Such support can utilize the content-based access control facility discussed in Section 4.3.

```
<connector id="UStoFranceConnector"
                xsi:type="SecureConnector">
  <type href="#BridgeConnector_type" />
  <security>
    <subject>US<subject/>
    <policies>
      <PolicySet PolicyCombiningAlgId="permit-overrides">
        <Policy RuleCombiningAlgId="permit-overrides">
          <Rule Effect="Permit">
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>SecureManagedSystem
              <AttributeDesignator>subject-id
            <ResourceMatch MatchId="regexp-string-match">
              <AttributeValue>.*
              <AttributeDesignator>resource-id
            <ActionMatch MatchId="string-equal">
              <AttributeValue>urn:xadl:action:AddWeld
              <AttributeDesignator>action-id
        </Policy>
      </PolicySet>
    </policies>
  </security>
</connector>
```

Figure 4-6, Policy for Architectural Connection

In C2, there are two types of message routing. The *external message* routing occurs when messages are sent from one interface of a component to another interface of a connector. The *internal message* routing happens when a message received from one interface of the connector is forwarded to another interface of the same connector. The standard C2 semantics for a connector is to unconditionally route such messages. However, this might leak sensitive information, if any receiver at the receiving end is untrustworthy. Thus, a more discreet policy could be adopted by the system or the connector that permits only routing safe messages to appropriate interfaces.

4.5 Summary of Modeling Concepts

This section summarizes the modeling concepts proposed in Chapter 3 and Chapter 4. These concepts can be used for checking architectural access control using the algorithms defined in Section 3.5.

Secure xADL is based on a unified access control model that incorporates the classic discretional access control model [78], the role-based access control model [124], and the role-based trust management model[81]. The latter two models can be viewed as extensions to the classic model. Such a unified model can cover access control requirements for a large class of software systems.

Secure xADL extends two base languages based on XML. One language, our extensible architecture description language, xADL [27], provides basis for describing different architectural constructs, such as components, connectors, types, sub-architectures, and the complete architecture. The other language, the XACML language [106], supplies a logic and set theory based policy language utilizing matching subjects, actions, and resources to rules and policies. Both languages support modular extension.

Secure xADL extends descriptions of architectural constituents (components, connectors, types, sub-architectures, and the global architecture) with constructs necessary to model access control: **subject**, **principal**, **permission**, **resource**, **privilege**, **safeguard**, and **policy**. **Subject** is the user on whose behalf software constituents execute. A subject can take multiple principals. Each **principal** encapsulates a credential that the subject possesses to acquire permissions. It is the synonym as subject in the classic access control model, a role in the role-based model, and a decentralized role in the trust management model. A **permission** is an allowed operation on a resource. A **resource** is an entity whose access should be protected. A resource can be passive, like files, or it can be active, like components and connectors. A **privilege** describes permissions components and connectors possess, depending on the executing subject. A **safeguard** describe permissions required to access the protected interfaces of components and connectors. A **policy** ties all these concepts together, and specifies what access is allowed and what access should be denied.

5 Tools Support

In this chapter we describe support tools we have developed for the connector-centric approach to software architectural access control. We first describe the evaluation engines we have developed for this approach. These engines, while being an integral part of our approach, can also be used separately. Then we give an overview of the base architecture design environment, ArchStudio [27]. After that we illustrate the design-time tools we have developed for the environment, including editors and analyzers. Finally, we discuss the run-time tools that we have developed to fully support secure execution of event-based software systems.

5.1 Evaluation Engine of Access Control Models

5.1.1 Implementing Role-based Access Control

Because Secure xADL supports the Core RBAC and the Hierarchical RBAC parts of the RBAC standard [5], we write a Java class library to implement these parts. The RBAC standard is specified as a set of functions, but Java prefers object-oriented design. Thus we provide a procedural interface for the set of classes. The RBAC standard also requires that each part should be able to be deployed and utilized independently, thus we choose to let the Hierarchical RBAC part inherit from the Core RBAC part so the Core part can be used standalone and the Hierarchical part does not need to duplicate unnecessary code.

For the Core part, we design a set of interfaces and classes: `User`, `Role`, `Operation`, `Object`, `Permission`, and `Session`, just as specified by the standard. An `RBACCoreImpl` implements all functions specified by the standard, and can be accessed by the following procedure-oriented interface:

```
public interface RBACCore {
    User addUser(Name User);
    Role addRole(Name Role);
    void assignUser(User aUser, Role aRole);
    void deassignUser(User aUser, Role aRole);
    void grantPermission(Operation p, Object b, Role r);
    void revokePermission(Operation p, Object b, Role r);
    void checkAccess(Session s, Operation p, Object b);
    Set  assignedUsers(Role aRole);
    Set  assignedRoles(User aUser);
    Set  roelPermissions(Role aRole);
    Set  userPermissions(User aUser);
    ...
}
```

Figure 5-1, Core RBAC Interface

The RBAC Hierarchical part is based on the RBAC Core part, and adds the classes and interfaces in Figure 5-2 to provide access for the role-inheritance relationships. In Hierarchical RBAC, adding an inheritance relationship between two roles can trigger a wide propagation of permissions, because the newly added relationship could bridge two large existing role hierarchies.

```
public interface RoleHierarchical extends Role {
    void addAscendant(RoleHierarchical ascendant);
    void addDescendant(RoleHierarchical descendant);
    void addJunior(RoleHierarchical junior);
    void addSenior(RoleHierarchical senior);
    ...
}
public interface RBACHierarchical extends RBACCore {
    void addInheritance(RoleHierarchical a, d);
    void deleteInheritance(RoleHierarchical a, d);
    Set  authorizedRoles(User aUser);
    Set  authorizedUsers(Role aRole);
    ...
}
```

Figure 5-2, Hierarchical RBAC Interface

5.1.2 Integrating Role-based Trust Management

The Role-based Trust Management (RBTM) framework was implemented by its author, Ninghui Li [81]. We adapt it to suit our own needs. The original framework answers two types of queries: which roles (foreign or local) an entity is allowed to have, and to which entities (foreign or local) a role is granted. We limit the inter-autonomous domain trust relationships to roles of domains, and use the previously developed RBAC engine to query the role-user relationships within an autonomous domain. A `RoleDecentralized` role maintains to which domain it belongs. Each domain's RBAC engine uses a trust manager to manage its trust relationships with other domains. It informs the trust manager what roles from other domains it trusts and revokes the established trust relationships when appropriate. The trust manager for a domain is queried for these relationships and decides whether an access from a foreign role should be allowed.

```
public interface RoleDecentralized
                    extends RoleHierarchical {
    void setDomain(Domain owning);
    ...
}
public interface RBACDecentralized
                    extends RBACHierarchical {
    void setDomain(Domain domain);
    void setTrustManager(RBTM trustManager);
    ...
}
public interface RBTM {
    void grantTrust(RoleDecentralized l,RoleExpression r)
    void revokeTrust(RoleDecentralized, RoleExpression r)
    Set  getTrustedForeignRoles(RoleDecentralized local);
    Set  getTrustingForeignRoles(RoleDecentralized l);
    boolean checkAccess(Name localUser, Name localDomain,
         Name foreignDomain,Name operation, Name object);
    ...
}
```

Figure 5-3, RBTM Interface

5.1.3 Integrating with SunXACML

We use the SunXACML [135] open source library as the underlying policy decision engine. SunXACML provides a `SimplePDP` that reads a policy and a request that are both described in XACML, tries to find a rule from the policy that matches the request, and returns a `permit` or `deny` answer based on the found matching rule and the rule combination algorithm.

Based on the `SimplePDP`, we develop our own policy evaluation engine. Our PDP is constructed with a set of current polices and a set of potential policies. A policy finder is developed to retrieve potential polices when referred to by the current polices.

The XACML RBAC Profile specifies how policy sets can be used to implement RBAC (Section 4.1.3), but it does not specify where the policy sets can be found. Using our PDP, we supply the role policy set as the current policy, and the permission policy set as the potential policy. This not only solves the problem of locating policy, but also avoids directly evaluating using the permission set, which is prohibited by the XACML RBAC Profile.

Because we already have a RBAC engine and a RBTM engine, we decide to reuse them to both save efforts and avoid inconsistencies. We design three classes: `RBACHierarchicalWithXACML`, `RBACDecentralizedWithXACML`, and `RBTMWithXACML.` These classes read role-based and trust management policies specified in Secure xADL (Section 4.1.4 and Section 4.2.2), and evaluate against an XACML request. Internally they populate the existing engines with information from the XACML policies. A role attribute finder is developed to find the correct role policy set using specified principals or the original RBAC engine.

5.2 Overview of ArchStudio

ArchStudio [27] is an architecture development environment. It supports designing software architectures specified in xADL, and executing them if they architectures are in the C2 architecture style.

ArchStudio has two editors that an architect can use to construct and modify architectures. The first one, ArchEdit, is a syntax-directed, text-based editor. It can automatically inspect xADL extensions and provide means to add constructs introduced the extensions. The second editor, Archipelago, is a powerful graphical editor. It gives an intuitive view of the architecture.

Tron is the analysis framework of ArchStudio. Tron supports many types of architectural analysis, such as checking each component has a unique identifier and each interface is of the correct type.

ArchStudio is a C2 style application and written with the c2.fw framework. The framework is a class library that eases developing C2 style software. It provides support for both components and connectors (termed `Brick` in the framework). A broadcasting connector is the standard connector for c2.fw, as required by the C2 style.

When given a xADL description of a C2 style architecture, ArchStudio instantiates a `ManagedSystem` to execute the architecture. The `ManagedSystem` uses an `ArchitectureController` to control the execution. The controller consists of three parts: an `ArchitectureManager` that creates bricks and links them together, an `ArchitectureEngine` that manages the running threads of the bricks, and a `MessageHandler` that delivers messages from one interface of a brick to another interface of another brick.

5.3 Design-time Support

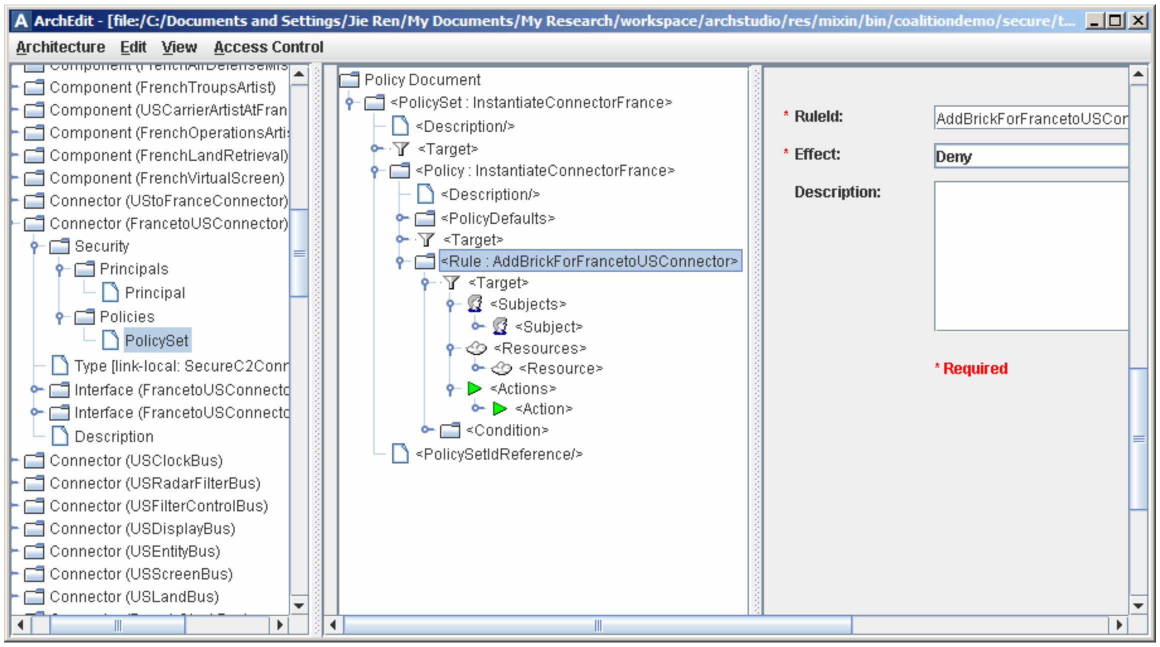
At design-time a security architect needs to specify the security properties of an architecture and check whether the specification meets intended accesses. The editor and checker of ArchStudio enable these activities.

5.3.1 Integrating the XACML Policy Editor

No extra effort is needed to support editing the subject, principals, privileges, and safeguards of Secure xADL in ArchEdit, because the constructs follow the established pattern used by pervious extensions, and ArchEdit's syntax-directed capability automatically supports editing these elements.

Editing the XACML policy requires more integration. We need to perform three changes. Firstly, each XACML PolicySet is exposed to other parts of ArchStudio as a string, even though internally it is a complex XML document. Secondly, we adopt the UMU-XACML-Editor developed by University of Murcia (UMU), and make it to read and write a policy in the string form. The editor provides a syntax-directed manner to construct a correct XACML policy. It adopts the same tree-based user interface design as employed by ArchStudio components, making integrating it seamless from a user perspective. Finally, we supply the editor with Secure xADL's subjects, resources, and actions to ease constructing a Secure xADL related policy. The integrated editor can be used to edit policies for components, connectors, their types, and architecture structures.

The policy editor integration is reused in the graphical editor, Archipelago. To support editing other constructs of Secure xADL, a context menu plug-in is written, and is invoked when the editing focus is on secure components and connectors.



ArchEdit - [file:/C:/Documents and Settings/Jie Ren/My Documents/My Research/workspace/archstudio/res/mixin/bin/coalitiondemo/secure/t...

```
Component (FrenchTroupsArtist)
Component (USCarrierArtistAtFran
Component (FrenchOperationsArti
Component (FrenchLandRetrieval)
Component (FrenchVirtualScreen)
Connector (UStoFranceConnector)
Connector (FrancetoUSConnector)
  Security
    Principals
      Principal
    Policies
      PolicySet
  Type [link-local: SecureC2Conn
  Interface (FrancetoUSConnecto
  Interface (FrancetoUSConnecto
  Description
Connector (USClockBus)
Connector (USRadarFilterBus)
Connector (USFilterControlBus)
Connector (USDisplayBus)
Connector (USEntityBus)
Connector (USScreenBus)
Connector (USLandBus)
```

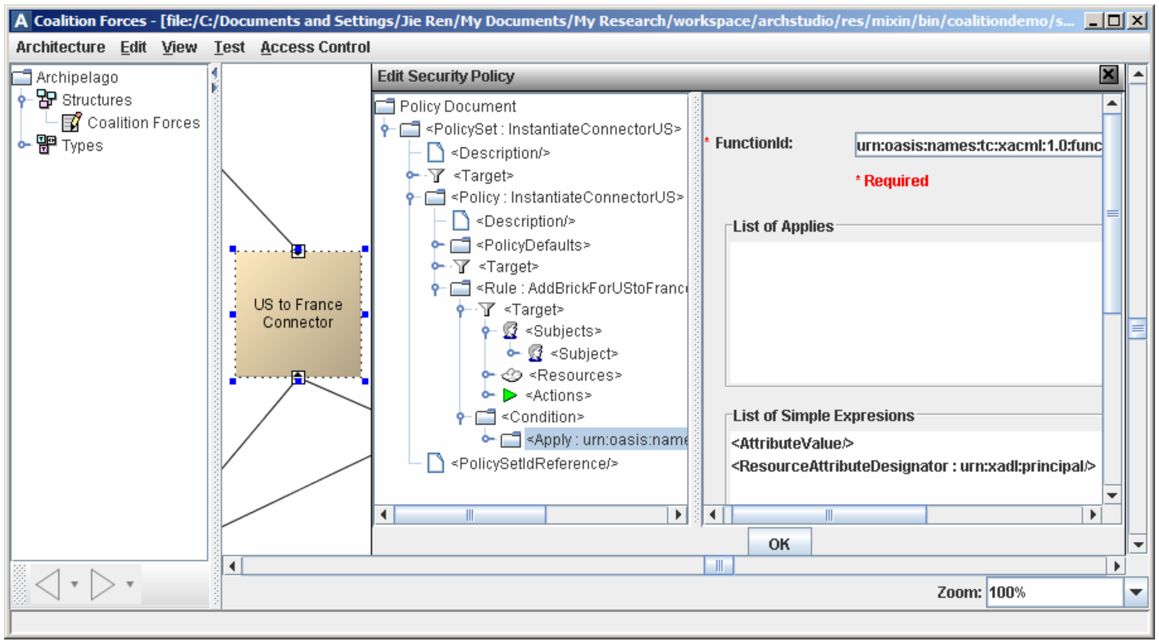
```
Policy Document
  <PolicySet : InstantiateConnectorFrance>
    <Description/>
    <Target>
    <Policy : InstantiateConnectorFrance>
      <Description/>
      <PolicyDefaults>
      <Target>
      <Rule : AddBrickForFrancetoUSConnector>
        <Target>
          <Subjects>
            <Subject>
          <Resources>
            <Resource>
          <Actions>
            <Action>
        <Condition>
    <PolicySetIdReference/>
```

* RuleId:	AddBrickForFrancetoUSCon
* Effect:	Deny
Description:	

* Required

Figure 5-4, Policy Editor in ArchEdit

Figure 5-4 depicts the policy editor in ArchEdit. Figure 5-5 shows the editor in Archipelago. Notice that the external editor is visually well integrated with other user interface components.



Coalition Forces - [file:/C:/Documents and Settings/Jie Ren/My Documents/My Research/workspace/archstudio/res/mixin/bin/coalitiondemo/s...

```
Archipelago
  Structures
    Coalition Forces
  Types
```

Edit Security Policy

```
Policy Document
  <PolicySet : InstantiateConnectorUS>
    <Description/>
    <Target>
    <Policy : InstantiateConnectorUS>
      <Description/>
      <PolicyDefaults>
      <Target>
      <Rule : AddBrickForUStoFranc
        <Target>
          <Subjects>
            <Subject>
          <Resources>
          <Actions>
        <Condition>
          <Apply : urn:oasis:name
    <PolicySetIdReference/>
```

* FunctionId:	urn:oasis:names:tc:xacml:1.0:func

* Required

List of Applies

List of Simple Expresions

<AttributeValue/>
<ResourceAttributeDesignator : urn:xadl:principal/>

Figure 5-5, Policy Editor in Archipelago

5.3.2 Access Control Analysis

Figure 5-6 illustrates how in the Archipelago graphical editor the security architect can access the architectural access control algorithm specified in Section 3.5. The architect specifies both an interface of an accessing component or connector and an interface of an accessed component or a connector. The algorithm checks whether the accessing component or connector has sufficient privilege to access the accessed interface and reports the result. If the result is not as the intended, the architect can check the involved components and connectors along the access path and modify their security properties (using editors of Section 5.3.1) to achieve the desired consequence. The check can be performed on any pairs of interfaces within the architecture. This access control check can also be invoked through the other editor, ArchEdit.

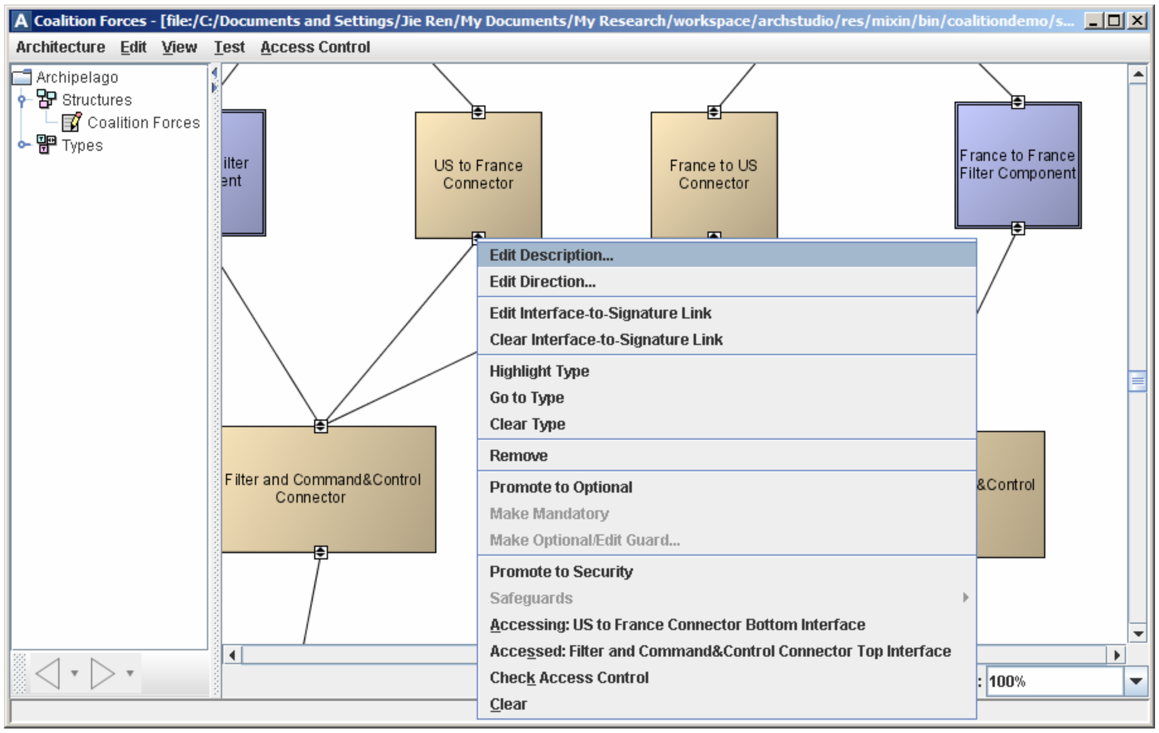

Coalition Forces - [file:/C:/Documents and Settings/Jie Ren/My Documents/My Research/workspace/archstudio/res/mixin/bin/coalitiondemo/s...
Architecture Edit View Test Access Control
Archipelago
Structures
Coalition Forces
Types
ilter
ent
US to France Connector
France to US Connector
France to France Filter Component
Filter and Command&Control Connector
&Control
Edit Description...
Edit Direction...
Edit Interface-to-Signature Link
Clear Interface-to-Signature Link
Highlight Type
Go to Type
Clear Type
Remove
Promote to Optional
Make Mandatory
Make Optional/Edit Guard...
Promote to Security
Safeguards
Accessing: US to France Connector Bottom Interface
Accessed: Filter and Command&Control Connector Top Interface
Check Access Control
Clear
100%

Figure 5-6, Menu for Access Control Check

When given a Secure xADL description, the algorithm implementation first checks whether the accessing interface and the accessed interface belong to the same architecture structure. If they are, the implementation constructs an interface graph for that structure, where each node represents an interface, and each link in the Secure xADL description becomes an edge. An edge is also drawn between an incoming interface and an outgoing interface of a connector.

If the accessing interface and the accessed interface do not belong in the same structure, then the implementation uses the top level architecture as the common container, flattens that architecture by replacing each contained structure with an interface graph for the contained structure, possibly renaming duplicated architectures and linking mapped interfaces with edges.

With such a connected interface graph, the algorithm implementation finds a path between the accessing interface and the accessed interface. The algorithm uses the standard Floyd's algorithm [26] to find a shortest path, if such a path exists. This step takes $O(n^3)$ time, where n is the number of interfaces contained in the constructed interface graph.

With this path, the implementation retrieves the privileges of the accessing interface and the safeguards of the accessed interface, from the interface, the containing connector, the type of the containing connector, and the containing architecture structure. Then the implementation propagates the privileges along the path. During this propagation process, each connector can decide whether a privilege can be propagated through it, based on its policy. This step uses the evaluation engine (Section 5.1.3). Finally, the implementation checks whether the privileges eventually reaching the accessed interface satisfy the safeguards.

The privilege propagation step takes O(m) time, where m is the length of the path. At each step, by default the privilege will propagate. But if the connector supplies a policy, then the XACML-based evaluation engine is executed. This execution needs memory resource and execution time because of the XML usage. Overall, the static analysis provides satisfactory performance for interactive usage by an architect at design-time.

5.4 Run-time Support

5.4.1 Policy Decision Point and Policy Enforcement Point

To effectively enforce secure architectural access, we need to provide necessary run-time support. The language from which we base our policy description, XACML, uses an enforcement and decision framework. In this framework, when a policy enforcement point (PEP) needs an access control decision about whether the access should be granted, it constructs a request and sends the request to the policy decision point (PDP). The PDP retrieves the applicable policies and uses them to calculate a response of permission or denial. Then the response is sent back to the PEP, and the PEP can either permit or deny the original access request.

Within this framework, the important design questions are: 1) What operations should be controlled? 2) Where is the PEP located? 3) Where is the PDP located? 4) Where should the PDP retrieve the relevant policies? Section 4.4 has answered the first questions: the controlled architectural operations should include architectural instantiation, architectural connection, and message routing for message-based architecture styles. Section 3.2 and Section 3.4 have answered the last question: architectural access control policies should come from all relevant sources: the components, the connectors, their types, their containing architectures, and the complete architecture. Thus, the unanswered questions are about the locations for the PDP and the PEP.

Because ArchStudio is written in the c2.fw framework, and a c2.fw-based application is executing under the control of the Managed System and the Architecture Controller, the natural choice is to combine the PDP and PEP together, and use both the individual bricks of the c2.fw framework and the controller as the combined enforcement/decision point. An individual brick can handle more local access control decisions, and the architecture controller can handle more global decisions. We stress that each of them can retrieve policies from all relevant sources, and an architectural operation could involve both of them. The next two subsections describe the framework and the controller, respectively.

5.4.2 The c2.fw.secure Framework

In the c2.fw framework, the basic class is `Brick`, which can be either a component or a connector. The c2.fw.secure framework is an extension of the c2.fw framework. The `Brick` class is extended to a `SecureBrick` class in the c2.fw.secure framework. The `SecureBrick` class stores the subject for which it executes, the principals of the subject, and the associated privileges. More importantly, `SecureBrick` has the capability to maintain a PDP. This PDP is consulted during various points of architectural execution, as we will see in the following sections on architectural operations.

The original c2.fw framework defines an `Interface` class to describe the top and bottom interfaces. This class is extended into a `SecureInterface` class in the c2.fw.secure framework. A `SecureInterface` carries safeguards to protect the interfaces of bricks.

5.4.3 The Secure Architecture Controller

There are three parts of an architecture controller: the engine, the architecture manager, and the message handler. The latter two are related to secure execution of C2 systems, and are extended to the Secure Architecture Manager and the Secure Message Handler, respectively. Each of them can maintain a PDP, and the PDP is populated with a policy obtained from the security property of the global architecture. This policy controls how various architectural operations are performed, as we shall see.

The secure architecture manager and the secure message handler will raise security exceptions if some operations are rejected because of security reasons. To minimize the change on other parts of ArchStudio, the exception is a subtype of the unchecked run-time Java exception, so other parts do not have to handle the exception. The exception is caught and handled by the `SecureManagedSystem`.

While both the `SecureBrick` and the secure manager/handler can obtain and enforce security policies, the advantage of placing policy enforcement at the secure architecture controller is that existing applications do not need to be rewritten to benefit from the secure execution. Many policies can be specified and enforced at the architecture structure level. Of course, a `SecureBrick` provides more capabilities and offers finer controls for developing more advanced secure applications.

5.4.4 Sources and Defaults of Policies

In executing a secure C2 application, the security polices can come from different sources of a Secure xADL description: the brick, the type, and the architecture. The security architect should decide where the proper scope is to enforce a security policy.

Another important issue is to choose a default policy between an "open policy" [116] (where any requests that are not explicitly denied will be permitted) and a "close policy" (where any requests that are not explicitly permitted will be denied). For a single desired effect, both can be utilized, but the syntax is different. It is their implications for unspecified operations that would surprise an unscrupulous architect. The architect should carefully explore and inspect the effects of the specified policy.

5.4.5 Architectural Instantiation

Having discussed the placement of policy decision and policy enforcement, in the remaining part of this section we will discuss how the Secure Architecture Controller and the c2.fw.secure framework implement the different types of architectural operations.

The first architectural operation is instantiation, creating the components and connectors based on the architecture description. This is managed by the Secure Architecture Manager. Because at this stage the C2 brick is to be created, the Secure Architecture Manager is in full control of whether to create the brick or not, using a policy similar to the one specified in Section 4.4.1.

After the manager decides the component or the connector can be created, it creates the component or the connector using the implementation specified in its type, collects relevant policies from the brick, the type, and the architecture, and supplies these policies to the newly created brick so the brick can finish its own initialization, including creating the brick PDP.

5.4.6 Architectural Connection

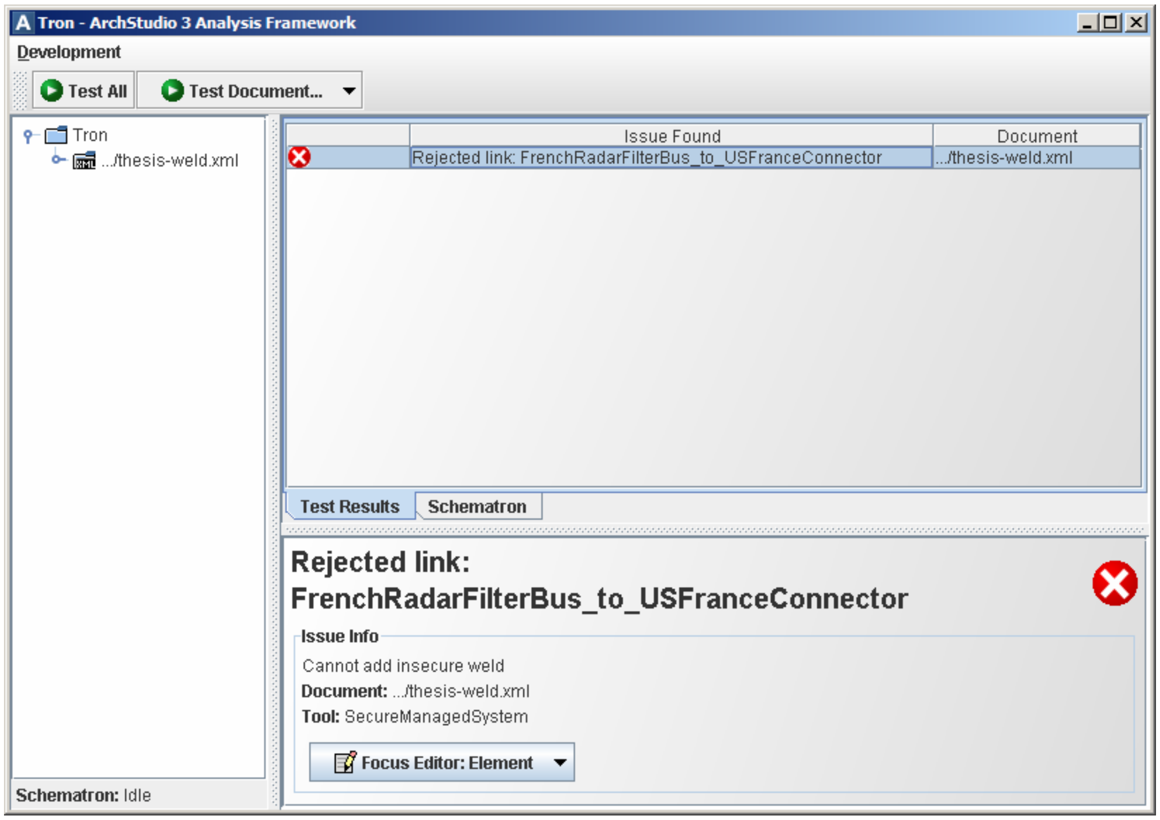
The second architectural operation is connection, binding the interfaces of components and connectors together. This is also handled by the Secure Architecture Manager.

When binding a component and a connector, since both bricks should have been created, the Secure Architecture Manager consults each of them to check whether any brick’s policy will reject the connection. Such policy is specified as in Section 4.4.2. By doing this the architecture manager gives the involved bricks the capability to control their own connections. If either of them rejects such a connection, then the connection operation should not be allowed.

The Secure Architecture Manager can also inspect the globally policy associated with the global architecture for operations that involve more than just the immediate connections. For example, when using a connector to connect a component with another component, the architectural policy is the most natural place to specify whether such connections should be allowed. This is an example of the nearby constituent context.

If the connection operation is rejected because of security reasons, the Secure Architecture Manager reports such problems. The Tron analysis tool of ArchStudio displays the unsuccessful connection attempts, as shown in Figure

5-7. Similar failures in architectural instantiation are also reported by the Tron tool. Some failed connection attempts are actually caused by failures in early failures of instantiating bricks. The architect can inspect the involved bricks and the global architecture to decide the reasons for the rejection and take appropriate actions.



	Issue Found	Document
✖	Rejected link: FrenchRadarFilterBus_to_USFranceConnector	.../thesis-weld.xml

Rejected link: FrenchRadarFilterBus_to_USFranceConnector

Issue Info
Cannot add insecure weld
Document: .../thesis-weld.xml
Tool: SecureManagedSystem

Figure 5-7, Architectural Connection Failure

5.4.7 External Message Routing

In C2, external message routing occurs when messages are sent from one interface of a component to another interface of a connector. This task is processed by the Secure Message Handler. During external message routing, the message handler can use architectural information on the message, such as the source interface and the destination interface to decide whether the message should be delivered. The message handler can also inspect more deeply into the content of the message to decide whether the message should be delivered, as discussed in Section 4.3 and Section 4.4.3.

Since evaluating a request against an XACML policy is a potentially computationally expensive task and the message handler is delivering all external messages in the system, requiring the handler to inspect each message before delivery could be very costly on performance. The current secure message handler only inspects a message in two occasions: when the message is delivered between a normal C2 brick and a secure C2 brick, and when the message is delivered between two secure C2 bricks that belong to different subjects. While this decision is a made as a tradeoff between security and performance, there is also a security rationale: a trust boundary is crossed in these two situations, so the access must be carefully inspected and regulated, as discussed in Section 4.2.3.

5.4.8 Internal Message Routing

In C2, internal message routing refers to when a message received from one interface of the connector is forwarded to another interface of that connector. The standard C2 semantics is to unconditionally route such messages. However, this might leak sensitive information, if any receiver at the receiving end is untrustworthy. Thus, a more discreet policy can be adopted by the connector, only routing safe messages to appropriate interfaces. Unlike external message routing, where the message handler is the PDP/PEP, in this case the local connector decides what it should do, since the message routed is completely under its control.

As in the case of external message routing, the connector can inspect both the architectural information and the message content to decide whether the message should be forwarded. Since the message is under full control of the connector, it is tempting to treat the message as simply traveling from one internal interface of the routing connector to another interface of the same connector, without consideration of larger contexts. However, depending on the application requirements and the capability of the messaging system, sometimes it is desirable to retain the original contextual information of a message that the message has before the message reaches the incoming interface. Since a C2 connector has only one top interface and all notifications come from this interface, whatever the sender of the message is, loosing the contextual information makes it difficult to differentiate between the sources of these messages. This would be undesirable, since the source of a message could play important roles even in internal message routing.

5.4.9 A Connector's Role in Secure Architectural Execution

In our connector-centric approach to software architectural access control, a secure connector plays two important roles in securely executing C2 style software: it participates in deciding whether architectural connections should be made, by rejecting inappropriate connections when the architecture manager consults it before the connection; and it assists in determining whether a message should be routed to the intended recipient, by discarding improper messages routed through it.

The secure connector makes these decisions based on the specified security policies and the message. It can inspect both the architectural properties of the message and the content of the message to make a decision on delivery.

6 Case Studies

In this chapter we present four case studies to assist in validating our connector-centric approach to software architectural access control. The first case study, Coalition, shows that our approach can describe how two parties that do not fully trust each other can share data without revealing more sensitive information, and how our tools support executing such a system with a secure routing connector to exchange shared data. The second case study uses a composite secure connector to connect various off-the-shelf components to construct a secure file sharing application for a local area network. The third case study models the component security architecture of the Firefox web browser, and demonstrates that our connector-centric approach can describe how Firefox security manager maintains trust boundaries and improves security through connector enhancement. The last case study illustrates Microsoft Distributed Component Object Model and shows how our approach can model its handling of architectural access control operations and its growth through various types of connectors to handle evolving security requirements.

These cases studies have helped validating three research hypotheses:

Hypothesis 1: An architectural connector may serve as a suitable construct to model architectural access control.

Hypothesis 2: The connector-centric approach can be applied to different types of componentized and networked software systems.

Hypothesis 4: In an architecture style based on event routing connectors, our approach can route events in accordance with the secure delivery requirements.

6.1 Coalition

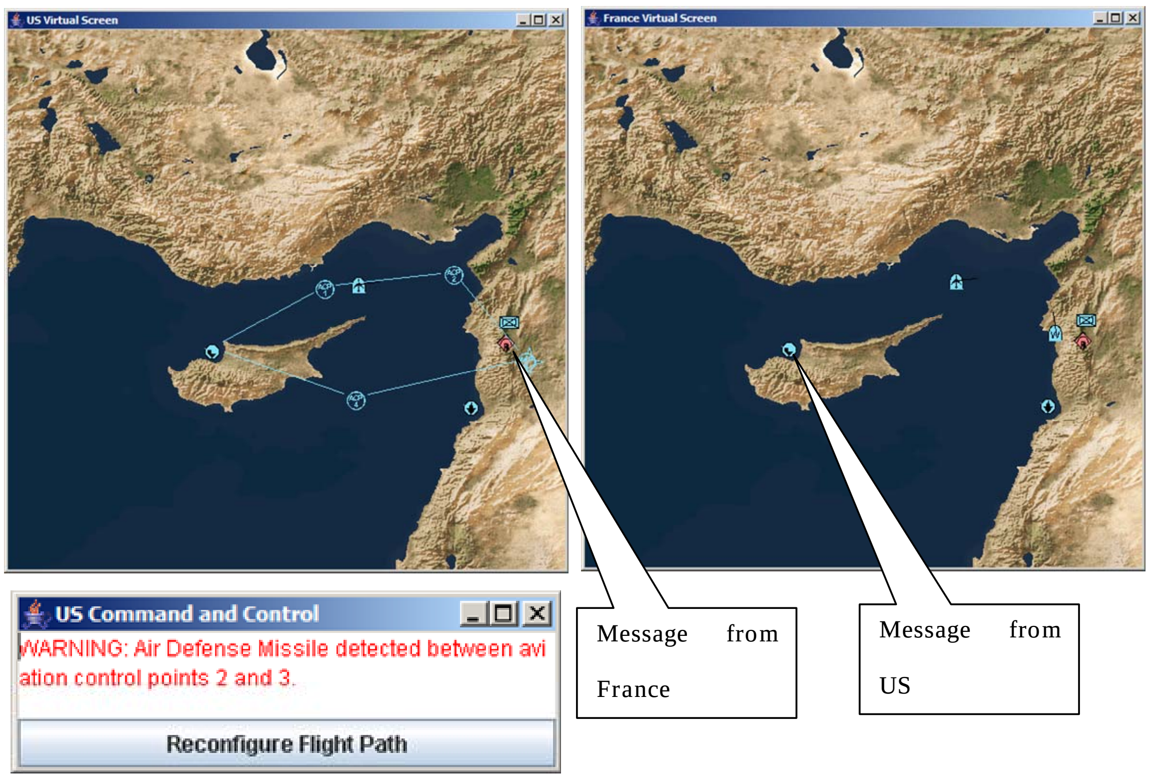
In this section, we illustrate the use of the connector-centric approach with a coalition application. We present three architectures, each has its own software and security characteristics. We also describe how to specify related architectural execution policies.

The software architecture is in the C2 architecture style. The coalition application allows two parties to share data with each other. However, these two parties do not necessarily fully trust each other, thus the data shared should be subjective to the control of each party.

The two parties participating in this application, depicted in Figure 6-1, are US and France. Each of them can operate independently, displaying the messages they receive from their own information collection devices. They can also share messages necessary to achieve a coalition mission. In Figure 6-1, US sends messages to France so France can display the additional messages. France also sends messages to US. One message about hostile air defense missile is especially important, and the US side would reconfigure the flight path of its planes after receiving the message.

6.1.1 The Original Architecture

Figure 6-2 illustrates the original coalition architecture, using our Archipelago architecture editor [27]. In this architecture, US and France each has its own process. US is on the left side, and France is on the right. The squares are components, and the regular rectangles are connectors.



WARNING: Air Defense Missile detected between aviation control points 2 and 3.

Figure 6-1, Coalition in Execution

The US Radar Filter Connector sends all notifications downward. The US to US Filter Component forwards all such notifications to the US Filter and Command & Control Connector. However, US does not want France to receive all the notifications. Thus it employs a US to French Filter Component to filter out sensitive messages, and sends those safe messages through the US Distributed Fred Connector, which connects to the French Local Fred Connector to deliver those safe messages. (A Fred connector broadcast messages to all Fred connectors in the same connectors group.) The France side essentially has the same architecture, using a French to US Filter Component to filter out sensitive messages and send out safe messages.

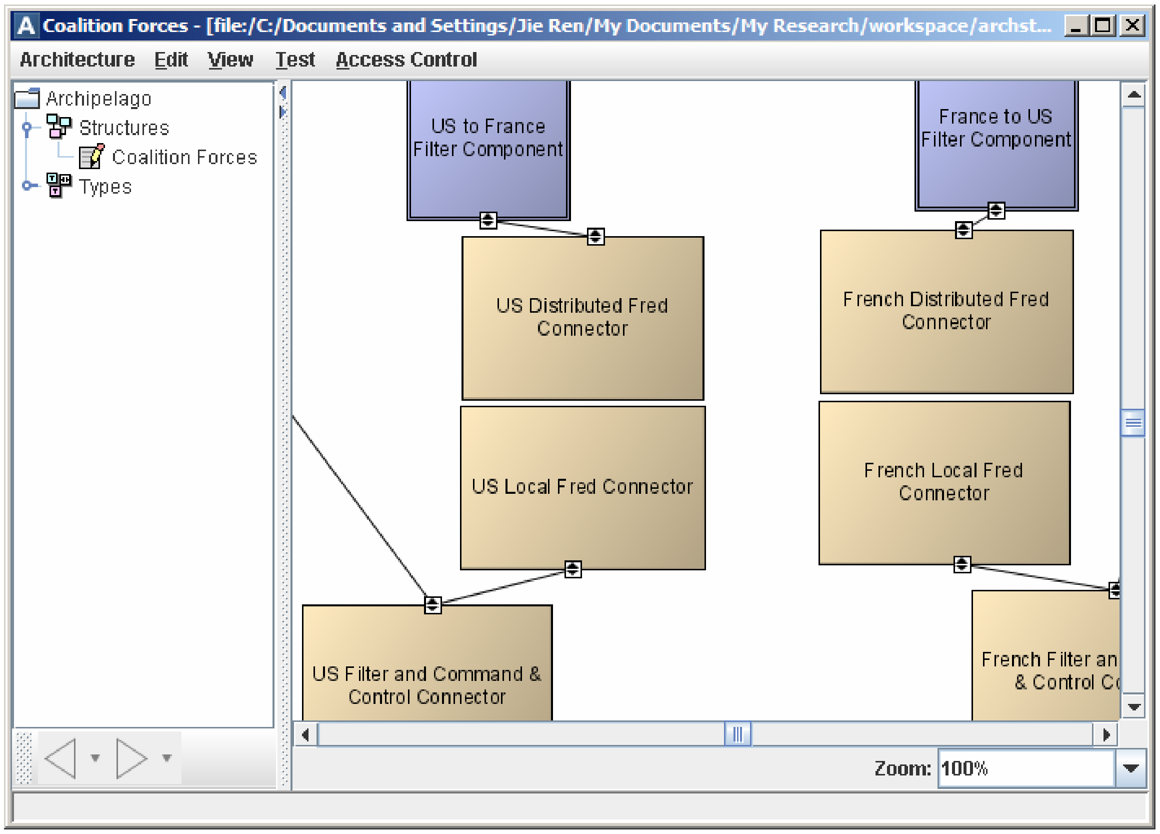
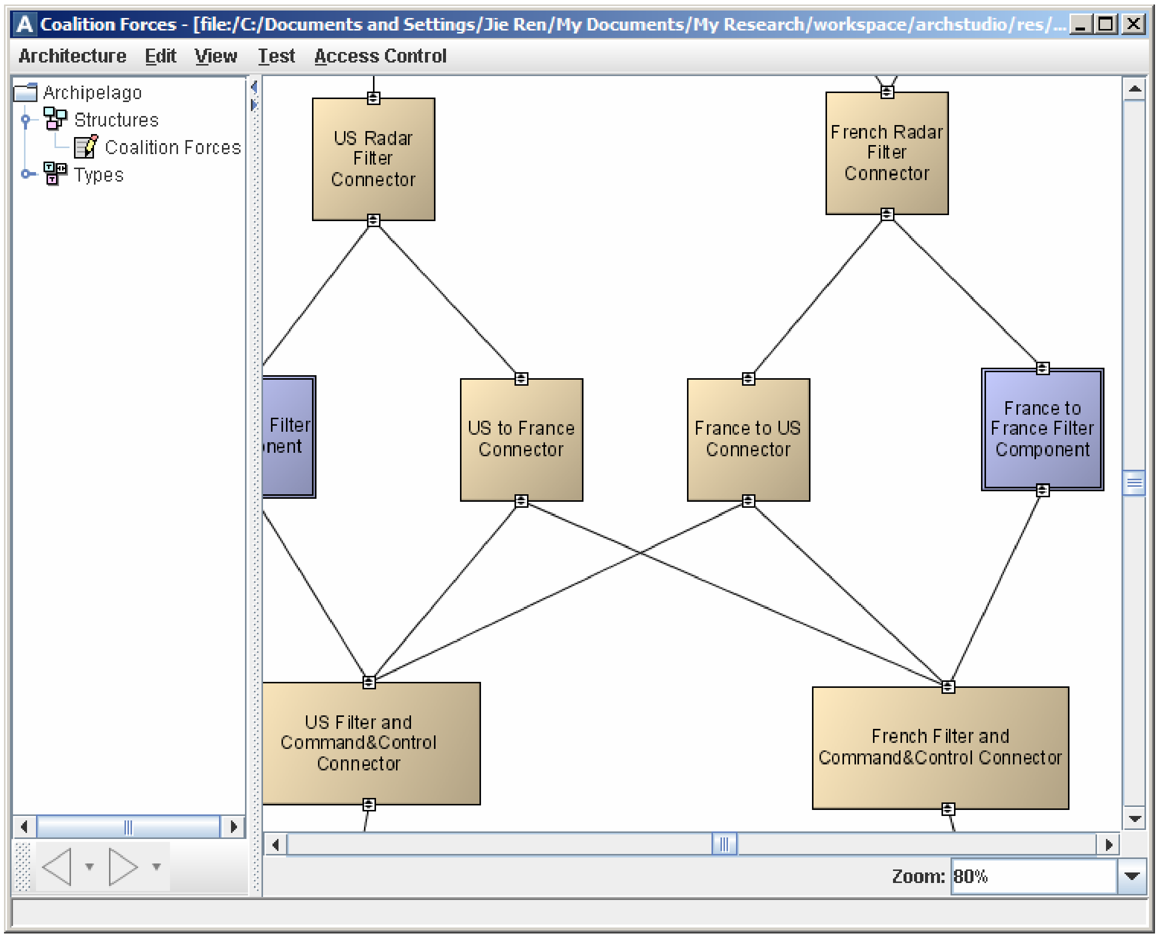

Coalition Forces - [file:/C:/Documents and Settings/Jie Ren/My Documents/My Research/workspace/archst...
Architecture
Edit
View
Test
Access Control
Archipelago
Structures
Coalition Forces
Types
US to France Filter Component
France to US Filter Component
US Distributed Fred Connector
French Distributed Fred Connector
US Local Fred Connector
French Local Fred Connector
US Filter and Command & Control Connector
French Filter an & Control Co
Zoom:
100%

Figure 6-2, Original Coalition

The advantage of this architecture is that it maintains a clear trust boundary between US and France. Since only the US to French Filter and the French to US Filter come across trust boundaries, they should be the focus of further security inspection. This architecture does have several shortcomings. Firstly, it is rather complex, This architecture uses 4 Fred connectors (US Local, US Distributed, French Local, and French Distributed) and 2 components (US to French Filter, French to US Filter) to implement secure data routing such that sensitive data only go to appropriate receivers. Secondly, it lacks conceptual integrity. It essentially uses filter components to perform data routing, which is a job more suitable for connectors. Thirdly, it lacks reusability, since each filter component has its own internal logic, and they must be implemented separately.

6.1.2 An Architecture with Two Secure Connectors



Coalition Forces - [file:/C:/Documents and Settings/Jie Ren/My Documents/My Research/workspace/archstudio/res/...
Architecture
Edit
View
Test
Access Control
Archipelago
Structures
Coalition Forces
Types
US Radar Filter Connector
French Radar Filter Connector
Filter
nent
US to France Connector
France to US Connector
France to France Filter Component
US Filter and Command&Control Connector
French Filter and Command&Control Connector
Zoom:
80%

Figure 6-3, Coalition with Two Secure Connectors

An alternative architecture uses two secure connectors, a US to France Connector and a France to US Connector. Both are based on the same connector type, `SecureC2Connector_type`. The US to France Secure Connector connects to both the US Filter and Command & Control Connector and the French Filter and Command & Control Connector. When it receives data from the US Radar Filter Connector, it always route it to the US Filter and Command & Control Connector. And if it detects that it is also connected to the French Filter and Command & Control Connector, and the data is releasable to the French side, then it also routes the messages to the French Filter and Command & Control

Connector. The France to US Secure Connector adopts the same logic. This architecture simplifies the complexity and promotes understanding and reuse: Only two secure connectors are used, these connectors perform a single task of secure message routing, and they can be used in other cases by adopting a different policy.

Figure 6-4 illustrates the relationship between the policy of a type and the policy of the instances of the type. The `SecureC2Connector_type` is specified with a principal of NATO, and has a policy that denies the instantiation operation (the `AddBrick` action) if the NATO principal is not present in the request for the operation. In the instance `UStoFranceConnector`'s policy, the type policy is referenced through `PolicySetIdReference`. The instance specifies that the instantiation operation is denied if the US principal is not present. Note that the policy combination algorithm for the instance's policy is deny-overrides. Thus, even if the US principal is present, unless the NATO principal is also present, the instantiation operation will be rejected by the type policy, and because of the deny-overrides algorithm, the denial by the type policy suffices to reject the instantiation operation. The other connector, `FrancetoUSConnector`, adopts a similar policy. This relationship between the type policy and the instance policy allows us to simply remove the type principal, NATO, from the type specification to disallow instantiation of any instances for this type. This flexibility in combining the type policy and the instance policy through the policy combination algorithm is one of the reasons that we choose XACML as the base policy language for Secure xADL (Section 3.2.4).

```
<connectorType id="SecureC2Connector_type"
                xsi:type="SecureConnectorType">
  <security>
    <principal>NATO</principal>
    <policies>
      <PolicySet PolicySetId="InstantiateConnectorType"
                 PolicyCombiningAlgId="deny-overrides">
        <Policy RuleCombiningAlgId="deny-overrides">
          <Rule Effect="Deny">
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>SecureManagedSystem
              <AttributeDesignator>subject-id
            <AnyResource />
            <ActionMatch MatchId="string-equal">
              <AttributeValue>urn:xadl:action:AddBrick
              <AttributeDesignator>action-id
            <Condition FunctionId="not">
              <Apply FunctionId="string-is-in">
                <AttributeValue>NATO</AttributeValue>
                <AttributeDesignator>principal
        </Policy>
      </PolicySet>
    </policies>
  </security>
</connectorType>
<connector id="UStoFranceConnector"
                xsi:type="SecureConnector">
  <type href="#SecureConnector_type" />
  <security>
    <principal>US</principal>
    <policies>
      <PolicySet PolicyCombiningAlgId="deny-overrides">
        <Policy RuleCombiningAlgId="deny-overrides">
          <Rule Effect="Deny">
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>SecureManagedSystem
            <ActionMatch MatchId="string-equal">
              <AttributeValue>urn:xadl:action:AddBrick
              <AttributeDesignator>action-id
            <Condition FunctionId="not">
              <Apply FunctionId="string-is-in">
                <AttributeValue>US</AttributeValue>
                <AttributeDesignator>principal
        </Policy>
        <PolicySetIdReference>InstantiateConnectorType
</connector>
```

Figure 6-4, Type Policy and Instance Policy

6.1.3 An Architecture with a Single Secure Connector

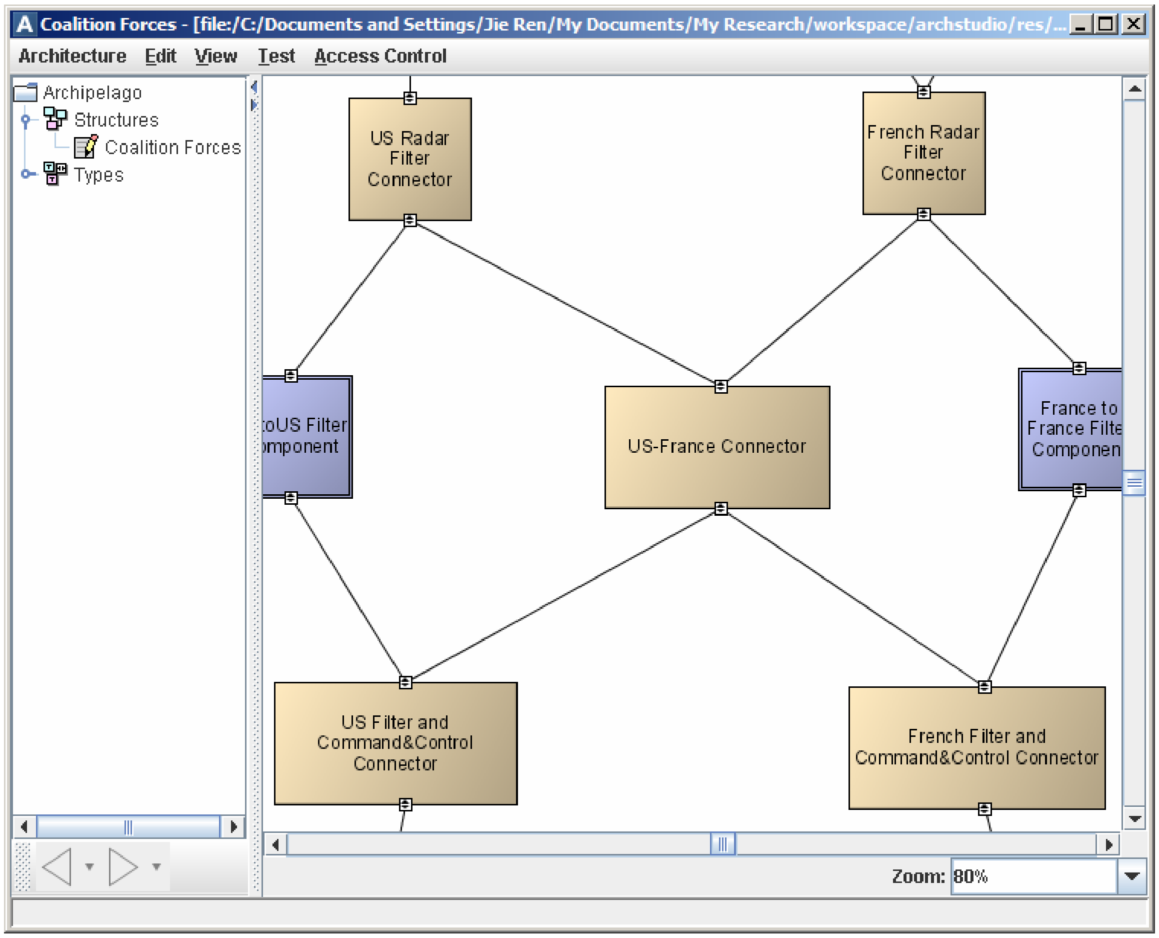

Coalition Forces - [file:/C:/Documents and Settings/Jie Ren/My Documents/My Research/workspace/archstudio/res/...
Architecture
Edit
View
Test
Access Control
Archipelago
Structures
Coalition Forces
Types
US Radar Filter Connector
French Radar Filter Connector
toUS Filter mponent
US-France Connector
France to France Filte Componen
US Filter and Command&Control Connector
French Filter and Command&Control Connector
Zoom: 80%

Figure 6-5, Coalition with One Secure Connector

Figure 6-5 depicts an architecture with a single secure connector. This simplifies the architecture description further, and has the conceptual clarity that a single connector is in charge of all communications between two parties that do not fully trust each other. The connector becomes the center of secure data sharing. A shortcoming of this architecture is that the secure connector can see all traffic, thus it is the obvious target for penetration, and its breach leads to secret leak. An architect should balance all such tradeoffs.

Since the single connector is the single bridge for sharing data, there are many manners to control the sharing by setting different polices on the connector.

The connector can be denied instantiation, thus no sharing will occur. Even if the connector is instantiated, the connections with other components and connectors can still be rejected, so no messages can be delivered and sharing still will not occur. When the connector is instantiated and properly connected with other constituents, it can still use its policy on internal message routing (Section 5.4.8) to decide what messages can be delivered.

Figure 6-6 specifies the internal message routing policy of the `UStoFranceConnector`. There are three noticeable features in this policy.

Firstly, the policy specifies a `Deny` rule that matches all requests. The rule combining algorithm for the policy is `permit-overrides`. The effect is unless a message is explicitly permitted to be routed, the connector will not forward the message. This achieves the “secure by default” effect by using a close policy.

Secondly, the connector uses content-based access control to deliver messages of certain types. The two rules use an XPath expression to specify that message routing will only happen when the “type” value of the message is either “Air Defense Missile” or “Fixed Military Wing”.

Lastly, the policies use roles to control message delivery. Two roles are defined, the US role and the France role. The “Air Defense Missile” message is delivered from France to US only when the connector acts as the France role. Likewise, the “Fixed Military Wing” message is delivered only when the connector acts as the US role. The connector can act under multiple roles, as is currently specified in the US and France principal. If the connector only acts as the US role, then the “Air Defense Missile” message will not be delivered. If the connector does not play any role, then no message will be routed.

```
<connector id="USFranceConnector"
                xsi:type="SecureConnector">
  <security>
    <principal>France</principal>
    <principal>US</principal>
    <policies>
      <PolicySet PolicySetId="InternalRouting"
                 PolicyCombiningAlgId="permit-overrides">
        <Policy RuleCombiningAlgId="permit-overrides">
          <Rule Effect="Deny" />
      <PolicySet PolicySetId="PPS:France"
                 PolicyCombiningAlgId="permit-overrides">
        <Policy RuleCombiningAlgId="permit-overrides">
          <Rule Effect="Permit">
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>USFranceConnector
              <AttributeDesignator>subject-id
            <ResourceMatch MatchId="string-equal">
              <AttributeValue>RouteMessage
              <AttributeDesignator>resource-id
            <ActionMatch MatchId="string-equal">
              <AttributeValue>xadl:action:RouteMessage
              <AttributeDesignator>action-id
            <Condition FunctionId="string-equal">
              <AttributeValue>Air Defense Missile
              <AttributeSelector RequestContextPath=
    "//context:ResourceContent/security:routeMessage/
     messages:namedProperty[messages:name='type']/
     messages:value/text()"/>
      <PolicySet PolicySetId="PPS:US"
                 PolicyCombiningAlgId="permit-overrides">
        <Policy RuleCombiningAlgId="permit-overrides">
          <Rule Effect="Permit">
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>USFranceConnector
              <AttributeDesignator>subject-id
            <ResourceMatch MatchId="string-equal">
              <AttributeValue>RouteMessage
              <AttributeDesignator>resource-id
            <Condition FunctionId="string-equal">
              <AttributeValue>Military Fixed Wing
              <AttributeSelector RequestContextPath=
    "//context:ResourceContent/security:routeMessage/
     messages:namedProperty[messages:name='type']/
     messages:value/text()"/>
</connector>
```

Figure 6-6, Role-based and Content-based Routing

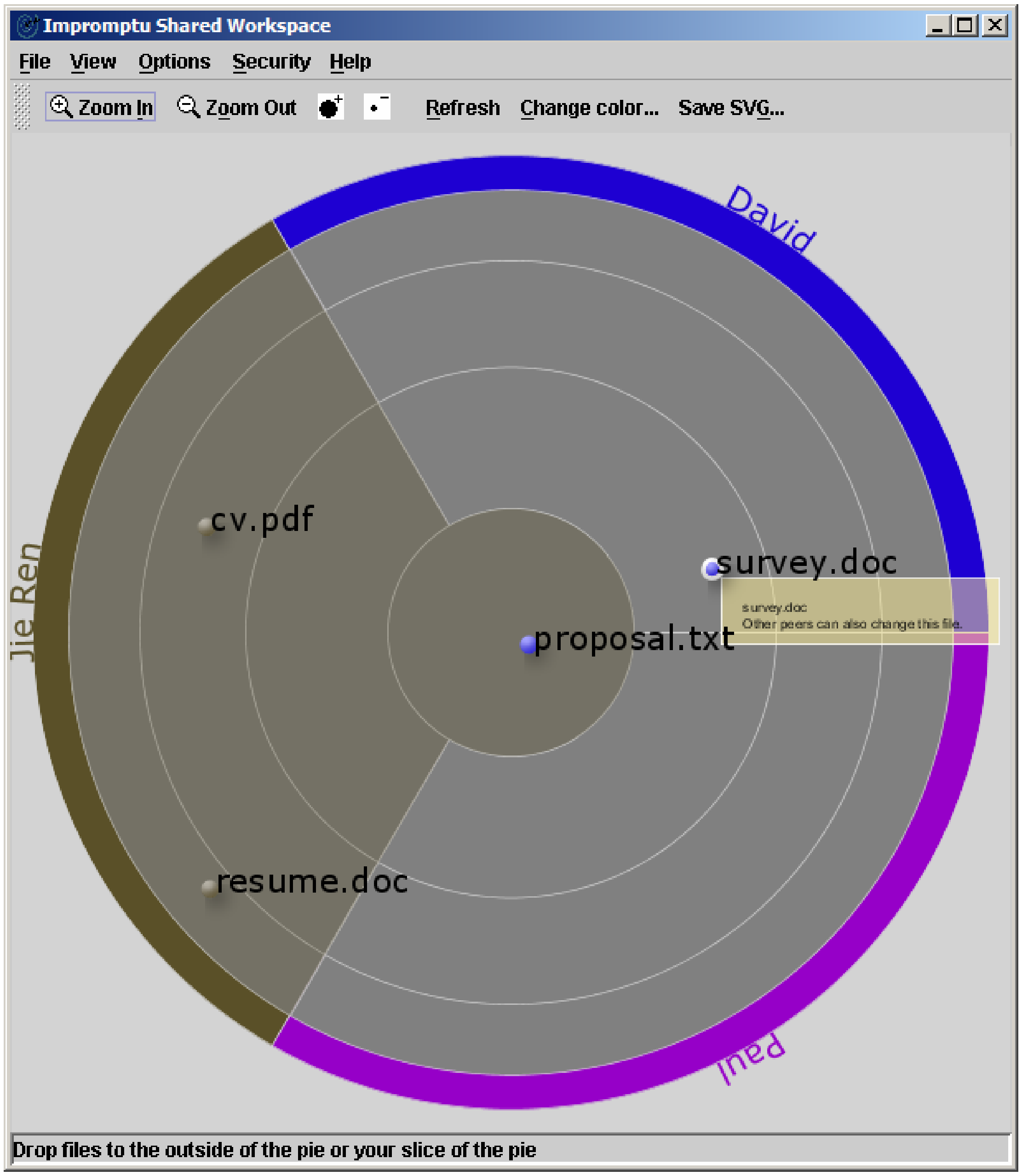
6.2 Impromptu

This section uses Project Impromptu, an application for sharing files in a local area network, to illustrate that the connector-centric approach can be used to develop composite secure connectors for componentized and networked software. In Section 6.2.1, we give an overview of the project, specifying the general context in which we make design decisions about security. Section 6.2.2 enumerates software components of the system and establishes security goals. Section 6.2.3 describes how a secure connector connects these components to accomplish security goals. Lastly, Section 6.2.4 illustrates how the secure connector can be replaced by another composite secure connector that are more secure and standard compliant.

6.2.1 Overview of Project Impromptu

Project Impromptu is a subproject of Project Swirl [31]. The hypotheses of the Swirl Project are as follows. Firstly, traditional security mechanisms must be utilized in a user-centered context to provide effective security for users. Secondly, users make security related decisions within a context. Different contexts require different degrees of security. Thirdly, users' perceptions of the context can be facilitated by visualizing security related events that come from heterogeneous sources. Finally, perceptions and decisions related to security should be well integrated with users' main tasks.

Project Impromptu develops an ad-hoc file sharing application as a test bed to investigate and evaluate these hypotheses. Each Impromptu user can share files and decide how the shared files can be accessed by other users. A file can be "see-only", which means other users can only know its existence but cannot access its content. A file can be "read-only", where other users can read its content but cannot modify it. A file can also be "read-write", allowing other uses to read and modify its content. Finally, a file can be "persistent", which will still exist for read/write access even after the original owner has left the ad-hoc sharing group.



survey.doc
Other peers can also change this file.

Drop files to the outside of the pie or your slice of the pie

Figure 6-7.Impromptu User Interface

Figure 6-7 depicts what a user will see when Impromptu launches. The "pie" designates the entire ad-hoc file sharing group. Each slice of the pie represents a participant. The participant representing the current executing user is highlighted by the darker shaded slice. Each dot is a shared file. The position of the file determines the sharing level for each file. From the outermost ring inward, each ring represents "see-only", "read-only", and "read-write", respectively. The center circle collects all "persistent" files.

6.2.2 Architectural Components and Connectors

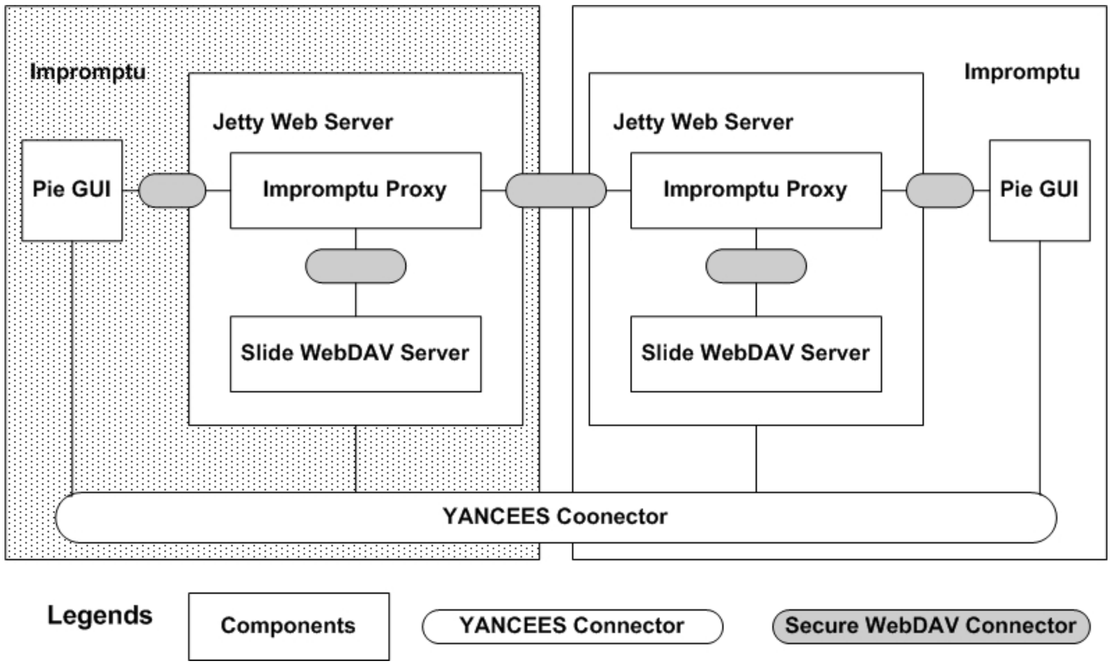


Figure 6-8, Impromptu Architecture

Internally, the Impromptu application consists of the following components: the graphical user interface, the Jetty web server, the Impromptu WebDAV proxy, and the Slide WebDAV repository. The secure WebDAV connector and the YANCEES [37] event notification connector connect these components together. The architecture is graphically depicted in Figure 6-8. Jetty and Slide are external open source software components. The user interface component, the proxy component, the secure WebDAV connector, and the YANCEES connector are developed by us.

The YANCEES connector provides a high-level event notification channel. This connector delivers relevant events to interested subscribers. These events include functionality related events, such as an indication that a file is created, and security related events, such as that the file's sharing level has been changed from "read-only" to "read-write".

Jetty serves as a dynamic application server that allows an add-on component to decide what a response will be when Jetty receives a request. Slide is an add-on component that provides WebDAV [24] repository support. WebDAV is an HTTP extension that provides Internet-scale resource storage, retrieval, and modification capability. It is an open standard, easily available in different platforms, and is thus chosen as the foundation storage for the ad-hoc file sharing application.

Participants store their own files in their own Slide server. However, this local storage is not directly seen by the participant. A user only interacts with the Impromptu proxy, using the Pie GUI depicted in Figure 6-7. The proxy provides an illusion of a unified, shared file storage work space. When an Impromptu proxy receives a file operation request, it determines whether the request is directed at a local file or a remote file belonging to another participant. In the former case, it retrieves the file from the local Slide server. In the latter case, it issues a standard WebDAV request against the remote Impromptu proxy, which will accomplish the operation using its own local Slide server.

We designed this application for a relatively friendly, ad-hoc file sharing environment. The participants are assumed to be not malicious, and the major risk in such an environment is unintentional disclosure of information. In traditional file sharing applications, when a user operates on files it is not always clear to the user what files are shared, how they might be accessed and changed, and who is currently reading and changing files. However, neither do we want to require a user to use complex configuration operations to express secure file sharing intentions. Such complexity might be overwhelming to the user, and thus affect usability. In summary, the security goals for the Impromptu file sharing application are 1) make security visible; 2) ease security configuration.

As can be clearly seen from Figure 6-8, the secure WebDAV connector is the key communication mechanism that connects the Slide server, the Impromptu proxies, and the GUI. The next two sections outline how two generations of secure WebDAV connectors achieve these goals.

6.2.3 Connector Using IP Address Authentication

Our first WebDAV security connector employs an IP address-based authentication scheme and a method-based authorization mechanism. The connectors connect the local Impromptu proxy and the Slide server, which store files that should be secured, and also connect the GUI and the remote Proxy, which access secured files. The security architecture of a single Impromptu system is described in Figure 6-9, using Secure xADL. The secure WebDAV connector type extends a base xADL connector type, `ConnectorType`, using the extensible feature of the xADL language. Three instances of the secure WebDAV connector type connect related components.

```
<connectorType type="ConnectorType"
  id="SecureWebDAVConnector">
    <signature id="WebDAVClient"></signature>
    <signature id="WebDAVServer"></signature>
    <description>
        IP-based authentication
        Method-based authorization
    </description>
</connectorType>
<component type="ProxyType" id="Local">
    <principal>me</principal>
</component>
<component type="ProxyType" id="Remote">
    <principal>other</principal>
</component>
<component type="GUIType" id="GUI">
    <principal>me</principal>
</component>
<component type="SlideType" id="Slide">
    <principal>me</principal>
</component>
<connector type="SecureWebDAVConnector"
    id="GUI_Impromptu">
    <interface signature="#WebDAVClient"
        id="GUI"/>
    <interface signature="#WebDAVServer"
        id="Impromptu"/>
</connector>
<connector type="SecureWebDAVConnector"
  id="Impromptu_Impromptu">
    <interface signature="#WebDAVClient"
      id="Remote"/>
    <interface signature="#WebDAVServer"
      id="Local"/>
</connector>
<connector type="SecureWebDAVConnector"
  id="Impromptu_Slide">
    <interface signature="#WebDAVClient"
      id="Local"/>
    <interface signature="#WebDAVServer"
      id="Slide"/>
</connector>
```

Figure 6-9, Secure WebDAV Connector

The connector connects a WebDAV client and a WebDAV server. It employs two security facilities. Firstly, the connector uses an IP address-based authentication mechanism to separate a local client from a remote client. When the connector receives a WebDAV operation request from the client, it determines, using the IP address of the client, whether the request comes from the same machine as the server (thus from the local participant), or from a different machine (thus from a remote participant). In the former case, the client component will execute as the local principal, “me”. In the latter case, the client component executes as the remote principal, “other”. For example, in Figure 6-9, connector `GUI_Impromptu` connects the GUI and the local Impromptu. The GUI executes as the “me” principal because it executes on the same machine as the local Impromptu. The connector `Impromptu_Impromptu` connects two Impromptu proxies. The remote Impromptu proxy executes under the “other” principal because it resides on a different machine than that of the local Impromptu proxy. Any non local participants will execute as the “other” principal. This is the Role-based Access Control (RBAC) model discussed in Section 4.1, where components executing for different remote subjects (participants of the sharing session) have the same “other” role.

Secondly, the connector uses both the principal and the file sharing level to decide what WebDAV methods a client can perform against that file. The local GUI component, executing as the “me” principal, can do anything towards local files. A remote participant, executing as the “other” principal, is subject to the sharing level of a file. This decision process is transparent to a user, so there is no need for the user to finish any complex setups. If a file is shared as “see-only”, the connector will only allow the WebDAV PROPFIND method to pass from the client to the server. This method permits other participants to retrieve information about the file such as the creation date, the resource type, etc. For a "read-only" file the connector permits, in addition to the PROPFIND method, the WebDAV GET method, enabling a remote participant to get the content of a file. Finally, the secure connector permits the WebDAV PUT method for a "read/write" file, so a remote user can store back modifications for a retrieved file.

We have conducted an initial user study to assess whether the proposed security architecture can achieve the security goals [31]. The preliminary results of this study suggest that the system gives users a clearer sense of perception and manipulation of security, and it does not overwhelm users with technical details.

6.2.4 Standard-Compliant Composite Connector

After the initial investigation of Project Impromptu, we have revised the security architecture to address some issues we encountered during the investigation.

Firstly, we want to deploy the Impromptu system into handheld devices, which might require a more secure authentication connector. The current Impromptu software is a tightly integrated suite of components, some of which might require too many resources to execute on handheld devices. A possible solution is to only execute the GUI on the handheld device (so we can still investigate how users perform their regular and security-related tasks on such hardware platforms), and deploy the rest of the Impromptu software on more powerful machines. Under such a configuration, the IP address-based authentication mechanism is insufficient, because even requests from the owning participant (who is using a handheld device) actually comes from a different IP address. We adopt a more secure authentication connector, the HTTP digest authenticator. Such a connector enables deploying the Impromptu system to an environment that might contain malicious adversaries, and mitigate some limitations of an address-based authentication mechanism [7].

Secondly, we want to utilize existing authorization mechanisms supported in Jetty and Slide to enable richer authorization semantics. Utilizing existing mechanism enables better integration with mechanisms provided by the Jetty application server and the Slide WebDAV server, and leverages the standard WebDAV ACL [24] access control features provided by Slide.

Thus, we have developed a second secure connector using standard-compliant authentication and authorization mechanisms to replace the original secure WebDAV connector. This new secure connector is a composite connector. It consists of a digest authentication connector, a standard HTTP authorization connector using web.xml deployment descriptor [18], and a standard WebDAV ACL authorization connector. Using the sub-architecture support of Secure xADL, the composite connector is described in Figure 6-10. This connector will only allow an HTTP WebDAV request to succeed when all the constituent connectors grant permissions for the request. That is, the request will only succeed when it can pass the authentication challenge from the Digest Authenticator, when the requested method is allowed in the web.xml connector, and when the requested resource is permitted by WebDAV ACL permissions.

One architectural advantage enabled by this new connector is that the standard-compliant connector allows the Jetty/Slide repository to be accessed by the built-in WebDAV file system support in Windows XP and Mac OS X, so now users can share arbitrary files and manipulate the shared files through any applications, such as text editors that do not know how to directly handle WebDAV requests. In the first secure connector architecture, users can only share and change files that can be manipulated by WebDAV-aware applications, such as recent versions of Microsoft Office.

```
<archStructure id="composite">
    <connector id="DigestAuthenticationConnector">
        <interface id="DA_input" />
        <interface id="DA_output" />
    </connector>
    <connector id="WebXMLAuthorizationConnector">
        <interface id="WebXML_input" />
        <interface id="WebXML_output" />
    </connector>
    <connector id="WebDAVACLConnector">
        <interface id="DAVACL_input" />
        <interface id="DAVACL_output" />
    </connector>
    <link>
        <point id="#DA_output">
        <point id="#WebXML_input">
    </link>
    <link>
        <point id="#WebXML_output">
        <point id="#DAVACL_input">
    </link>
</archStructure>
<connectorType id="SecureWebDAVConnector">
    <signature id="client" />
    <signature id="server" />
    <subArchitecture>
        <archStructure href="#composite" />
        <signatureInterfaceMapping>
            <outerSignature href="client" />
            <innerInterface href="DA_input" />
        </signatureInterfaceMapping>
        <signatureInterfaceMapping>
            <outerSignature href="server" />
            <innerInterface href="DAVACL_output" />
        </signatureInterfaceMapping>
    </subArchitecture>
</connectorType>
```

Figure 6-10, Composite Secure WebDAV Connector

6.3 *Firefox Component Security*

Firefox is an open source web browser first released in November 2004. Its development started in 2002, shortly after the first official release of the Mozilla Application Suite, which had been under development since 1998. Firefox is a simplified version of the browser from the suite, but its enormous success has made it to replace the application suite as the main product produced from the Mozilla organization.

Firefox is a very large open source project. The source code consists of about 10 thousand C/C++/JavaScript files. The files include about 5 million lines of code. This section demonstrates how our approach can be applied to model the component security architecture of this very complex, componentized and networked software system.

6.3.1 Firefox Architecture

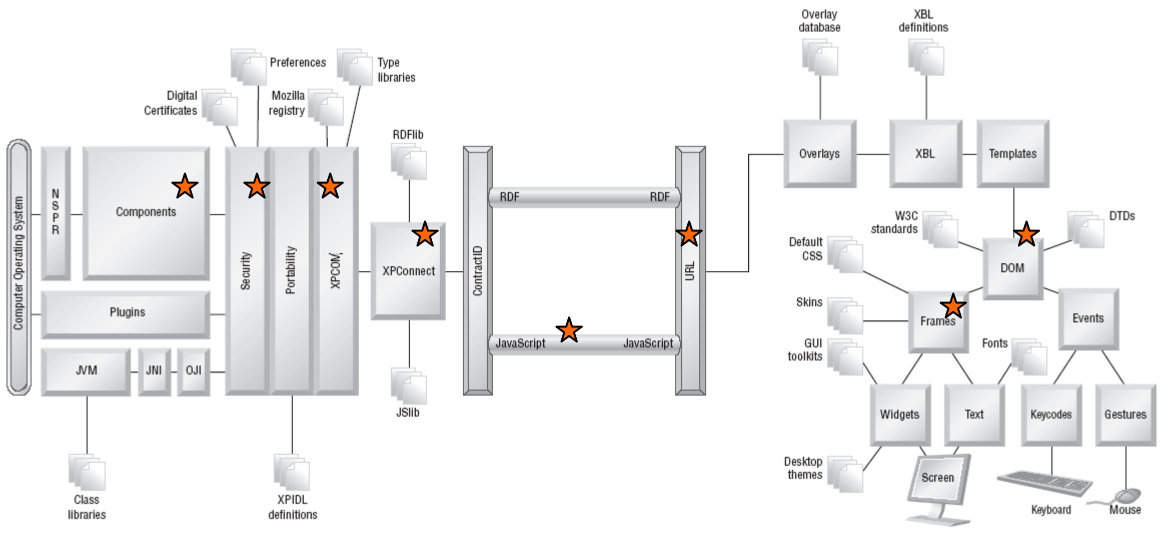


Figure 6-11, Firefox Architecture, from [88]

Figure 6-11, from [88], gives a high-level picture of the Firefox architecture. This diagram is only one possible representation of the complex internal interactions, and it is highly simplified. As can be seen from this architectural overview, the browser has two major halves: the front end on the right and the back end on the left. The front end handles presentation of the visible content loaded from the Internet and interacts with the user through events. The back end deals with the underlying services, such as reading files and storing user privacy information.

Roughly speaking, the component security architecture modeled in this section handles how to prevent the front end, which originates from Internet sources that are not necessarily trustworthy, from unduly accessing the back end, where important user information is kept. The problem here is an architectural access control problem, and our approach can be used to model how the Firefox solution works. The component security architecture touches the constituent blocks in the above diagram that are marked with stars: DOM, Frames, URL, JavaScript, XPConnect, XPCOM, Security, and Components.

The Firefox security architecture contains another part, the Public Key Infrastructure (PKI) support. This case study chooses not to model that part, since the part mostly handles cryptography and digital certificates. Firefox also suffers from usual buffer overrun vulnerabilities. Most of these vulnerabilities come from the mail handler and the image processing component. Such vulnerabilities are not modeled by this case study, either.

In the following subsections, we first present the platform technologies of Firefox, and then we delineate the most important boundaries in browser security: the boundary between the chrome and the content and the boundary between contents from different origins. After that, we establish the principals as the foundation of Firefox component security, and how they are represented in top level containers and individual DOM nodes. Then we outline the security policies, and inspect how they are enforced by the script security manager. We also brief how security is handled in URI protocol handlers. Finally, we summarize the modeling of architectural access control with our connector-centric approach, and discuss some noteworthy issues.

6.3.2 Platform Technologies: XPCOM, JavaScript, and XPConnect

Firefox/Mozilla intends to be not just an application but also a development platform on which more third-party applications can be built. There are three core architectural technologies for this platform [52]: XPCOM, JavaScript, and XPConnect.

XPCOM is a cross platform component model. It maintains binary compatibility with the Microsoft COM component model, but is portable across different operating systems. Major functional components of Firefox, such as networking and layout, are built with this model. Each component has a component type. A component type's interfaces are described by a cross platform interface definition language, XPIDL. Each component should only be accessed by other components through these well defined interfaces. Many component types have only one instance, and this instance is running as a service in the platform. Other components can request a service to perform actions.

JavaScript [39] has long been used by web developers for authoring dynamic web pages. Firefox uses JavaScript extensively in programming different functionalities of the browser, especially the user interface elements, such as dialogs and drag-n-drop handling. Third party extensions for Firefox are also mostly developed in JavaScript. Unlike Java, the language definition of JavaScript does not specify how security should be handled, so the security features implemented in Firefox must balance how various language features interact with security requirements.

XPConnect provides bidirectional communication capabilities between XPCOM and JavaScript. It allows a component built by XPCOM, the native component, to be accessed as a JavaScript object. It also permits a JavaScript object to be accessed by a native XPCOM component as an ordinary XPCOM component. As we shall see, this is the architectural connector where security check is conducted.

6.3.3 Trust Boundary between Chrome and Content

When a user uses the Firefox browser to browse the web, the visible window contains two areas. The chrome, which consists of decorations of the browser window, such as the menu bar, the status bar, and the dialogs, are controlled by the browser. The browser needs to perform arbitrary actions to accomplish the intended task, and it is also trusted to perform such actions. Borrowing the chrome term that originally refers to the user interface elements, the browser's code is called the **chrome code**. Such code can perform arbitrary actions. Any installed third party extensions also become chrome code.

The other area, the content area, is contained within the browser chrome. The content area contains content coming from different sources that are not necessarily trustworthy. Some contents contain active code that leads to executing JavaScript scripts. Such **content code** should not be allowed to perform arbitrary actions unconditionally and must be confined accordingly.

Otherwise they could abuse the unlimited privileges to damage users. This boundary between the chrome code and the content code is the most important trust boundary in Firefox.

Because of the architectural choice of using XPCOM, JavaScript, and XPConnect to develop the Firefox browser and extensions, both chrome code and content code written in JavaScript can use XPConnect to access interfaces of XPCOM components that interact with the underlying operating system services. The XPCOM components are represented as the global `Components` collection in JavaScript. This access process is the architectural access control process discussed in Section 3.1. XPConnect, as the connector between the possibly untrustworthy accessing code and the accessed XPCOM components, should protect the XPCOM interfaces and decide whether the access should be permitted.

6.3.4 Trust Boundary between Contents from Different Origins

Another trust boundary is between contents coming from different origins. The origin of content is defined by the protocol, the host name, and the port used to retrieve the content. Contents differ in either the protocol, the host name, or even the port would be considered of different origins. Users generally browse many different sites, and any page can contain contents from different origins. The content coming from one origin should only be able to read or write content coming from the same origin. This is called the **same-origin** policy. Otherwise, a malicious page from one origin could use this cross domain access to retrieve or modify sensitive information for another origin, such as the password that the user uses for authentication with the other origin. This process is another architectural access control process, where interfaces of one content component from one origin should not be unduly accessed by another content component from another origin.

The trust boundary between the chrome code and the content code and the trust boundary between the content code and contents of different origins are the main trust boundaries maintained by the Firefox browser. Such boundaries can be loosened by users. Users can grant the `UniversalXPConnect` privilege to signed content code, essentially giving such content code full privileges as chrome code. User can also fine tune what accesses content code from different origins can have on different interfaces of XPCOM components.

6.3.5 Principals

Since the JavaScript language does not specify how security should be handled, the Firefox JavaScript implementation defines a principal-based security infrastructure to support enforcing the trust boundaries. There are two types of principals. When a script is accessing an object, the executing script has a **subject principal**, and the object being accessed has an **object principal.**

Firefox uses principals to identify code and content coming from different origins. Each unique origin is represented by a unique principal. The principal in Firefox corresponds to the Subject construct in Secure xADL (Section 3.1.1), and such Subjects are used to regulate in architectural access control, as will be discussed in Section 6.3.8.

There is one special subject principal, the system principal. All chrome code components and resources are identified by the system principal. A special case of the system principal is the null principal, where a principal cannot be found. This null principal is treated as equivalent of the system principal. Code executing under the system principal, i.e. chrome code, can perform arbitrary actions.

6.3.6 Container: Document and Window

When a user browses the web, generally the Firefox browser loads each HTML document into a window. This document/window pair performs a very important role in executing content JavaScript code. The JavaScript language definition defines an **execution context** that specifies the semantics of executing JavaScript programs. Firefox's implementation of this context is called `JSContext`. Among other things, the context maintains a run-time stack for executing JavaScript functions. The role of this stack will be discussed in Section 6.3.8. An execution context maintains a **scope chain** for each object. This scope chain decides how identifiers referenced through the object are resolved. At the end of the scope chain is the **global object**, where predefined types in the JavaScript language, such as Object and Date, are defined, and thus all references to Object and Date will be eventually resolved by the global object. In the Firefox JavaScript definition, both the execution context and the global object are attached to the document/window pair.

The document in the window is also the ultimate source of the security credentials for the loaded content. In the principal-based security infrastructure of Firefox, the document (thus the window) maintains a principal that is constructed based on the origin URI of the document.

A special type of window is the frame window, when an HTML document uses a frame set to include a set of frames, and each frame window can contain an HTML document that comes from a different origin. A frame window, even contained within another window, has its own principal that is based on the origin of the document that it contains. This principal may be different than the principal of the top level window. A frame window can further contain its own frame sets, so the frames within a top level window form a containment hierarchy.

6.3.7 DOM Node

When a browser loads an HTML document and presents it in a window, it creates a document object model (DOM) tree internally. Each element contained within the HTML document is represented by a node. JavaScript scripts contained within these documents manipulate the nodes to achieve their purposes. Some other functionalities of the browser, such as the navigation history and the top level window, are also represented as DOM nodes, so the JavaScript code can uniformly manipulate them. Specially, the underlying native XPCOM components of Firefox are represented as the global `Components` JavaScript collection. Writing to one property of one node could result in significant changes. For example, changing the location property of a window object to a new URI will instruct the window to load a new document from the URI into that window, and manipulating through `Components` is actually operating on the underlying operating system services.

From a security viewpoint, each node has a principal that identifies the origin of the node. If the principal is not explicitly specified for a node, then the node will inherit the principal from the top level container.

Each DOM node belongs to a class. For example, all nodes created for HTML form elements belong to the Form class. The classes of DOM nodes can be used to fine tune the same origin policy. Firefox allows user to define whether scripts from an origin can set or read different properties of various classes of DOM nodes.

Each class has a `DOMClassInfo` that represents relevant information for accessing the properties of the nodes. For example, since changing the location property of document and window nodes results in navigating to a different place, the `DOMClassInfo` for the Window and Document classes advertise that these changes should be checked for security. `DOMClassInfo` provides a general mechanism that architectural components can present their security requirements. It is similar to the secure bricks of the c2.fw.secure framework defined in Section 5.4.2, where the existence of a secure brick can be used to trigger the decision on whether a security check is needed (Section 5.4.7).

6.3.8 Enforcing Security: Security Manager

To ensure the proper trust boundaries in the architectural access control operations, the XPConnect architectural connector uses a security manager to control both the access between content code and chrome code and the access between content code of one origin and content nodes of other origins. In this subsection we first investigate how the security manager discovers relevant principals, and then discuss how the security manager controls different types of architectural operations: access of DOM properties and functions, instantiation through creation, and instantiation through `loadURI`.

Discover Object Principals and Subject Principals. Each JavaScript object in Firefox is associated with an object principal that specifies the creator of the object. Each top level window has a principal, which is created based on the URI that is used to load the document and create the top level window. Each

DOM node contained within the window can also be tagged with a principal. If the node does not have an explicit principal, then it will inherit the principal of its container. For most nodes this will be the principal of the top level window. For nodes contained within a frame window, this principal is the principal that is associated with the frame window. The node's principal is the object principal that is used when the properties of this node are read or written.

One special type of object principal, the principal associated with a function object, is the subject principal that designates the subject executing the function. To find the subject principal at run-time, the security manager inspects the run-time stack of the `JSContext`, and uses the principal from the inner most stack frame. If a principal cannot be found at the inner most stack frame, then the search follows the stack frame chain to find the outer calling frames. Since the stack frame chain of a `JSContext` always ends up with the principal of the global object, in most cases this principal, associated with the document/window pair, is the subject principal of the executing code. If this principal is the system principal, then the code is assumed to be chrome code. If no such principal can be found, it is assumed that the chrome code, possibly native C++ components, is executing and has not set up a principal yet.

DOM Access. When some code tries to read a property of a DOM node, write a property of a DOM node, or call a function of a DOM node, the security manager discovers both the object principal and the subject principal and decides whether the access should be granted. Because of the integration supported by XPConnect, the accessing components could be actually either JavaScript or C++, and the accessed component could be either simple DOM node or wrappers of native XPCOM services.

If the subject principal is either a system principal or an equivalent null principal, then the security manager allows the DOM access. If the subject principal is a regular principal, and this principal is granted the `UniversalXPConnect` privilege, then the access is also granted. Otherwise, the security manager inspects the object principal, and only grants the access if both the subject principal and the object principal are the same. Since a regular principal differs from the system principal and regular principals for different origins differ from each other, the security manager uses the principal infrastructure to effectively enforce both the trust boundary between chrome code and content code and the trust boundary between contents of different origins.

Instantiation by Creation. To further protect accessing XPCOM native components, the security manager also checks the following types of access attempted by a JavaScript script: when the script tries to get a service, when the script tries to create an instance of a XPCOM component type, and when the script tries to create a wrapper to such an instance. The operations of getting a service, creating an instance, and creating a wrapper are all architectural instantiation operations (Section 4.4.1), and the security manager decides whether the attempted architectural components should be created for the running architecture so the JavaScript script can accomplish its intended tasks.

The security manager inspects the subject principal of the script to decide whether the attempted instantiations should be permitted. Chrome components that execute as the system principal are always allowed. Content components signed by digital certificates are also granted such permissions if the user approves the signed scripts and grant the `UniversalXPConnect` privilege to the specific regular principal.

Instantiation by LoadURI. Before a window can load the content from one URI, a security check is performed by the security manager on whether such loading is allowed, based on the target URI and the loader's URI. Because of the open nature of Web, generally any loading should be permitted. Specially, loading a URI of one scheme within a window that is originally loaded from the same scheme is always allowed by the security manager. However, loading from different schemes might be limited or rejected depending on the schemes and the loading situation. For example, a web page loaded through an http URI cannot load a pop3 or imap URI, because the pop3 or imap URI could encode deleting a folder and blindly loading them would damage users.

This check on loading URI is used in many places in the browser. Part of the Firefox browser is written directly in JavaScript, contained in the browser.js file. The file has three types of security checks: checking whether a URI can be loaded when loading an image, checking whether a dialog can load a URI, and check whether a URI can be loaded when it is dropped onto the browser. The JavaScript code in browser.js uses XPConnect to get the security manager, and consults it to see whether the URI can be loaded.

From an architectural access control viewpoint, `loadURI` is similar to the architectural instantiation operation discussed in Section 4.4.1. When such a URI is permitted to load, Firefox essentially create a new component with a new principal and a new JavaScript execution context in the existing architecture. The security manager of the XPConnect connector decides whether such architecturally important operations should be allowed.

A special type of `LoadURI` occurs when loading documents in frames. Each document, whether is contained directly in a top-level window or contained in a frame window, is contained in a `DocShell`. The `DocShell` has methods that check whether it can load a URI or a set of URIs in a frame set. Because of historical reasons, the script within one frame can get a reference to another frame that is contained under the same root `DocShell`. The security manager monitors this reference to prevent possible exploits. For example, when the script within Frame A tries to load contents into Frame B using the reference, the principal of Frame A must be the same as that of Frame B, or be the same as the principal of any of the ancestors of Frame B in the frame containment hierarchy. This policy is different than the typical loading policy, where generally a source can load any target. It resembles the same origin policy used with DOM access.

6.3.9 Transport: URI, Channel, Protocol Handler

One service of the Firefox browser, the IO service, manages the network transportation of retrieving content from the Internet. When given an URI, the IO service analyzes the URI, uses the scheme to locate a protocol handler that can handle the URI, and asks the protocol handler to create a channel for the URI. The IO service then uses the channel to finish loading the content.

There are many types of protocol handlers. Besides the most obvious ones, such as protocol handlers for http, https, and ftp, Firefox also uses other types of pseudo handlers to accomplish other tasks. For example, there are a “view-source" protocol to view the source of a loaded HTML document and a "wyciwyg" protocol ("what you cache is what you get") to retrieve the cached copy of a rendered content.

From the security viewpoint, there are two interesting types of pseudo protocol handlers. One type is the "about" protocol. An "about:config" URI allows a user to change the configuration of Firefox, and an "about:credits" URI lists the contributors to the open source project. When a window is first initialized, it generally has the "about:blank" URI, which maintains no special privileges. The different about URIs are handled by different about protocol modules. Each module redirects the about URI to a real URI. For example, the "about:config" URI is redirected to a chrome page where user can use the chrome code to change the configurations of the browser. Some of the redirections lead to chrome pages, but the corresponding about module drops the chrome privilege by setting the owner of the about channel to the principal based on the URI redirected to, so future content loaded in the same window will not accidentally obtain unnecessary privileges.

Another interesting protocol handler handles the "javascript" pseudo protocol. In such a URI, the scheme is followed by a JavaScript snippet. When a javascript URI is loaded (or navigated to), instead of retrieving some regular contents, the `JSProtocolHandler` creates a `JSChanenl`, and the `JSChannel` uses a `JSThunk` to actually evaluate the script specified in the URL. So, if a URI supplied by content is used by the chrome code and the chrome code blindly loads the URI without first checking whether such a URI is a javascript URI, then a malicious hacker can supply a javascript URI and the JavaScript will be loaded and thus executed by the chrome code, with full chrome privileges. This is the reason for vulnerabilities reported in [45].

6.3.10 XPConnect as the Architectural Connector

As have been seen, XPConnect is an architectural connector in Firefox, and the security manager coordinates critical architectural operations: it regulates the access by scripts running as one principal to objects owned by another principal (if the subject principal is not the system principal, then both principals should be the same for the access to be allowed), it decides whether a native service can be created, obtained, and wrapped (one type of architectural instantiation operation), and it also arbitrates whether a URI can be loaded by in a window (another type of architectural instantiation operation).

The Firefox trust boundary policies can be specified using Secure xADL as in Figure 6-12. Each component has a subject, which is decided by the origin URI of the component. Each component also has a principal (the Secure xADL principal) to specify its role. There are two basic roles, the chrome role and the content role. The first policy specifies that a component executing as the chrome subject and the chrome role can perform any actions over any resources. The second policy specifies that a component executing as the content role can only access components of the same subject, which are components from the same URI. The policy also specifies that signed content components, executing under a signed content subject, can be granted the chrome role. This is a type of Role-Based Access Control policy (Section 4.1.2), where a role can be played by different subjects. In Firefox both chrome components and signed content components can act as the chrome role.

```
<component id="ChromeCode">
    <security>
        <subject>ChromeCode</subject>
        <principal>Chrome</principal>
<component id="ContentCode">
    <security>
        <subject>URI</subject>
        <principal>Content</principal>
<component id="SignedContentCode">
    <security>
        <subject>SignedURI</subject>
        <principal>Chrome</principal>
<connector id="XPConnectSecurityManager"
                xsi:type="SecureConnector">
  <security>
    <policies>
      <PolicySet PolicySetId="PPS:Chrome"
                 PolicyCombiningAlgId="permit-overrides">
        <Policy RuleCombiningAlgId="permit-overrides">
          <Rule Effect="Permit">
            <Subjects>
              <Subject>
                <SubjectMatch MatchId="string-equal">
                  <AttributeValue>ChromeCode
                  <AttributeDesignator>subject-id
              <Subject>
                <SubjectMatch MatchId="string-equal">
                  <AttributeValue>SignedURI
                  <AttributeDesignator>subject-id
            <AnyResource />
            <AnyAction />
      <PolicySet PolicySetId="PPS:Content"
                 PolicyCombiningAlgId="deny-overrides">
        <Policy RuleCombiningAlgId="deny-overrides">
          <Rule Effect="Permit">
            <SubjectMatch MatchId="string-equal">
              <AttributeValue>URI
              <AttributeDesignator>subject-id
            <ResourceMatch MatchId="string-equal">
              <AttributeValue>URI
              <AttributeDesignator>resource-id
            <ActionMatch MatchId="string-equal">
              <AttributeValue>AccessDOM
              <AttributeDesignator>action-id
          <Rule Effect="Deny">
</connector>
```

Figure 6-12, Firefox Security Policy

Using our connector-centric approach, the Firefox component security architecture can be described with the architectural description shown in Figure 6-13. Interfaces of the native XPCOM components, executing with the chrome role, are accessible from other chrome components but should be protected from other content components. The XPConnect connector maintains this boundary between content code and chrome code. The content components from one origin, including the containing window or frame and the DOM nodes contained within them, form one sub-architecture (Section 3.4.3). Their interfaces can be manipulated by chrome components, but should be protected from content components from other origins. The XPConnect connector maintains this boundary of same origin.

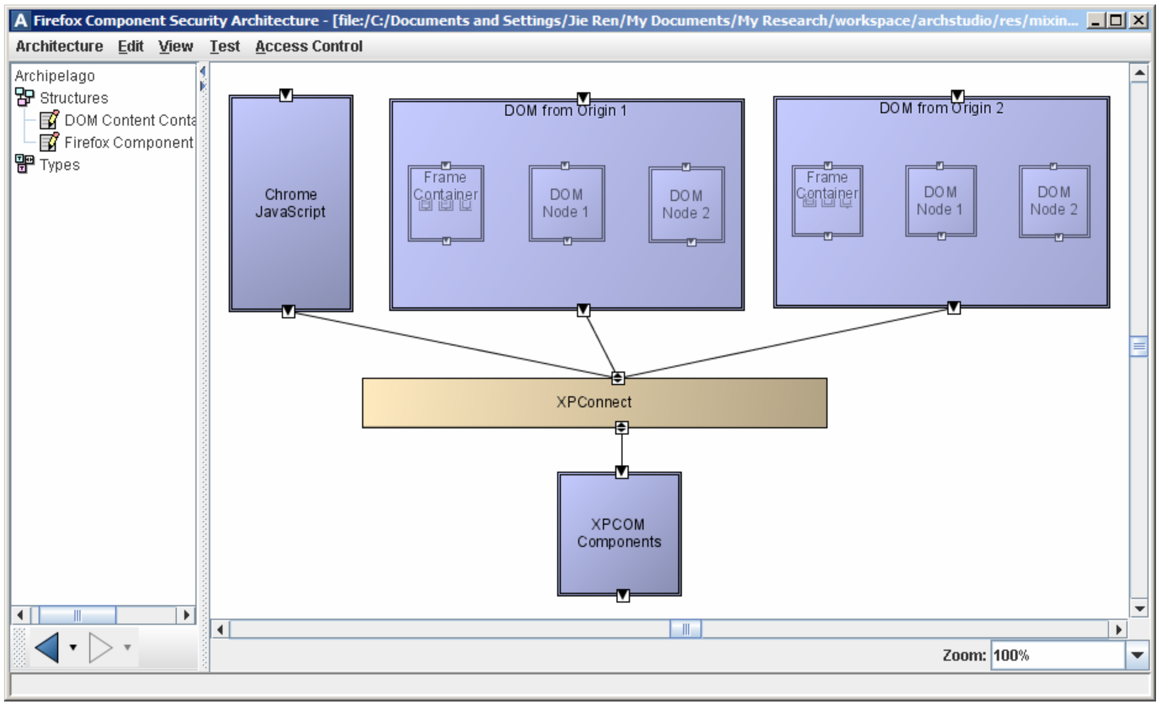


Figure 6-13, Firefox Component Security Architecture

The XPConnect connector, executing with the chrome role, connects various components and coordinates accesses to protected sensitive interfaces during normal operations. The XPConnect connector is a privilege retaining connector, which prohibits content components escalate their privileges to obtain the status of chrome components. XPConnect does not allow a connector between content components coming from different origins, thus obeying the same-origin policy. Our architectural access control check algorithm from Section 3.5 can show that ideally the XPConnect connector does not result in a privilege escalation exploit, because content DOM node cannot access the protected interfaces of XPCOM Components, but there exist paths from content code to the native XPCOM components so the XPConnect connector should carefully monitor the secure execution of content components [126].

XPConnect is a strategic place to improve the security for the overall architecture that contains both the browser proper and the extensions. Since Firefox serves as a platform on which many applications have been developed, it is insufficient that the browser itself is secure. Any installed extension, which also runs as chrome code, should also be secure. These extensions can use JavaScript to access DOM supplied by untrustworthy content pages. If any of these extensions is not scrupulous, then there could a vulnerability of privilege escalation.

The XPConnect connector in Firefox 1.5 (released in November 2005) provides a new regulation feature to improve the security of extensions [46]. The extension can ask XPConnect for a `XPCNativeWrapper` that wraps a DOM node supplied by content pages. The wrapper assures that the property access and function access on standard DOM interfaces through these wrappers will go to the standard implementations of these properties and functions supplied by

Firefox, and save the extensions from being tricked into using overridden versions of the properties and functions supplied by content. The `XPCNativeWrapper` also assures any properties returned form the wrapper is also an `XPCNativeWrapper`, thus the developer can naturally use the properties of DOM nodes without dangers of violating security.

This regulation feature has several advantages for security development: it has fixed several existing potential vulnerabilities, simplifies development for both Firefox and extensions so abiding code can benefit from the improved connector security automatically, and also prevents those extensions that do not safely manipulate the untrustworthy content from being exploited [151]. This improved security demonstrates that having a secure connector to coordinate secure collaborations and improving the connector could have positive impacts on the overall security architecture of a complex and componentized software system.

6.3.11 Discussions

The security manager of Firefox has substantially evolved since its inception. By the time of the Mozilla 1.0 release the earliest code of the security manager had almost completely been replaced. The content of the security manager has changed, so is the way to use it by other components of the browser. Along the evolution, the security manager has become the central place to make security related decisions, and more types of security check have been implemented within it. This requires the security manager to become more independent of contexts and the invoker to supply sufficient security context information.

We have modeled the component security architecture of the Firefox browser using our connector-centric approach for architectural access control. Our illustration shows that Firefox has a well-designed security architecture, and our approach can model this architecture of such a complex software system.

During our modeling we have observed some issues that are worth discussions.

Choice of Programming Language. The language used to develop Firefox browser and extensions, JavaScript, is also the programming language used by HTML pages to provide dynamic interactivity for the otherwise static contents. Given that the *same language* is used by both the browser authors (who are trusted) and webpage authors (who are not generally trusted by the browser), and the connection capability to XPCOM-built components enabled by XPConnect, it is critical to implement proper access control for downloaded JavaScript code.

The JavaScript language has some special features that can be abused by exploits. For example, the language allows the setter and getter functions of a property of an object to be *redefined*. This feature is intended to support redefinition of methods in objects that inherit other objects. However, if the redefinition is done by the content code on properties of objects originally provided by chrome code, such as those properties of the wrapped native XPCOM components, then other unsuspicious chrome code, which intends to invoke the original definitions, would accidentally use the content supplied code and give these content code undue privileges. The language even allows overriding the `eval` function, a function that executes any supplied string as JavaScript. These overrides have caused the critical vulnerability discussed in [44]. Defining and redefining interfaces on protected resources are security sensitive architectural operations. The fixes for this vulnerability in the XPConnect connector forbids such dangerous architectural operations by content code and finds the right principals to execute content code.

JavaScript allows content code to specify *timeout* functions, where a function is executed after a time delay. There is a possibility that the original context used to specify the timeout function no longer exists when the timeout occurs and thus the timeout function needs to be executed within a different context. If the new context is a chrome context and it blindly executes the timeout function that is originally supplied by a content context, then an undue escalation of privilege incurs. Thus the timeout function must be associated with the principal from which the function comes from, and only executes as such principals even in a different context. Similarly, event handlers should also be supplied with principals.

The Firefox JavaScript implementation supports *pre-compilation* of JavaScript programs. When an event handler is loaded, the JavaScript implementation could compile it with a principal associated with the handler function. When the event handler is later executed, the security manage should use the executing principal, instead of the precompiled principal, in deciding whether an access should be allowed. Otherwise, an undue access could be permitted.

Security Manager's Dependence on JavaScript. The current security manager is deeply dependent on the JavaScript language definition and its implementation in Firefox. There has been ongoing development work to use the Python scripting language to develop Firefox applications. Since the underlying XPCOM platform is independent of scripting languages, it is possible to factor out the security manager so that it can serve security needs of multiple scripting languages.

Protocol Handlers. Currently the security manager has directly made many decisions based on protocols when deciding whether a URI should be loaded or not. Another possibility regarding loading URI is to let the security manager inquire each individual protocol handler so that the security manager can become protocol independent, and the different policy choices can be centralized for ease of management.

Another issue with protocol handlers is that Firefox treats javascript as a pseudo protocol and a JavaScript snippet can be supplied where a URI is needed. While such treatment has long been an industry standard, from a security viewpoint a JavaScript URI is conceptually different than a normal http URI. The former results in code execution, and the execution context is determined by the loader of the URI. The latter mostly loads a passive resource, possibly will not execute any code, and the security of the http URI can be decided by simply inspecting its origin. A more prudent inspection of the javascript URI is possibly necessary.

Principals. Firefox uses principal as the basis for maintaining secure executions of JavaScript. A JavaScript script, either coming as a javascript pseudo URI, or as the script element of an HTML document, is associated with the principal of the containing document. However, when the script is executed, through loading the URI, executing a timeout function, or even retrieving a cached copy, the executor might be of a different principal, and blindly executing the script in this new principal could result in security breach. It is vital to track the principals of the JavaScript components, ideally through all traveling paths of these components within the browser. Such principal closure tracking feature was implemented in the classic Netscape browser, but it is missing in the current Firefox browser [35].

6.4 DCOM

DCOM, the Distributed Component Object Model, was the prominent object middleware for Microsoft during the 1990s. It keeps playing an important role in current Windows operating systems. In this section we show how our approach can model the security architecture of this middleware technology. This modeling demonstrates that our approach can be applied to a networked software environment.

6.4.1 DCOM Architecture

DCOM was developed to extend the object-oriented programming model of the Component Object Model (COM) to a distributed environment. The three core concepts of classic COM are [34]: *interface*, *class*, and *component*. An *interface* is a set of functions. Each interface is designated by a Global Unique Identifier (GUID). An interface is immutable after its publication. A *class* implements a set of interfaces. It is also designated by a Global Unique Identifier. A *component* is an instance of a class. When a client needs services from a component, it gets a reference to the component, in the form of an interface pointer, through either instantiating a new instance or getting a reference for a running instance. The client can inquire a component whether it supports a specific interface. COM uses a *source interface* as a reverse communication channel from components to clients. A source interface is declared in the component, but is implemented by the client. When a component processes a call from a client, it can invoke functions in the source interface and *call back* to the client.

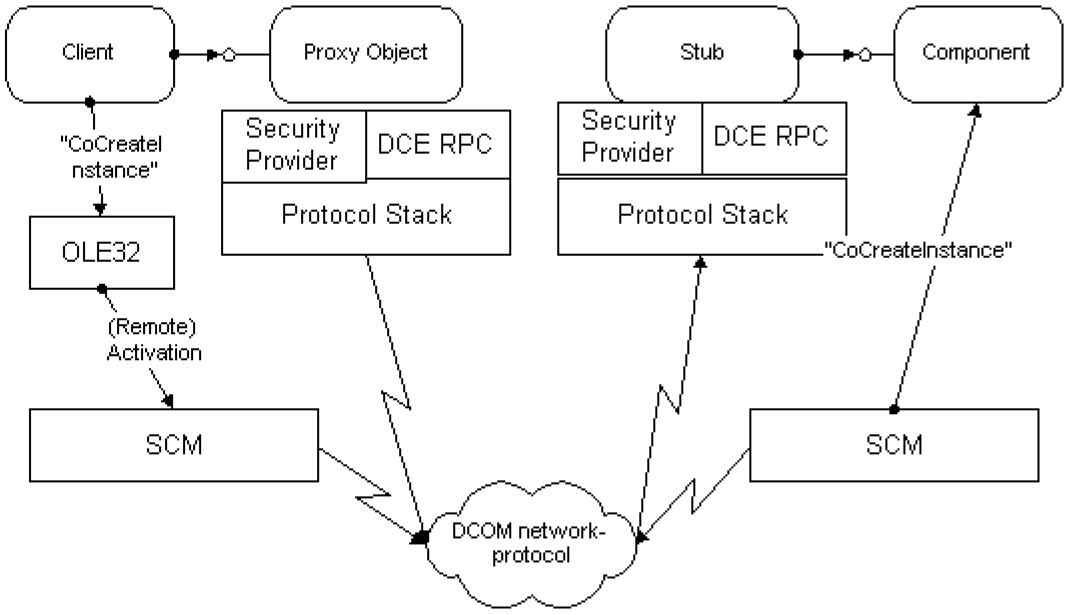


Figure 6-14, DCOM Architecture

Figure 6-14 depicts the underlying architecture for this distributed programming model [63]. The client first tries to get a reference for a component on the remote server side. Part of the DCOM infrastructure on the client machine, the service control manager (SCM), contacts its counterpart on the server side. The server SCM checks whether a satisfactory component is already running. If not, the server SCM **launches** a requested component. Otherwise, the server SCM just creates a reference for the running component and returns the reference to the client (**activates** the component for the client). The reference actually consists of a pair of a stub and a proxy. The stub is on the server side, and the proxy is on the client side. The stub and proxy handle communication details, such as marling and unmarshaling. Once the client gets the reference back, it can use the reference to **access** the functionalities provided by the component.

The protocol utilized by DCOM is called Object Remote Procedure Call (ORPC). It is an object-oriented extension to the Distributed Computing Environment Remote Procedure Call (DCE RPC) protocol. DCE RPC can run on top of different types of network protocol stacks. Various protocols have different security implications.

Another constituent of the DCOM architecture is the security provider. DCOM, like many other types of Windows components, supports a provider interface so different types of security providers can be used to supply various levels of security for DCOM clients and components.

6.4.2 Anonymous, Local, Remote, Activate, Launch, and Access

The security services provided by DCOM for clients and components consist of authentication and authorization. There are different levels of authentication, such as no authentication at all, authentication when a connection begins, and authentication for each method invocation. Both the client and the server can specify the intended authentication level, and DCOM assures that the higher authentication level is used, thus a client and a server can be assured that they get the authentication level they need. This is a case where the DCOM connector performs the security coordination between two architectural components.

The authentication information is used to determine whether an architectural operation request should be granted. As discussed in the last subsection, a client would ask for permissions on launch, activation, and access. Generally, at run-time a component has the final decision on whether a request for an architectural operation from the client should be granted. However, several other authorities are also involved in the decision making process: the computer can have a computer wide policy for components that do not explicitly specify their security requirements, and a component can specify its requirements statically through registry settings. Thee parties can be viewed as the different contexts discussed in Section 3.4: component registry settings as the type context (Section 3.4.2) and machine wide settings as the sub-architecture/container context (Section 3.4.3).

The decision on launch and activation cannot be programmatically determined by the component, since launching and activating components are executed before a component can even get the chance to make any decisions at run-time. Thus, a component can specify its safeguards on launch and activation statically through configuring Windows registry, and DCOM uses the settings to guard launch and activation. Properly enforcing launching and activating permissions is critical in defending against denial-of-service attacks.

One special type of authentication is *anonymou*s, where a client does not reveal its identity to the server. It is desirable when a client wants to preserve its privacy, and it is also necessary sometimes if the component needs to call back to the originating client through a source interface.

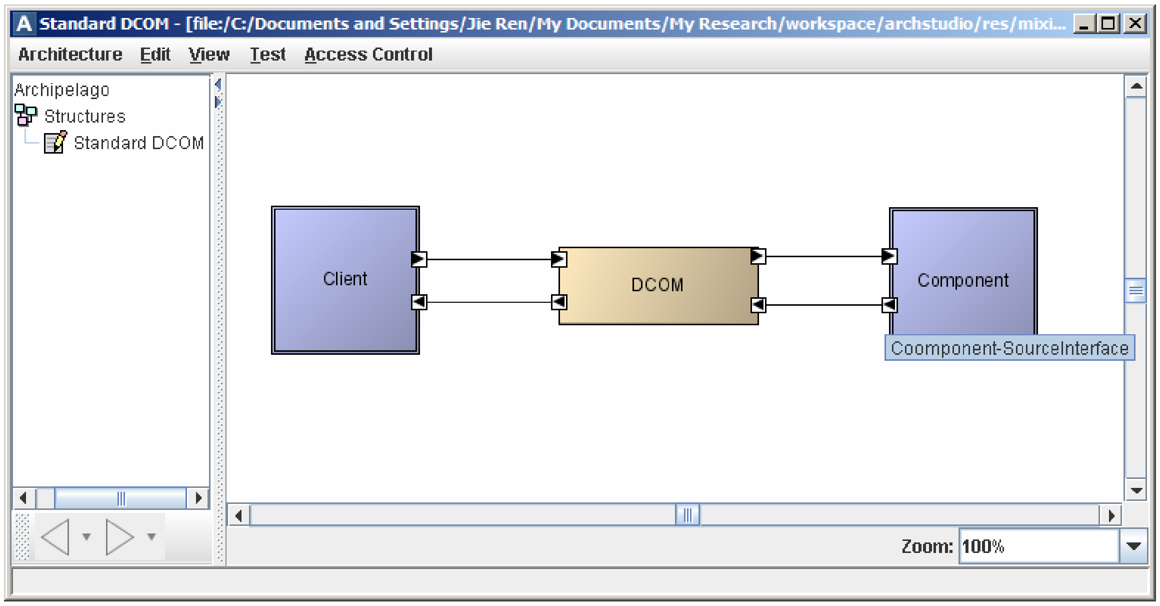

Standard DCOM - [file:/C:/Documents and Settings/Jie Ren/My Documents/My Research/workspace/archstudio/res/mixi...
Architecture
Edit
View
Test
Access Control
Archipelago
Structures
Standard DCOM
Client
DCOM
Component
Coomponent-SourceInterface
Zoom:
100%

Figure 6-15, DCOM Authentication and Authorization

The standard DCOM connector that performs authentication and authorization between a client and a component, possibly also supports callbacks, is depicted in Figure 6-15. The top links depict the normal invocation, and the bottom links show the call backs from the component to the client.

This architecture is not without its problems, though. The MSBlast worm [96] exploited a buffer overrun vulnerability in DCOM through anonymous authentication and has caused devastating damages. To reduce such risks, Microsoft has made several architectural security improvements for DCOM in Service Pack 2 for Widows XP and Service Pack 1 for Windows Server 2003 [97].

Previously DCOM does not differentiate client requests coming from different sources. The new architecture separates the local activation requests from the remote ones. The former comes through Local Remote Procedure Call (LRPC), a local communication facility used by the Windows kernel. The latter comes through regular Object RPC (ORPC). Separating the local requests from remote requests in DCOM can be viewed as introducing two different types of DCOM connectors, as depicted in Figure 6-16. These connectors enable supporting different security polices based on the origin of requests.

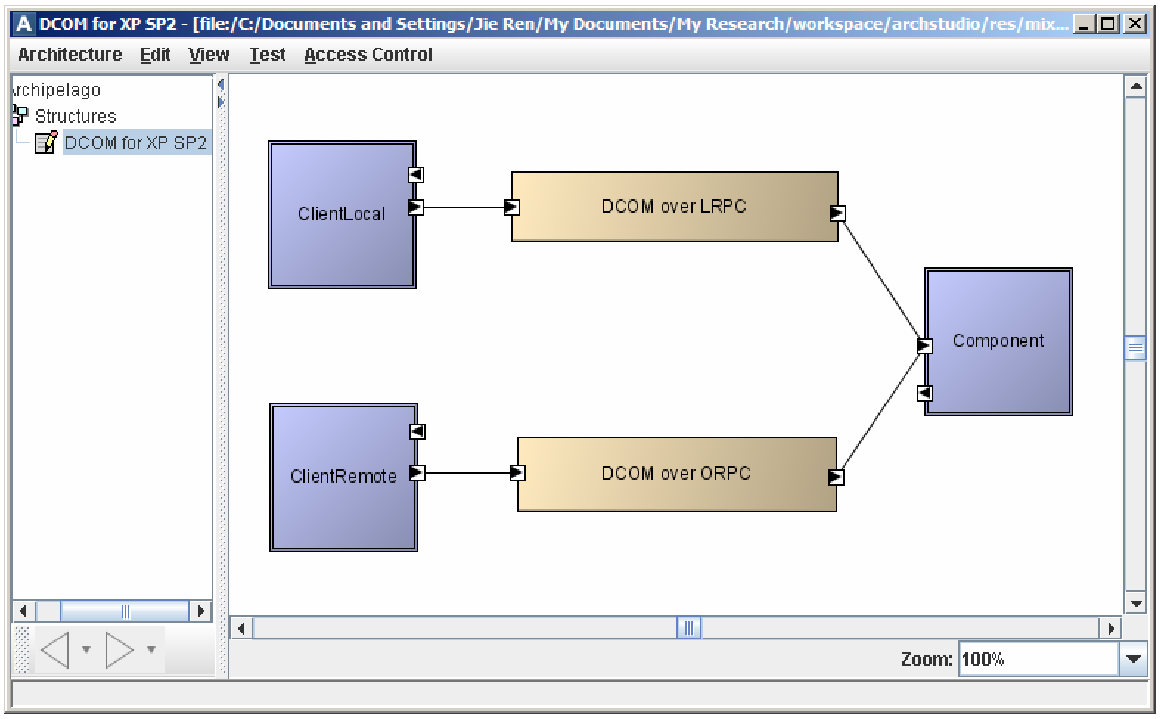


Figure 6-16, DCOM for XP SP2

Launch and activation through anonymous authentication from the DCOM over ORPC connector is disabled by default in the new DCOM. Unless administrators specifically permit anonymous authentication to support functionalities like callbacks, a future worm cannot exploit this capability as it would be able to.

The last improvement introduced in the new DCOM is about architectural concept. Previously permissions on activation and access are combined together. The new DCOM separates the activation permission from the access permission, and groups the activation permission together with the launch permission. The rationale is that both launch and activation involve acquiring an interface pointer and thus belong together logically. Viewing from our approach, both launch and activation are the architectural instantiation operations discussed in Section 4.4.1.

6.4.3 Impersonation and Delegation

When a component executes on behalf of a client, it can use the credentials of the client to perform necessary actions. When the authentication level negotiated between the client and the component through DCOM is *impersonation*, the component can use the credentials to access resources on the same machine as the component. When the authentication level is *delegation*, then the component can use the credentials to access resources on other computers through another DCOM interaction. In this delegation scenario, the intermediate component acts as the client in the second DCOM interaction, as depicted in Figure 6-17.

The client decides whether it trusts the intermediate to act as a delegate for it. Essentially, it grants its role to the intermediate component, so the last component can treat the intermediate as the original client. This can be viewed as a role-based trust management policy (Section 4.2). The DCOM connectors essentially propagate the privileges of the client to the last component through the intermediate component. This propagation occurs when the client allows delegation, the intermediate cloaks its own identity, and the underlying DCOM network protocol provides sufficient support.

6.4.4 DCOM and Internet

DCOM was designed in the early 1990s. As the networked software environment has greatly evolved ever since, some original decisions of DCOM has made several architectural adaptations necessary.

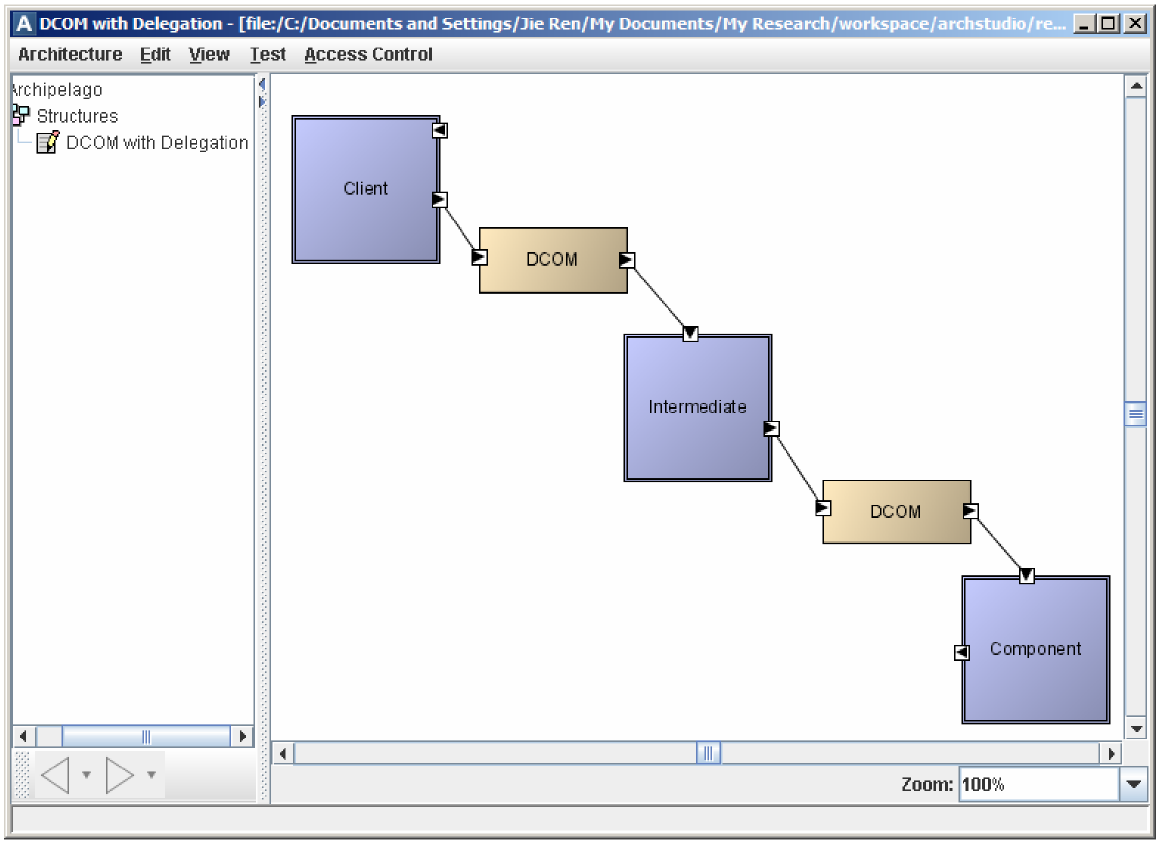

DCOM with Delegation - [file:/C:/Documents and Settings/Jie Ren/My Documents/My Research/workspace/archstudio/re...
Architecture Edit View Test Access Control
Archipelago
Structures
DCOM with Delegation
Client
DCOM
Intermediate
DCOM
Component
Zoom: 100%

Figure 6-17, DCOM with Delegation

DCOM is based on RPC. Like RPC, DCOM uses its own naming service. The naming service listens on one port, and dynamically allocates new ports for DCOM client/server interactions. This makes it harder to deploy DCOM through firewalls by opening a fixed set of ports, even though DCOM can be configured to use a limited range of ports.

To mitigate this problem, DCOM can be tunneled through HTTP, a protocol that is mostly permitted by firewalls. This feature is named COM Internet Services. The standard DCOM operates above TCP directly. Tunneling DCOM traffic through HTTP allows it to travel through firewalls. Tunneling is a useful technology to construct more complex connectors, where one connector, DCOM, is embedded inside another connector, HTTP.

Tunneling DCOM actually uses RPC over HTTP [98], which tunnels regular RPC traffic over HTTP to pass through firewalls. RPC over HTTP is necessary because many Windows kernel services run as RPC services and these services need to pass through firewalls. RPC over HTTP itself has also evolved to improve security. The later version of RPC over HTTP supports SSL encryption and mutual authentication, and disallows anonymous authentication, making the connector more complex yet more secure.

Architecturally, the DCOM over HTTP can be individually disabled, and the RPC over HTTP can also be turned off completely. In situations where these services are not needed, disabling them can reduce attack surfaces exploitable by malicious traffic.

The difficulty of DCOM with firewalls originates from the fact that DCOM was originally designed as a middleware platform for a corporate environment, which would be generally within a firewall. Because of this origin, some security practioners argue to disable DCOM for home users, where the features provided by DCOM are not generally used.

7 Conclusion

7.1 Summary

In this research, we have explored how to describe and enforce access control at the software architecture level. We combine core concepts from both software architecture and security into an integrated and coherent modeling mechanism, develop an algorithm to check the validity of access control, and implement an associated set of tools to design, analyze, and execute software modeled with these concepts.

The central question that this research has been trying to answer is **the software architectural access control question**: ***how can we describe and check access control issues at the software architecture level?*** Because the most basic elements of an access control scheme are subjects, objects, and actions and the most basic elements of an architecture description language are components and connectors, we combine them to find an answer to this question.

We identify what objects and actions are to be protected. In an architecture environment, the objects to be protected are the architectural constituents, i.e., the components and connectors themselves. In a finer granularity, the interfaces of components and connectors are the **resources** that are protected by **safeguards**.

We associate components and connectors with **subjects** that these components and connectors act for. These subjects represent the executing users, and ultimately provide the **privileges** for accessing protected architectural resources.

We have investigated several relationships involved in a software architecture model and how these relationships affect access control decisions. We call these relationships architectural **contexts**. A software component or connector exists within the following contexts: the nearby architectural constituents, the type of the constituent, the containing sub-architecture of the constituent, and the global architecture. All these relationships can affect an access control decision by supplying privileges and safeguards.

We have extended our base architecture description language, xADL, to describe these concepts. We have also extended the eXtensible Access Control Markup Language, XACML, to describe architectural access control **policies**. These two extensible languages are combined into a secure software architecture description language, Secure xADL. We have adopted these languages because they are extensible, provide excellent tool support, and are flexible enough to be adapted to suite our needs.

Based on these modeling concepts, we have developed an algorithm to check whether an accessing interface possesses sufficient privileges to access another protected interface on an architectural constituent. The algorithm can be applied to interfaces not directly connected together, and to interfaces that do not even reside within the same level of an architecture structure.

This is the most basic architectural control scheme. It establishes using constituents' subjects to obtain access on these constituents within architectural contexts. We have added more expressive extensions to this basic scheme.

Firstly, we have explored how the role-based access control model can be utilized to handle larger scale access control. Roles essentially provide an extra level of indirection between actions and subjects. Subjects can take different roles, in the form of **principals**. Each role gives the subject additional capabilities for performing actions.

Secondly, we have employed the trust management model to deal with the heterogeneous nature of decentralized software systems. In a modern software system, the components frequently operate in heterogeneous administrative domains and do not always fully trust each other. Trust management systems are introduced to authorize different users in a distributed environment, and to model the interoperation and delegation relationship among heterogeneous components. We have explored how each component can express its preference of protection, and how it will trust foreign components as components performing roles locally defined in its own domain.

Thirdly, we have extended interface-based access control into content-based access control. When protected resources cannot be completely described by interfaces, such as when sensitive messages are routed among generic interfaces of connectors, we support inspecting the content passing through the interfaces to accommodate finer access control.

Finally, we support executing run-time actions that are meaningful at the architectural level. The two basic operations are instantiating components and connectors and connecting components through connectors. Controlling instantiation of the architectural constituents restricts what components and connectors can be created, an essential architecture problem that has been largely neglected by previous software security research. Regulating connection of architectural constituents determines what connectors can be deployed and what components can connect to them. For event-based architecture styles, we have also identified external message routing and internal message routing as important architectural operations.

Our approach to architectural access control is centered around connectors. We believe connectors not only provide the essential glue to form an architectural topology, but are also vital to answering the architectural access control question: they propagate credentials for decision making, participate in determining validity of architectural connections, and can route messages in accordance with established policies.

We have extended our base architecture-based development environment, ArchStudio, to support modeling and analyzing the access control issues of software architectures. We have extended its editors to edit architecture policies textually and graphically, and have developed analysis tools to check the validity of a software architecture modeled using Secure xADL based on the formerly mentioned algorithm.

We have also developed a framework that allows constructing software architecture securely based on the C2 message passing style. This framework provides new security capabilities to the existing c2.fw framework. A designer can specify how security sensitive messages should be routed between participants that are not fully mutually trustful. The secure connector ensures secure delivering requests and notifications to proper receivers, subject to proper trust verification and content inspection.

Our research hypotheses are that:

Hypothesis 1: An architectural connector may serve as a suitable construct to model architectural access control.

Hypothesis 2: The connector-centric approach can be applied to different types of componentized and networked software systems.

Hypothesis 3: With a Secure xADL description, the access control check algorithm can check the suitability of accessing interfaces.

Hypothesis 4: In an architecture style based on event routing connectors, our approach can route events in accordance with the secure delivery requirements.

To validate hypothesis 3, we have analyzed the algorithm by mapping it to a well known graph reachability problem. A Secure xADL architecture description is transformed into a graph, where the nodes stand for privileges and safeguards, and the edges represent connections permitted by policies of connectors. We have established that permitting an architectural access between a pair of interfaces roughly equals to finding a path in the constructed graph, and thus any standard solution to the reachability problem can be used.

To validate hypotheses 1, 2, and 4, we have applied our modeling concepts and support tools to conduct four significant case studies. With the first case study, development of a secure coalition application based on the c2.fw.secure framework, we have demonstrated how our modeling mechanism and support tools can be used to fully support describing and executing architectural access control polices for instantiation, connection, external message routing, and internal message routing.

In our second case study Impromptu, a secure file sharing application, we have demonstrated how a networked software system can be built from third party software components using our connector-centric approach. We have also illustrated how a composite secure connector can be constructed from more primitive connectors.

In our third case study, analyzing Firefox, we have demonstrated how a major open source browser's security model, which includes external components installed or downloaded from other sources, can be modeled with our technique. We have shown how the installed/downloaded components can perform their tasks within the control of the containing browser and not interfere with each other.

In our last case study, analyzing DCOM, we have demonstrated how middleware can be modeled through our modeling technique. We illustrate the delegation feature of the DCOM subsystem and discuss how proper launch and activation are introduced to mitigate previous vulnerabilities.

In summary, we have proposed a connector-centric approach that helps answer the architectural access control question, and developed support tools and an execution framework for software architects to adopt the approach. We believe our approach can advance the understanding of both software architecture and access control technologies, and will empower software developers to develop more secure software for complex, networked, componentized systems.

7.2 Future work

We plan to further investigate our connector-centric approach in the following areas.

7.2.1 Different Types of Connectors

In this research, our study has been mostly focused on procedure call-based connectors and message passing connectors. There are other types of connectors proposed in software architecture literature [95] such as blackboard and database connectors. We will investigate how security is handled in these types of connectors, and how these connectors can be architected to further improve security.

7.2.2 Different Mechanisms to Construct Connectors

Connectors can be composed together to make more complex types of connectors. We have proposed one method to conjuncture a sequence of constituent connectors to obtain a composite connector. We plan to expand this investigation to study other mechanisms for composing secure connectors.

7.2.3 Security as an Aspect

Functional software architecture, expressed as components and connectors, forms a graph. The trust and access control relationships among them, expressed and enforced through connectors, can be viewed as an overlay network on top of this functional topology. We will investigate how this overlay network interacts with the base network.

Aspect-Oriented Programming [76] has been proposed as an effective means to address cross-cutting concerns. Access control can be viewed as one such concern. We plan to explore how access control can be expressed as a concern at the architecture level and how to apply AOP techniques to enhance our framework, especially how to construct composite connectors from primitive connectors.

7.2.4 Reflective Architectural Model

When a system maintains an architectural model of itself at run-time, it has a reflective architectural model. A reflective model enables a component and a connector to query, and possibly change its architectural context. A reflective architectural model gives components and connectors, at run-time, additional information to make security-related decisions based on different types of contexts. This enables flexibility in decision making and enforcement. In the meantime, it also creates security issues, since such a model must be properly protected to ensure that no unauthorized retrieval or modification of that model happens. We intend to incorporate a reflective architectural model and study how it facilitates security modeling and enforcement, and how it might enable applying the algorithm in Section 3.5 at run-time.

Reflective operations generally consume more resources than their non-reflective counterparts. Such potentially expensive operations should be monitored to counter denial-of-service attacks. These attacks, along with other types of attacks, can possibly be detected by a reflection-based intrusion detection system. The interaction between resource management and security is another future research area.

7.2.5 Dynamic Architecture

Components and connectors should have an interface that allow their policies to be changed at run-time, so they can execute new, possibly more secure policies, without a costly shutdown and reboot. Similarly, they can acquire new subjects or role principals, and thus enable for themselves more access. Supporting dynamism provides flexibility necessary for changing and evolving environments. We plan to extend our existing support for architecture dynamism to handle security dynamism as well.

7.2.6 Policy Conflict Resolution

Our algorithm currently gives only an answer of grant or denial. It does not identify whether there is any conflict in the policies retrieved from different contexts, and it does not provide assistance in identifying the source of a grant or denial decision. We plan to investigate work on policy conflict resolution [8] and policy change impact [38] to enhance the algorithm.

Bibliography

[1] Abadi, M. and Lamport, L., *Composing Specifications*, ACM Transactions on Programming Languages & Systems, **15**(1): pp. 73-132, 1993.

[2] Allen, R. and Garlan, D., *A Formal Basis for Architectural Connection*, ACM Transactions on Software Engineering and Methodology, **6**(3): pp. 213-249, 1997.

[3] Alpern, B. and Schneider, F.B., *Defining Liveness*, Information Processing Letters, **21**(4): pp. 181-5, 1985.

[4] Anderson, J.P., *Computer Security Technology Planning Study*,ESD-TR-73-51[NTIS AD-758 206], ESD/AFSC, Hanscom AFB: Bedford, MA, 1972.

[5] ANSI, *Role Based Access Control*,ANSI INCITS 359-2004, 2004.

[6] Bell, D.E. and LaPadula, L., *Secure Computer System: Unified Exposition and Multics Interpretation*,ESD-TR-75-306, ESD/AFSC, Hanscom AFB: Bedford, MA, 1975.

[7] Bellovin, S.M., *Security Problems in the Tcp/Ip Protocol Suite*, ACM SIGCOMM Computer Communication Review, **19**(2): pp. 32-48, 1989.

[8] Benferhat, S. and Rania. *A Stratification-Based Approach for Handling Conflicts in Access Control*, in Proceedings of the 8th ACM Symposium on Access Control Models and Technologies, pp.189-195, 2003.

[9] Berghel, H., *The Code Red Worm*, Communications of the ACM, **44**(12): pp. 15-19, 2001.

[10] Biba, K., *Integrity Considerations for Secure Computer Systems*,ESD-TR-76-372, ESD/AFSC, Hanscom AFB: Bedford, MA, 1977.

[11] Bidan, C. and Issarny, V. *A Configuration-Based Environment for Dealing with Multiple Security Policies in Open Distributed Systems*, in Proceedings of the 2nd European Research Seminar on Advances in Distributed Systems, pp.240-245, 1997.

[12] Bidan, C. and Issarny, V. *Security Benefits from Software Architecture*, in Proceedings of the 2nd International Conference on Coordination Languages and Models, pp.64-80, 1997.

[13] Bieber, P. *Security Function Interactions*, in Proceedings of the 12th IEEE Computer Security Foundations Workshop, pp.151-160, 1999.

[14] Bishop, M., *Computer Security: Art and Science*, Addison-Wesley, 2003.

[15] Blaze, M., Feigenbaum, J., and Keromytis, A.D. *Keynote: Trust Management for Public-Key Infrastructures*, in Proceedings of the 6th International Workshop on Security Protocols, pp.59-63, 1998.

[16] Blaze, M., Feigenbaum, J., and Lacy, J. *Decentralized Trust Management*, in Proceedings of the 1996 IEEE Symposium on Security and Privacy, pp.164-173, 1996.

[17] Blaze, M., Ioannidis, J., and Keromytis, A.D. *Experience with the Keynote Trust Management System: Applications and Future Directions*, in Proceedings of the 1st International Conference on Trust Management, pp.284-300, 2003.

[18] Bodoff, S., Armstrong, E., Ball, J., Carson, D., Evans, I., and Green, D., *The J2ee™ Tutorial*, 2nd ed., Addison-Wesley Professional, 2004.

[19] Bonatti, P. and Sabrina, *An Algebra for Composing Access Control Policies,* ACM Transactions on Information and System Security, **5**(1): pp. 1-35, 2002.
[20] Brewer, D.F.C. and Nash, M.J. *The Chinese Wall Security Policy*, in Proceedings of the 1989 IEEE Symposium on Security and Privacy, pp.206-214, 1989.
[21] Burrows, M., Abadi, A., and Needham, R., *A Logic of Authentication,* ACM Transactions on Computer Systems, **8**(1): pp. 18-36, 1990.
[22] Caplan, K. and Sanders, J.L., *Building an International Security Standard,* IT Professional, **1**(2): pp. 29-34, 1999.
[23] Clark, D.D. and Wilson, D.R. *A Comparison of Commercial and Military Computer Security Policies*, in Proceedings of the 1987 IEEE Symposium on Security and Privacy, pp.184-94, 1987.
[24] Clemm, G., Reschke, J., Sedlar, E., and Whitehead, J., *Web Distributed Authoring and Versioning (Webdav) Access Control Protocol,* RFC 3744, 2004.
[25] Coppersmith, D., *The Data Encryption Standard (Des) and Its Strength against Attacks,* IBM Journal of Research & Development, **38**(3): pp. 243-50, 1994.
[26] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C., *Introduction to Algorithms,* 2nd ed., MIT Press, 2001.
[27] Dashofy, E.M., Andr, Hoek, v.d., and Taylor, R.N., *A Comprehensive Approach for the Development of Modular Software Architecture Description Languages,* ACM Transactions on Software Engineering and Methodology, **14**(2): pp. 199-245, 2005.
[28] DeLine, R., *Avoiding Packaging Mismatch with Flexible Packaging,* IEEE Transactions on Software Engineering, **27**(2): pp. 124-143, 2001.
[29] Deng, Y., Wang, J., Tsai, J.J.P., and Beznosov, K., *An Approach for Modeling and Analysis of Security System Architectures,* IEEE Transactions on Knowledge and Data Engineering, **15**(5): pp. 1099-1119, 2003.
[30] Denning, D.E., *A Lattice Model of Secure Information Flow,* Communications of the ACM, **19**(5): pp. 236-43, 1976.
[31] DePaula, R., Ding, X., Dourish, P., Nies, K., Pillet, B., Redmiles, D., Ren, J., Rode, J., and Filho, R.S., *In the Eye of the Beholder: A Visualization-Based Approach to Information System Security,* International Journal of Human-Computer Studies, **64**(1-2): pp. 5-24, 2005.
[32] Dobson, J.E. and Randell, B. *Building Reliable Secure Computing Systems out of Unreliable Insecure Components*, in Proceedings of the 17th Annual Computer Security Applications Conference, pp.164-173, 2001.
[33] Ducasse, S. and Richner, T. *Executable Connectors: Towards Reusable Design Elements*, in Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, pp.483-499, 1997.
[34] Eddon, G.R. and Eddon, H., *Inside Com+ Base Services,* Microsoft Press, 1999.

[35] Eich, B., Fix All Non-Origin Url Load Processing to Track Origin Principals, 2005, https://bugzilla.mozilla.org/show_bug.cgi?id=293363#c10

[36] Feiertag, R., Redmond, T., and Rho, S. *A Framework for Building Composable Replaceable Security Services*, in Proceedings of the DARPA Information Survivability Conference & Exposition, pp.391-402 vol.2, 2000.

[37] Filho, R.S.S., Souza, C.R.B.d., and Redmiles, D.F. *The Design of a Configurable, Extensible and Dynamic Notification Service*, in Proceedings of the 2nd International Workshop on Distributed Event-based Systems, pp.1-8, 2003.

[38] Fisler, K., Krishnamurthi, S., Meyerovich, L.A., and Tschantz, M.C. *Verification and Change-Impact Analysis of Access-Control Policies*, in Proceedings of the 27th international conference on Software engineering, pp.196-205, 2005.

[39] Flanagan, D., *Javascript: The Definitive Guide*, 4th ed., O'Reilly, 2001.

[40] Focardi, R., *Analysis and Automatic Detection of Information Flows in Systems and Networks*, Thesis, University of Bologna, Italy, 1998.

[41] Focardi, R. and Gorrieri, R., *A Classification of Security Properties for Process Algebras*, Journal of Computer Security, **3**(1): pp. 5-33, 1994.

[42] Focardi, R. and Gorrieri, R., *The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties*, IEEE Transactions on Software Engineering, **23**(9): pp. 550-571, 1997.

[43] Focardi, R. and Gorrieri, R., *Classification of Security Properties: (Part I: Information Flow)*, in *Foundations of Security Analysis and Design : Tutorial Lectures*, Lecture Notes in Computer Science Vol. 2171, pp. 331-396, Springer-Verlag Heidelberg, 2001.

[44] Mozilla Foundation, Mozilla Bug 201132, https://bugzilla.mozilla.org/show_bug.cgi?id=201132

[45] Mozilla Foundation, Mozilla Foundation Security Advisory 2005-37: Code Execution through Javascript: Favicons, http://www.mozilla.org/security/announce/mfsa2005-37.html

[46] Mozilla Foundation, Safely Accessing Content Dom from Chrome, http://developer.mozilla.org/en/docs/Safely_accessing_content_DOM_from_chrome

[47] Ghosh, A.K. and McGraw, G. *An Approach for Certifying Security in Software Components*, in Proceedings of the 21st National Information Systems Security Conference, 1998.

[48] Gilham, F., Riemenschneider, R.A., and Stavridou, V. *Secure Interoperation of Secure Distributed Databases: An Architecture Verification Case Study*, in Proceedings of the 1999 Wold Congress on Formal Methods in the Development of Computing Systems, pp.701-717, 1999.

[49] Goguen, J.A. and Meseguer, J. *Security Policies and Security Models*, in Proceedings of the 1982 IEEE Symposium on Security and Privacy, pp.11-20, 1982.

[50] Gong, L., Ellison, G., and Dageforde, M., *Inside Java 2 Platform Security: Architecture, Api Design, and Implementation*, 2nd ed., Addison-Wesley, 2003.

[51] Hagstrom, A., Jajodia, S., Parisi-Presicce, F., and Wijesekera, D. *Revocations -a Classification*, in Proceedings of the 14th IEEE Computer Security Foundations Workshop, pp.44-58, 2001.

[52] Hallaraker, O. and Vigna, G. *Detecting Malicious Javascript Code in Mozilla*, in Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, pp.85-94, 2005.

[53] Halpern, J. and O'Neill, K. *Secrecy in Multiagent Systems*, in Proceedings of the 15th IEEE Computer Security Foundations Workshop, pp.32-46, 2002.

[54] Halpern, J.Y. and Weissman, V. *Using First-Order Logic to Reason About Policies*, in Proceedings of the 16th IEEE Computer Security Foundations Workshop, pp.187-201, 2003.

[55] Harrison, M.A., Ruzzo, W.L., and Ullman, J.D., *Protection in Operating Systems*, Communications of the ACM, **19**(8): pp. 461-471, 1976.

[56] Heckman, M.R. and Levitt, K.N. *Applying the Composition Principle to Verify a Hierarchy of Security Servers*, in Proceedings of the 31st Hawaii International Conference on System Sciences, pp.338-347 vol.3, 1998.

[57] Hemenway, J.A. and Fellows, J. *Applying the Abadi-Lamport Composition Theorem in Real-World Secure System Integration Environments*, in Proceedings of the 10th Annual Computer Security Applications Conference, pp.44-53, 1994.

[58] Hendrickson, S.A., Dashofy, E.M., and Taylor, R.N. *An Approach for Tracing and Understanding Asynchronous Architectures*, in Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003.

[59] Herrmann, P. *Information Flow Analysis of Component-Structured Applications*, in Proceedings of the 17th Annual Computer Security Applications Conference, pp.45-54, 2001.

[60] Herrmann, P. *Formal Security Policy Verification of Distributed Component-Structured Software*, in Proceedings of the 23rd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems, pp.257-272, 2003.

[61] Hinton, H.M. *Under-Specification, Composition and Emergent Properties*, in Proceedings of the 1997 New Security Paradigms Workshop, pp.83-93, 1997.

[62] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

[63] Horstmann, M. and Kirtland, M., Dcom Architecture, 1997, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomarch.asp

[64] Humenn, P., *The Formal Semantics of Xacml*, Syracuse University, 2003.

[65] Inverardi, P. and Tivoli, M., *Deadlock-Free Software Architectures for Com/Dcom Applications*, Journal of Systems and Software, **65**(3): pp. 173-183, 2003.

[66] Jaeger, T., Zhang, X., and Cacheda, F., *Policy Management Using Access Control Spaces,* ACM Transactions on Information and System Security **6**(3): pp. 327-364, 2003.
[67] Jim, T. *Sd3: A Trust Management System with Certified Evaluation*, in Proceedings of the 2001 IEEE Symposium on Security and Privacy, pp.106-115, 2001.
[68] Johnson, D.M. and Thayer, F.J. *Security and the Composition of Machines*, in Proceedings of the 1st IEEE Computer Security Foundations Workshop, pp.72-89, 1988.
[69] Joshi, J.B.D., Bertino, E., and Ghafoor, A., *An Analysis of Expressiveness and Design Issues for the Generalized Temporal Role-Based Access Control Model*, IEEE Transactions on Dependable and Secure Computing, **2**(2): pp. 157-175, 2005.
[70] Jürjens, J. *Umlsec: Extending Uml for Secure Systems Development*, in Proceedings of the 5th International Conference on The Unified Modeling Language, pp.412-425, 2002.
[71] Katara, M. and Katz, S. *Architectural Views of Aspects*, in Proceedings of the 2nd international conference on Aspect-oriented software development, pp.1-10, 2003.
[72] Khan, K., Han, J., and Zheng, Y. *A Framework for an Active Interface to Characterise Compositional Security Contracts of Software Components*, in Proceedings of the 2001 Australian Software Engineering Conference, pp.117-126, 2001.
[73] Khan, K.M. and Han, J., *Composing Security-Aware Software*, IEEE Software, **19**(1): pp. 34-41, 2002.
[74] Khan, K.M. and Han, J. *A Security Characterisation Framework for Trustworthy Component Based Software Systems*, in Proceedings of the 27th Annual International Computer Software and Applications Conference, pp.164-169, 2003.
[75] Khan, K.M., Han, J., and Zheng, Y. *Security Characterisation of Software Components and Their Composition*, in Proceedings of the 36th International Conference on Technology of Object-Oriented Languages and Systems, pp.240-249, 2000.
[76] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., and Irwin, J. *Aspect-Oriented Programming*, in Proceedings of the 11th European Conference on Object-Oriented Programming, pp.220-42, 1997.
[77] Lamport, L., *The Temporal Logic of Actions*, ACM Transactions on Programming Languages and Systems, **16**(3): pp. 872-923, 1994.
[78] Lampson, B.W., *A Note on the Confinement Problem*, Communications of the ACM, **16**(10): pp. 613-15, 1973.
[79] Lampson, B.W., *Protection*, ACM SIGOPS Operating Systems Review, **8**(1): pp. 18-24, 1974.
[80] Li, N., Grosof, B.N., and Feigenbaum, J., *Delegation Logic: A Logic-Based Approach to Distributed Authorization*, ACM Transactions on Information and System Security, **6**(1): pp. 128-171, 2003.

[81] Li, N. and Mitchell, J.C. *Rt: A Role-Based Trust-Managemant Framework*, in Proceedings of the DARPA Information Survivability Conference & Exposition III, pp.201-212, 2003.

[82] Lodderstedt, T., Basin, D.A., J, and Doser, r. *Secureuml: A Uml-Based Modeling Language for Model-Driven Security*, in Proceedings of the 5th International Conference on The Unified Modeling Language, pp.426-441, 2002.

[83] Lopes, A., Wermelinger, M., and Fiadeiro, J.L., *Higher-Order Architectural Connectors*, ACM Transactions on Software Engineering and Methodology, **12**(1): pp. 64-104, 2003.

[84] Magee, J. and Kramer, J. *Dynamic Structure in Software Architectures*, in Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering, pp.3-14, 1996.

[85] Mantel, H. *On the Composition of Secure Systems*, in Proceedings of the 2002 IEEE Symposium on Security and Privacy, pp.81-94, 2002.

[86] Marks, D.G., Sell, P.J., and Thuraisingham, B.M., *Momt: A Multilevel Object Modeling Technique for Designing Secure Database Applications*, Journal of Object-Oriented Programming, **9**(4): pp. 22-9, 1996.

[87] McCullough, D. *Noninterference and the Composability of Security Properties*, in Proceedings of the 1988 IEEE Symposium on Security and Privacy, pp.177-186, 1988.

[88] McFarlane, N., *Rapid Application Development with Mozilla*, Prentice Hall PTR, 2003.

[89] McLean, J. *A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions*, in Proceedings of the 1994 IEEE Symposium on Security and Privacy, pp.79-93, 1994.

[90] McLean, J., *Security Models*, in *Encyclopedia of Software Engineering*, Vol., 1994.

[91] McLean, J., *A General Theory of Composition for a Class of "Possibilistic" Properties*, IEEE Transactions on Software Engineering, **22**(1): pp. 53-67, 1996.

[92] McLean, J. *Twenty Years of Formal Methods*, in Proceedings of the 1999 IEEE Symposium on Security and Privacy, pp.115-116, 1999.

[93] Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., and Robbins, J.E., *Modeling Software Architectures in the Unified Modeling Language*, ACM Transactions on Software Engineering and Methodology, **11**(1): pp. 2-57, 2002.

[94] Medvidovic, N. and Taylor, R.N., *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, **26**(1): pp. 70-93, 2000.

[95] Mehta, N.R., Medvidovic, N., and Phadke, S. *Towards a Taxonomy of Software Connectors*, in Proceedings of the 22nd International Conference on Software Engineering, pp.178-187, 2000.

[96] Microsoft, Buffer Overrun in Rpc Interface Could Allow Code Execution, http://www.microsoft.com/technet/security/Bulletin/MS03-026.mspx

[97] Microsoft, Dcom Security Enhancements in Windows Xp Service Pack 2 and Windows Server 2003 Service Pack 1,

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomarch.asp
[98] Microsoft, Using Http as an Rpc Transport, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/rpc/rpc/using_http_as_an_rpc_transport.asp
[99] Millen, J. *20 Years of Covert Channel Modeling and Analysis*, in Proceedings of the 1999 IEEE Symposium on Security and Privacy, pp.113-114, 1999.
[100] Milner, R., *Communication and Concurrency*, Prentice Hall, 1989.
[101] Minsky, N.H. *Should Architectural Principles Be Enforced?*, in Proceedings of the Computer Security, Dependability and Assurance: From Needs to Solutions, pp.89-102, 1998.
[102] Minsky, N.H. and Ungureanu, V. *Unified Support for Heterogeneous Security Policies in Distributed Systems*, in Proceedings of the 7th USENIX Security Symposium, pp.131-42, 1998.
[103] Moriconi, M., Qian, X., and Riemenschneider, R.A., *Correct Architecture Refinement*, IEEE Transactions on Software Engineering, **21**(4): pp. 356-372, 1995.
[104] Moriconi, M., Qian, X., Riemenschneider, R.A., and Gong, L. *Secure Software Architectures*, in Proceedings of the 1997 IEEE Symposium on Security and Privacy, pp.84-93, 1997.
[105] Myers, A.C. and Liskov, B., *Protecting Privacy Using the Decentralized Label Model*, ACM Transactions on Software Engineering & Methodology, **9**(4): pp. 410-42, 2000.
[106] OASIS, Extensible Access Control Markup Language (Xacml), http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf
[107] Olawsky, D., Payne, C., Sundquist, T., Apostal, D., and Fine, T. *Using Composition to Design Secure, Fault-Tolerant Systems*, in Proceedings of the 3rd IEEE International High-Assurance Systems Engineering Symposium, pp.29-32, 1998.
[108] Olawsky, D., Payne, C., Sundquist, T., Apostal, D., and Fine, T. *Using Composition to Design Secure, Fault-Tolerant Systems*, in Proceedings of the DARPA Information Survivability Conference & Exposition, pp.380-390 vol.2, 2000.
[109] Owre, S., Rushby, J.M., and Shankar, N. *Pvs: A Prototype Verification System*, in Proceedings of the 11th International Conference on Automated Deduction, pp.748-52, 1992.
[110] Payne, C.N., Jr. *Using Composition and Refinement to Support Security Architecture Trade-Off Analysis*, in Proceedings of the 22nd National Information Systems Security Conference, pp.238-44, 1999.
[111] Peri, R.V., Wulf, W.A., and Kienzle, D.M. *A Logic of Composition for Information Flow Predicates*, in Proceedings of the 9th IEEE Computer Security Foundations Workshop, pp.82-94, 1996.
[112] Ray, I., France, R., Li, N., and Georg, G., *An Aspect-Based Approach to Modeling Access Control Concerns*, Information and Software Technology, **46**(9): pp. 575-587, 2004.

[113] Ren, J., *Modular Security: Design and Analysis*,UCI-ISR-04-4, Institute for Software Research, University of California, Irvine, 2004.
[114] Ren, J., Taylor, R., Dourish, P., and Redmiles, D. *Towards an Architectural Treatment of Software Security: A Connector-Centric Approach*, in Proceedings of the Workshop on Software Engineering for Secure Systems, held in conjunction with the 27th International Conference on Software Engineering, 2005.
[115] Ren, J. and Taylor, R.N. *A Secure Software Architecture Description Language*, in Proceedings of the Workshop on Software Security Assurance Tools, Techniques, and Metrics, held in conjunction with the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005.
[116] Ribeiro, C., Zúquete, A., Ferreira, P., and Guedes, P. *Spl: An Acess Control Language for Security Policies with Complex Constraints*, in Proceedings of the 2001 Network and Distributed System Security Symposium, pp.89-107, 2001.
[117] Ryan, P., McLean, J., Millen, J., and Gligor, V. *Non-Interference, Who Needs It?*, in Proceedings of the 14th IEEE Computer Security Foundations Workshop, pp.237-238, 2001.
[118] Ryan, P.Y.A., *Mathematical Models of Computer Security*, in *Foundations of Security Analysis and Design : Tutorial Lectures*, Lecture Notes in Computer Science Vol. 2171, pp. 1-62, Springer-Verlag Heidelberg, 2001.
[119] Ryan, P.Y.A. and Schneider, S.A. *Process Algebra and Non-Interference*, in Proceedings of the 12th IEEE Computer Security Foundations Workshop, pp.214-227, 1999.
[120] Sabelfeld, A. and Myers, A.C., *Language-Based Information-Flow Security*, IEEE Journal on Selected Areas in Communications, **21**(1): pp. 5-19, 2003.
[121] Saltzer, J.H. and Schroeder, M.D., *The Protection of Information in Computer Systems*, Proceedings of the IEEE, **63**(9): pp. 1278-308, 1975.
[122] Samarati, P. and Vimercati, S.d.C.d., *Access Control: Policies, Models, and Mechanisms*, in *Foundations of Security Analysis and Design : Tutorial Lectures*, Lecture Notes in Computer Science Vol. 2171, pp. 137-196, Springer-Verlag Heidelberg, 2001.
[123] Sandhu, R. and Munawer, Q. *How to Do Discretionary Access Control Using Roles*, in Proceedings of the 3rd ACM Workshop on Role-based Access Control, pp.47-54, 1998.
[124] Sandhu, R.S., Coyne, E.J., Feinstein, H.L., and Youman, C.E., *Role-Based Access Control Models*, Computer, **29**(2): pp. 38-47, 1996.
[125] Santen, T., Heisel, M., and Pfitzmann, A. *Confidentiality-Preserving Refinement Is Compositional - Sometimes*, in Proceedings of the 7th European Symposium on Research in Computer Security, pp.194-211, 2002.
[126] Schmidt, J., Chrome-Plated Holes, 2005, http://www.heise.de/security/artikel/61652/0
[127] Schneider, F.B., *Enforceable Security Policies*, ACM Transactions on Information and System Security **3**(1): pp. 30-50, 2000.

[128] Schneider, F.B., Morrisett, G., and Harper, R., *A Language-Based Approach to Security*, in *Informatics. 10 Years Back, 10 Years Ahead*, Lecture Notes in Computer Science Vol. 2000, pp. 86-101, Springer-Verlag, 2001.
[129] Sewell, P. and Vitek, J. *Secure Composition of Untrusted Code: Wrappers and Causality Types*, in Proceedings of the 13th IEEE Computer Security Foundations Workshop, pp.269-284, 2000.
[130] Sini, R. and Kutvonen, L. *Trust Management Survey*, in Proceedings of the 3rd International Conference on Trust Management, pp.77-92, 2005.
[131] Spitznagel, B. and Garlan, D. *A Compositional Approach for Constructing Connectors*, in Proceedings of the 2nd Working IEEE/IFIP Conference on Software Architecture, pp.148-157, 2001.
[132] Spitznagel, B. and Garlan, D. *A Compositional Formalization of Connector Wrappers*, in Proceedings of the 25th International Conference on Software Engineering, pp.374-384, 2003.
[133] Stavridou, V., Dutertre, B., Riemenschneider, R.A., and Saidi, H. *Intrusion Tolerant Software Architectures*, in Proceedings of the DARPA Information Survivability Conference & Exposition II, pp.230-241 vol.2, 2001.
[134] Stavridou, V., Riemenschneider, R.A., and Gilham, F. *Sdtp: A Verified Architecture for Secure Distributed Transaction Processing*, in Proceedings of the DARPA Information Survivability Conference & Exposition, pp.369-379 vol.2, 2000.
[135] SunXACML, Sunxacml, http://sunxacml.sourceforge.net
[136] Sutherland, D. *A Model of Information*, in Proceedings of the 9th National Computer Security Conference, pp.175-183, 1986.
[137] Szyperski, C., *Component Software - Beyond Object-Oriented Programming*, 2nd ed., Addison-Wesley, 2002.
[138] Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., Jr., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D.L., *A Component- and Message-Based Architectural Style for Gui Software*, IEEE Transactions on Software Engineering, **22**(6): pp. 390-406, 1996.
[139] Tisato, F., Savigni, A., Cazzola, W., and Sosio, A. *Architectural Reflection. Realising Software Architectures Via Reflective Activities*, in Proceedings of the 2nd International Workshop on Engineering Distributed Objects, pp.102-15, 2000.
[140] Tripunitara, M.V. and Li, N. *Comparing the Expressive Power of Access Control Models*, in Proceedings of the 11th ACM conference on Computer and communications security, pp.62-71, 2004.
[141] W3C, Xml Path Language (Xpath) 2.0, http://www.w3.org/TR/xpath20/
[142] Wang, X., Yu, H., and Yin, Y.L. *Efficient Collision Search Attacks on Sha-0*, in Proceedings of the 25th Annual International Cryptology Conference Advances on Cryptology, pp.1-16, 2005.
[143] Weeks, S. *Understanding Trust Management Systems*, in Proceedings of the 2001 IEEE Symposium on Security and Privacy, pp.94-105, 2001.

[144] Wijesekera, D. and Jajodia, S., *A Propositional Policy Algebra for Access Control*, ACM Transactions on Information and System Security, **6**(2): pp. 286-325, 2003.

[145] Win, B.D., *Engineering Application-Level Security through Aspect-Oriented Software Development*, Thesis, Katholieke Universiteit Leuven, 2004.

[146] Wing, J.M., *A Call to Action: Look Beyond the Horizon*, IEEE Security & Privacy, **1**(6): pp. 62-67, 2003.

[147] Wittbold, J.T. and Johnson, D.M. *Information Flow in Nondeterministic Systems*, in Proceedings of the 1990 IEEE Symposium on Security and Privacy, pp.144-61, 1990.

[148] Zakinthinos, A., *On the Composition of Security Properties*, Thesis, University of Toronto: Toronto, Ontario, 1996.

[149] Zakinthinos, A. and Lee, E.S. *Composing Secure Systems That Have Emergent Properties*, in Proceedings of the 11th IEEE Computer Security Foundations Workshop, pp.117-122, 1998.

[150] Zaremski, A.M. and Wing, J.M., *Specification Matching of Software Components*, ACM Transactions on Software Engineering and Methodology, **6**(4): pp. 333-369, 1997.

[151] Zbarsky, B., Stop Sharing Dom Object Wrappers between Content and Chrome, 2005, https://bugzilla.mozilla.org/show_bug.cgi?id=281988#c27

[152] Zhang, L., Ahn, G.-J., and Chu, B.-T., *A Rule-Based Framework for Role-Based Delegation and Revocation*, ACM Transactions on Information and System Security, **6**(3): pp. 404-441, 2003.