A Secure Software Architecture Description Language

Jie Ren, Richard N. Taylor
Department of Informatics
University of California, Irvine
Irvine, CA 92697-3425
1-949-8242776
{jie, taylor }@ics.uci.edu

ABSTRACT

Security is becoming a more and more important concern for software architecture and software components. Previous modeling approaches provide insufficient support for an in-depth treatment of security. This paper argues for a more comprehensive treatment of an important security aspect, access control, at the architecture level. Our approach models security subject, resource, privilege, safeguard, and policy of architectural constituents. The modeling language, Secure xADL, is based on our existing modular and extensible architecture description language. Our modeling is centered around software connectors that provides a suitable vehicle to model, capture, and enforce access control. Combined with security contracts of components, connectors facilitate describing the security characteristics of software architecture, generating enabling infrastructure, and monitoring run-time conformance. This paper presents the design of the language and initial results of applying this approach. This research contributes to deeper and more comprehensive modeling of architectural security, and facilitates detecting architectural vulnerabilities and assuring correct access control at an early design stage.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Modules and interfaces.

General Terms

Design, Security, Access Control, Languages, Secure xADL

Keywords

Software architecture, secure software connector, security, architectural access control

1. INTRODUCTION

Consider the example of spam emails. With more and more proliferation of such emails (arguably now there is more spam traffic than normal traffic), effectively handling them is becoming a prominent security problem. Conceptually there are several measures that can be taken to mitigate the issue. The most radical route requires changing the email protocols, which was invented a quarter century ago for a friendly, trustworthy, and benign environment. Before such new protocols can be developed and widely deployed, the more realistic solution lies in "hardening" the existing facilities. If the administrators of the email servers take the responsibility, they can drop mails from known spammers, or delay mails from unknown senders, which will deter spammers that do not resend their spam. Such administrative changes, however, might have adversary effects on normal email operations, since they could possibly change the otherwise normal latency of emails. The users could also adopt their own countermeasures. If their incoming mail servers support spam control features, the users can configure the mail servers and let them either drop the spasm or filter them to special folders. Depending on how accurate the spam filters can be, completely dropping the spam might not be the best choice since the user will not be aware of the existence of possible misclassifications. If the users' email clients support spam filters, which is the case for almost all modern clients, then the users can adopt a client-only solution, relying on the client filters to be properly trained for filtering spam emails, and reviewing such emails for possible misclassifications. If the user adopts both a server-filtering solution and a client-filtering solution, then the user should be cautious about how these two mechanisms interoperate with each other, since the configuration results of one solution cannot be easily transferred to another solution. This spam filtering example illustrates how many components a modern security problem can touch and how challenging it might be for the different defensive mechanisms to cooperate and provide the desired functionalities securely.

With rapidly advancing hardware technologies and ubiquitous use of computerized applications, modern software is facing challenges that it has not seen before. More and more software is built from existing components. These components may come from different sources. This complicates analysis and composition, even if a dominant decomposition mechanism is available. Additionally more and more software is running in a networked environment. These 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. 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.

Take a popular web server, Microsoft Internet Information Server (IIS), as an example. The 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 [2]. A major architectural change was introduced in 2003 for its Version 6.0. This version is much safer than previous versions, due to these architectural changes [32]. No major security 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.

Component-based software engineering and software architecture provide the necessary higher-level perspective. 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.

Facing the new challenges of security for networked componentized software and given the base provided by existing software architecture research, we propose a software architecture description language that focuses on access control. The language enables a comprehensive treatment of security at the architecture level, striving for assurance on correct access control among architectural constituents.

Section 2 of this paper surveys related work. Section 3 outlines our approach, introducing the base architecture description language and the modeling extensions necessary for security development. Section 4 gives an example of applying the approach to a coalition application. Section 5 summarizes initial results of our research and outlines future work.

2. RELATED WORK

Since our work is focused on semantically rich secure connectors, this section first surveys existing research on connector-based software architectures. It also surveys security modeling based on other design notations, such as UML.

2.1. Architectural Connectors

Architecture Description Languages (ADLs) provide the foundation for architectural description and reasoning [18]. Most existing ADLs support descriptions of structural issues, such as components, connectors, and configurations. Several ADLs also support descriptions of behaviors [1, 17]. The description of behaviors is either centered around components, extending the standard “providing” and “requiring” interfaces, or is attached to connectors, if the language supports connectors as first class citizens [1]. These formalisms enable reasoning about behaviors, such as avoidance and detection of deadlock. Some early efforts have been invested on modeling and checking security-related behaviors, such as access control [21], encryption, and decryption [3].

Among the numerous ADLs proposed, some do not support connectors as first class citizens [6, 17]. Interactions between components are modeled through component specifications in these modeling formalisms. This choice is in accordance with 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 [5]. It also hinders capturing design rationales and reusing implementations of communication mechanisms, which is made possible by standalone connectors [7]. We believe a first class connector that explicitly captures communication mechanisms provides a necessary design abstraction.

Several efforts are focused on understanding and developing connectors in the context of ADLs. A taxonomy of connectors is proposed in [19], 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 [27] and to compose high-order connectors from existing connectors [16] are also proposed.

However, these efforts are not completely satisfactory. They suffer from the fact that they are general techniques. All of them aim at providing general constructs and techniques to suit a wide array of software systems, which leave them ignoring specific needs that arise from different application properties. For example, both the connector transformation technique [27] and the connector composition technique [16] have been applied to design secure applications, but the treatment of security does not address the more comprehensive security requirements as understood by security practitioners. Those requirements have richer semantics. These semantics raise challenges, because the general techniques must handle them in a semantically compatible way instead of just decomposing the challenges into semantically neutral “assembly languages.” These semantics also provide opportunities, because they supply new contexts and information that can be leveraged. Such extra constraints are especially beneficial to the application of formal techniques, because these additional conditions could reduce the possible state space and lower the decidability and computational cost.

It is our position that a deeper treatment of security in the connector technology is needed for a comprehensive solution to the important software security problem. Such a treatment should handle and leverage the richer semantics provided by specific security properties, such as various encryption, authentication, and authorization schemes, instead of equating these security features with opaque abstract functions.

2.2. UML-based Security Modeling

UML is a standard design modeling language. There have been several UML-based approaches for modeling security. UMLsec [11] and SecureUML [15] 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 [23] 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 [12] uses concern diagram as a vehicle to support general architectural aspects. It collects relevant UML modeling elements into UML package diagrams.

3. SECURE xADL

This section details the elements of the security modeling approach we are taking. We first give an overview of our existing architectural description language, and then we outline the new modeling capabilities we propose to help assuring correct architectural access control.

3.1. Overview of xADL

We extend our existing Architecture Description Language (ADL), xADL 2.0 [4], to support new modeling concepts that are necessary for architectural access control. xADL is an XML-based extensible ADL. It has a set of core features, and it supports modular extensions.

The core features 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 adopt these two concepts, and extend 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 most 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 signatures of the inner types. This 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. Modeling Architectural Security

xADL has provided an extensible foundation for modeling architectural concerns. We extend it to model software security, focusing on architectural access control. We adopt the same modular and extensible approach utilized by the base xADL language, starting from a set of core security concepts and enabling future extensions. These extensions will eventually be subject to the extent that is made possible by both theoretical expressiveness and practical applicability.

3.2.1. Access Control

Our approach supports multiple security models that are being widely used in practice. Our first efforts are directed at the classic access control models [13], which is the dominant security enforcement mechanism.

In the classic access control model [13], a system contains a set of subjects that has permissions and a set of objects (also called resources) on which these permissions can be exercised. An access matrix specifies what permission 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 permissions 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 permissions 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 permissions (capabilities) that the subject has over all objects.

Other models, such as the more recent role-based access control model [26] and the trust management model [33], can be viewed as extensions to this basic access control model. The role-based model introduces the concept of roles as an indirection to organize the permissions assignments to subjects. Instead of assigning permissions directly to subjects, the permissions are assigned to roles. Such roles can be organized into hierarchies, so a more senior role can possess additional permissions in addition to the permissions it inherits from a junior role. Each subject can selectively take multiple roles when executing software, thus acquiring the related permissions.

The trust management model provides a decentralized approach to manage subjects and delegate permissions. Since it is difficult to set up a centrally managed repository of subjects in a decentralized environment, trust management models use the attributes of subjects to identify them, and each local subject can check these attributes based on the information that is present at the local subject. Because the subjects are independent of each other, they can delegate permissions between them. Several efforts have been made to provide a more unified view of these models [25, 29]. For example, the role-based trust-management framework [14] views the trust management relationship as the containment relationship between independently defined roles. Such a unified view provides the theoretical foundation for our architectural treatment of access control models.

3.2.2. Subject, Resource, Privilege, Safeguard, Policy

Inspired by such a unified view, we introduce the following core concepts that are necessary to model access control at the architecture level: ***subject***, ***principal, resource, privilege, safeguard,*** and ***policy***. We extend the base xADL language with these concepts to get a new language, Secure xADL. To the best of our knowledge, this is the first effort to model these security concepts directly in an architectural description language.

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 will not change during runtime, either advertently 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.

A subject can take multiple ***principals***. Essentially, principals encapsulate the credentials a subject possess to acquire permissions. In the classic access control model, the principal is synonymous with subject, directly designating the identity of the subject. In the role-based access control model, a principal can be a role that the subject takes. And since a subject can assume multiple roles, it can possess several principals. In the trust management model, a principal can be the public key credentials that a subject possesses. Principals provide indirection and abstraction necessary for more advanced access control models.

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 powers to improve assurance.

Permissions describes a possible operation on an object. Another important security feature that is missing from traditional ADLs is ***privilege***, which describe what permissions a component possess depending on the executing subjects. Current modeling approaches take a maximum privilege route, where a component's interfaces list all privileges that a component possibly needs. This is 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 avoid 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 handles active resources. These privileges include architecturally important privileges, such as instantiation and destruction of architectural constituents, connection of components with connectors, execution, and reading and writing of architecturally critical information. Little attention has been paid to these privileges, and the limited treatment neglects the creation and destruction of software components and connectors [31].

A corresponding notion is ***safeguard***, which are 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.

A ***policy*** ties all above mentioned concepts together. It specifies what privileges a subject should have to access resources protected by safeguards. It is the foundation for making access control decisions. There have been numerous studies on security policies [8, 20, 30]. Since our focus is on a more practical and extensible modeling of software security at the architectural 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.

Towards this goal, we feel the eXtensible Access Control Markup Language (XACML) [22] can serve as the basis for our architectural security policy modeling. The language is based on XML, which makes it a natural fit for our own XML-based ADL. The language is extensible. Currently it has a core that specifies the classic access control model, and a profile for role-based access control. A profile for trust management is also in development. This modular approach makes the language evolvable, just like our own xADL modular approach. The extensibility allows us to adopt it without loss of future expressiveness. Finally, the language has been equipped with a formal semantics [9]. While this semantics is an add-on artifact of the language, it does illustrate 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.

3.2.3. Contexts of Architectural Access Control

In traditional access control, context has been used to designate factors involved in decision making that are not part of the subject-operation-object tuple. The most prominent example is time, which has been extensively used to express temporal access control constraints [10].

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. We use *context* to designate those relationships involved in architectural access control. More specifically, the context can include 1) the nearby components and connectors of the component and the connector, 2) the explicitly modeled sub-architecture that contains the component and the connector, 3) the type of the component and the connector, and 4) the global architecture. Modeling the security context makes the architectural security implications more explicit, and any architectural changes that impact security become more apparent.

Such context should be integrated in the policy modeling. XACML provides the concept of policy combination, which combines several policies into an integrated policy set. Different policy combination algorithms, such as permit-override and deny-override, are provided as part of the standard, and we extend them with structure-override and type-override, which gives the structure and the type final authority on granting permissions. The XACML framework, combined with our explicit modeling of architectural context, supplies necessary flexibility in modeling architecture security.

3.2.4. 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 comprise of 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.

3.2.5. 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.

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. A detailed example is given in Section 4.

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.

Using connectors to regulate and enforce a security contract and leveraging advanced connector capabilities will facilitate supporting multiple security models [28]. These advanced connector capabilities include the reflective architectural derivation of connectors from component specifications, composing connectors from existing connectors [24], and replacing one connector with another connector.

3.2.6. Syntax of Secure xADL

Figure 1 depicts the core syntax of Secure xADL. The xADL ConnectorType is extended to a SecureConnectorType that has various descriptions for subject, principals, privileges, and policy. The policy is written in the XACML language. Similar extensions are made to other xADL constructs such as component types, structures, and instances.

```
<complexType name=”SecurityPropertyType">
  <sequence>
    <element name="subject"
           type="Subject"/>
    <element name="principal"
           type="Principals"/>
    <element name="privilege"
           type="Privileges"/>
    <element ref="xacml:PolicySet"/>
  </sequence>
<complexType>
<complexType name="SecureConnectorType">
  <complexContent>
    <extension base="ConnectorType">
      <sequence>
        <element mame="security"
           type="SecurityPropertyType"/>
      <sequence>
    <extension>
  <complexContent>
</complexType>
<!-- similar constructs for component,
structure, and instance -->
```

Figure 1, Secure xADL schema

4. A CASE STUDY: COALITION

Architectural modeling is instrumental for architects to design architecture and evaluate different alternatives for possibly competing goals. With the modeling capability introduced by Secure xADL and the regulation power enabled by secure connectors, architects are better equipped for such design and analysis on security.

In this section, we illustrate the use of the secure software architecture description language with a coalition application. We present two architectures, each has its own software and security characteristics. We also describe how to specify related architectural policies.

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 software architecture is written in the C2 architecture style. In this style, the components send and receive requests and notifications at their top and bottom interfaces, and the connectors forward messages (requests and notifications) between their top interfaces and bottom interfaces. The two parties participating in this application are US and France.

4.1. The Original Architecture

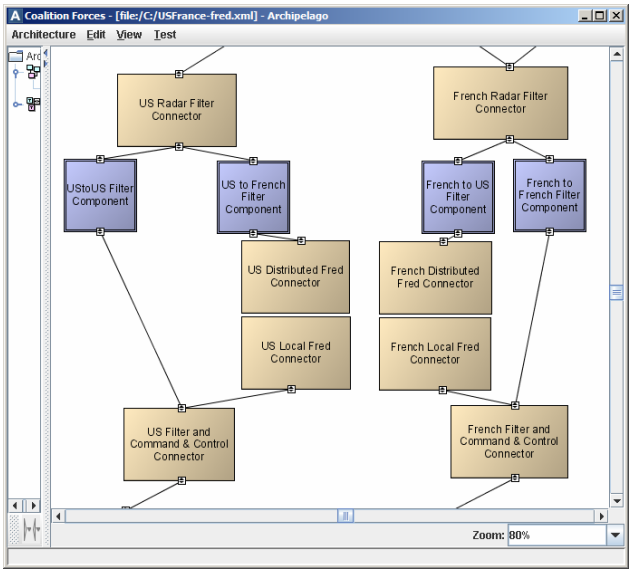

Figure 2, Original Coalition

Figure 2 illustrates the original coalition architecture, using our Archipelago architecture editor [4]. 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. The regular rectangles are connectors. 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 send those safe messages through 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.

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. First, it is rather complex, This architecture uses 4 Fred connectors (US Local, US Distributed, French Local, and French Distirbuted) and 2 components (US to French Filter, French to US Filter) to implement secure data routing such that sensitive data only goes to appropriate receivers. Second, it lacks conceptual integrity. It essentially uses filter components to perform data routing, which is a job more suitable for connectors. Third, it lacks reusability, since each filter component has its own internal logic, and they must be implemented separately.

4.2. An Alternative Architecture with a Secure Connector

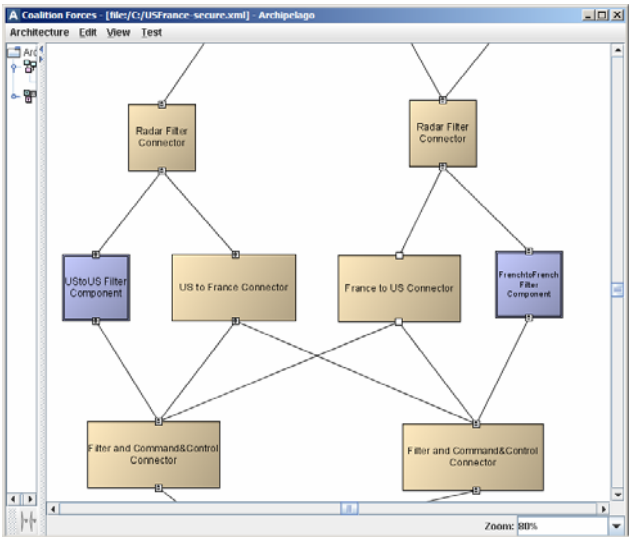

Figure 3, Coalition with a Secure Connector

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. 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 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. A shortcoming of this architecture is that the secure connectors can see all traffic, thus they are obvious targets for penetration, and their breach leads to secret leak. An architect should balance all such tradeoffs.

4.3. The Architectural Policies

Our approach bases the architectural access control decisions on security policies of architectural constituents. Different architectural constituents can execute different policies. For example, an individual constituent can execute its own local policy, while the architecture might adopt a global policy. There are also different types of policies about instantiating, connecting, and messaging to assure proper architectural access control.

```
<connector id="UStoFranceConnector">
 <security type="SecurityPropertyType">
  <subject>US</subject>
   <Policy RuleCombiningAlgId=
             "permit-overrides">
    <Rule Effect="Permit">
     <Target>
      <Subject>
       <AttributeValue>
        USToFranceConnector
        <SubjectAttributeDesignator
         AttributeId="subject-id"/>
      <Resource>
        <AttributeValue>RouteMessage
        <ResourceAttributeDesignator
         AttributeId="resource-id"/>
      <Action>
        <AttributeValue>RouteMessage
        <ActionAttributeDesignator
         AttributeId="action-id"/>
     <Condition
       FunctionId="string-equal">
      <AttributeValue>Aircraft Carrier
      <Apply>
       <AttributeSelector
        RequestContextPath =
         "//context:ResourceContent/
         security:routeMessage/
         messages:namedProperty
         [messages:name='type']/
         messages:value/text()"/>
      </Apply>
     <Rule RuleId="DenyEverythingElse"
       Effect="Deny"/>
```

Figure 4, Message Routing Policy

Figure 4 specifies part of the local message routing policy of the US to France Secure connector. The policy is written in Secure xADL, which adopts XACML as its policy sub-language. (The XML syntax is greatly abbreviated, and indentation is used to signify the markup structure.) The connector executes as the US subject, because it is executing in the US side of the coalition application. The policy has two rules. The last rule denies every request, and the first rule permitss one request. With the permit-overrides rule combining algorithm, this policy essentially allows the explicitly permitted operation and denies all other operations. Such a secure-by-default policy follows the best security practice.

The rule applies when a US subject (the subject for which the connector acts) requests a RouteMessage action on a RouteMessage resource. The resource is of active resource, which is the capability of routing messages from one interface of a connector to another. The condition of the rule uses the XPath language to specify a content-based routing policy. It permits routing a message whose “type” value is “Aircraft carrier”. What is not shown in Figure 4 is the destination of the message, which only applies to messages directed to France.

5. CONCLUSION

Component-based software operating in a modern networked environment presents new challenges that have not been fully addressed by traditional security research. Recent advancement on software architecture shed light on high-level structure and communication issues, but has paid insufficient attention to security.

We argue that architectural access control is necessary to advance existing knowledge and meet the new challenges. We extend component specifications with core security concepts: subject, principal, resource, privilege, safeguard, and policy. Component compositions are handled by connectors, which regulate the desired access control property. We propose a secure architecture description language, based on our xADL language. This language can describe the security properties of software architecture, specify intended access control policy, and facilitate security design and analysis at the architecture level. We illustrate our approach through an application sharing data among coalition forces, demonstrating how architectural access control can be described and enforced.

The contributions of this research lie in that 1) we address the security problem from an architectural viewpoint. Our use of an architecture model can guide the design and analysis of secure software systems and help security assurance from an early development stage; 2) we provide a secure software architecture description language for describing architectural access control, arguably the most important aspect of security; 3) the language enables specifying security contracts of components and connectors, laying the foundations for secure composition and operation.

This research is still on-going work. Our future work includes 1) exploring the formal semantics of the language and developing an algorithm that can check whether an architecture meets the access control policies specified in various architectural constituents; 2) developing a set of tools (visual editing and implementation generation) to support developing with the architectural security modeling; 3) implementing the necessary run-time support for executing and monitoring the security policies. These development activities will extend our existing development environment, ArchStudio [4].

6. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation award 0205724.

7. REFERENCES

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

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

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

[4] 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, 2005. **14**(2): p. 199--245.

[5] DeLine, R., *Avoiding Packaging Mismatch with Flexible Packaging*. IEEE Transactions on Software Engineering, 2001. **27**(2): p. 124-143.

[6] 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, 2003. **15**(5): p. 1099-1119.

[7] Ducasse, S. and Richner, T. *Executable Connectors: Towards Reusable Design Elements*. in Proceedings of 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, p.483-499, 1997.

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

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

[10] 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.* Dependable and Secure Computing, IEEE Transactions on, 2005. **2**(2): p. 157-175.

[11] Jürjens, J. *Umlsec: Extending Uml for Secure Systems Development*. in Proceedings of UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language, p.412--425, 2002.

[12] Katara, M. and Katz, S. *Architectural Views of Aspects*. in Proceedings of Proceedings of the 2nd international conference on Aspect-oriented software development, p.1-10, 2003.

[13] Lampson, B.W., *A Note on the Confinement Problem.* Communications of the ACM, 1973. **16**(10): p. 613-15.

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

[15] Lodderstedt, T., Basin, D.A., J, and Doser, r. *Secureuml: A Uml-Based Modeling Language for Model-Driven Security*. in Proceedings of UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language, p.426--441, 2002.

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

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

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

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

[20] Minsky, N.H. and Ungureanu, V. *Unified Support for Heterogeneous Security Policies in Distributed Systems*. in Proceedings of 7th USENIX Security Symposium, p.131-42, 1998.

[21] Moriconi, M., Qian, X., Riemenschneider, R.A., and Gong, L. *Secure Software Architectures.* in Proceedings of 1997 IEEE Symposium on Security and Privacy, p.84-93, 1997.

[22] OASIS, Extensible Access Control Markup Language (Xacml), http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

[23] Ray, I., France, R., Li, N., and Georg, G., *An Aspect-Based Approach to Modeling Access Control Concerns.* Information and Software Technology, 2004. **46**(9): p. 575-587.

[24] Ren, J., Taylor, R., Dourish, P., and Redmiles, D. *Towards an Architectural Treatment of Software Security: A Connector-Centric Approach*. in Proceedings of Workshop on Software Engineering for Secure Systems, 2005.

[25] Sandhu, R. and Munawer, Q. *How to Do Discretionary Access Control Using Roles*. in Proceedings of 3rd ACM Workshop on Role-based Access Control, p.47-54, 1998.

[26] Sandhu, R.S., Coyne, E.J., Feinstein, H.L., and Youman, C.E., *Role-Based Access Control Models.* Computer, 1996. **29**(2): p. 38-47.

[27] Spitznagel, B. and Garlan, D. *A Compositional Approach for Constructing Connectors*. in Proceedings of 2nd Working IEEE/IFIP Conference on Software Architecture, p.148-157, 2001.

[28] Tisato, F., Savigni, A., Cazzola, W., and Sosio, A. *Architectural Reflection. Realising Software Architectures Via Reflective Activities*. in Proceedings of 2nd International Workshop on Engineering Distributed Objects, p.102-15, 2000.

[29] Tripunitara, M.V. and Li, N. *Comparing the Expressive Power of Access Control Models.* in Proceedings of Proceedings of the 11th ACM conference on Computer and communications security, p.62-71, 2004.

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

[31] Win, B.D., *Engineering Application-Level Security through Aspect-Oriented Software Development*. 2004.

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

[33] Winslett, M. *An Introduction to Trust Negotiation*. in Proceedings of 1st International Conference on Trust Management, p.275-283, 2003.