Modular Security: Design and Analysis

Jie Ren
Univ. of California, Irvine
jie@ics.uci.edu

June 2004

ISR Technical Report # UCI-ISR-04-4

Modular Security: Design and Analysis

Jie Ren
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
jie@ics.uci.edu

ISR Technical Report # UCI-ISR-04-4

June 2004

Abstract:

Software systems made of software components are becoming more and more common. This paper surveys theories and techniques for designing and analyzing security of such systems. The paper gives an overview of security models, introduces formal foundations for system composition and security engineering, and investigates available techniques for security design and analysis, focusing on each technique's adoption of composition mechanisms and support for security property.

Modular Security: Design and Analysis

Jie Ren
Institute for Software Research
University of California
Irvine, CA 92697-3425
1-949-8242776

jie@ics.uci.edu

ISR Technical Report #UCI-ISR-04-4
June 2004

ABSTRACT

Software systems made of software components are becoming more and more common. This paper surveys theories and techniques for designing and analyzing security of such systems. The paper gives an overview of security models, introduces formal foundations for system composition and security engineering, and investigates available techniques for security design and analysis, focusing on each technique's adoption of composition mechanisms and support for security property.

1. INTRODUCTION

The word "component" [125] has been used in computer software for a long time, even though its original meaning carried a different implication than its current definition. Ever since the first software engineering conference, methodologies using mass-produced software components have been proposed in various forms. However, it is not until the early 90s, with the maturation of object and component technology, the proposal and adoption of component standards, the proliferation of PCs, and the ubiquity of Internet connections, did component-based software engineering become an important paradigm in practice [31].

Much of the previous work has been focusing on the functionality side [74]. The core set of questions that have already been investigated is: given the functionalities of a set of components and the desirable composite functionality, what components can be used and which one should be selected? How can they be adapted, if necessary? Is there any need for development of new components? Another set of important questions, also enjoying fruitful research, has been how to design a component to maximize its reusability and how to describe and expose the various ways to reuse a component. Significant progress has been made in answering these questions and transferring solutions into usable technologies.

However, before component-based software engineering can achieve its full potential, other extra-functional properties, such as performance, reliability, and security, must also be addressed. We need techniques for designing a software system to achieve the desired extra-functional properties from its constituent components. We also desire techniques to help us analyze such a system with respect to these extra-functional properties.

Among these extra-functional properties, security is of special importance. Given recent trends in technology deployment and advance in adversary techniques, it has become a national emergency to secure the information infrastructure. The modern networked and component-based software environment proposes several challenges for engineering security. First, components may come from different trust domains, and their security features are not always easy to certify. Second, operating such an environment needs interaction with several trust domains, requiring unprecedented flexibility of software systems. Third, given the full spectrum of possible components, expressing a complete and consistent security policy for the complete environment and each constituent component is difficult. Finally, security mechanisms should allow a wide range of granularity in software components.

This paper investigates the security property of software systems made of components, surveying techniques that have been proposed to design and analyze security for such systems. The paper proposes a framework for investigation, analyze available techniques against this framework, compare the advantages and drawbacks of each technique, and identify some issues meriting further exploration.

The paper is organized as follows. Section 2 briefly surveys proposed security models that a software system can adopt. Section 3 lists categories of components studied in this paper, mechanisms to connect them, and challenges on security imposed by them. Section 4 proposes a framework under which techniques studied in literature is surveyed and compared. Section 5 examines security design and analysis techniques in detail, focusing on the security issues each technique is trying to address and the composition mechanism each technique utilizes. Section 6 makes some discussion and outlines an agenda for future research.

2. SECURITY MODELS

Because security is a very broad subject, this section only gives a brief overview of security models. These models are the most common ones that are supported by the techniques surveyed in Section 5. For other security topics, Bishop provides a comprehensive and recent overview [15].

The main security properties are **confidentiality, integrity**, and **availability** [85]. Confidentiality ensures there is no improper information disclosure. Integrity ensures there is no improper information modification. Availability ensures there is no improper denial of service.

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 [112].

The most basic type of security mechanism to enforce secure access, solidly established ever since the seminal work of Anderson [5], 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 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 that it can be thoroughly verified. A more comprehensive and deeper treatment of reference monitors can be found at [15].

Security policy composition, which occurs when multiple sub policies coming from different sources are combined into an integral policy, has been extensively studied [16, 60]. 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. Since the topic of policy composition is about composition of passive policy, and the interest of this survey lies in the composition of active software and its security implication, the topic will not be covered further.

The focus of the rest of this section is about security models. These models are utilized by the techniques surveyed in Section 5. There are different types of security models. Two common types are access control models and information flow models.

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.

The Access Matrix Model is the most common discretionary access control model. It was first proposed by Lampson [77] and later formalized by Harrison, Ruzzo, and Ullmann [48]. In this model, a system contains a set of subjects (also called principals) that has privileges (also called permissions) and a set of objects on which these privileges can be exercised. An access matrix specifies what privilege 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.

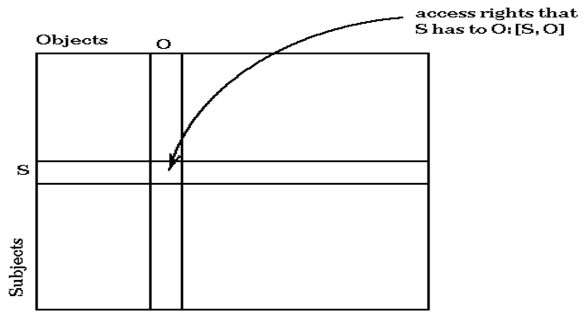


Figure 1, Access Control Matrix

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 [28]. For example, in Figure 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.

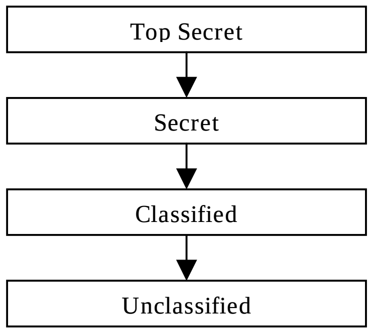


Figure 2, Dominance Lattice

The most famous MLS MAC model, which is a model for confidentiality, is the **Bell-LaPadula model** [9]. 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 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 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** [17] 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 [109].

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** [22] 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 duty. The model have are 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 the certified procedures. This model is not as formal as other models, though. It is not easy to analyze and enforce.

The **Role-based Access Control Model (RBAC)** [113] is a more recent development. It introduces roles as the entities that are authorized. In real environments, a user can perform different roles in different contexts, even though the identity of the user remains the same. A role-based access control model captures this concept naturally by introducing an extra level of indirection, role, into the norm of subject/object/privilege. Instead of authorizing a subject's access to an object, the authorization is expressed as a role's access to an object, and the subjects can be assigned to different roles. This model eases management of users, roles, and accesses. It allows roles to form a hierarchy. It enforces principles such as least privilege and separation of duty. The model supports more timing and dynamic constraints than usual access control models.

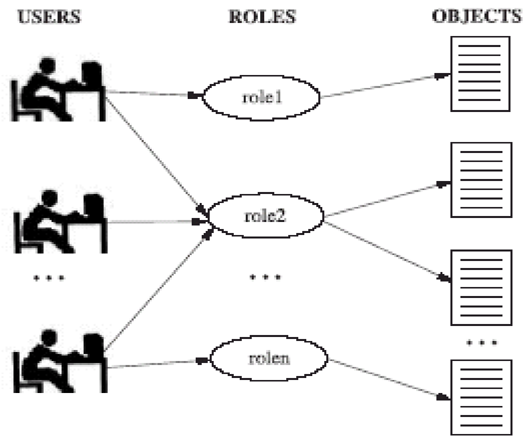


Figure 3, Role-based Access Control, from [112]

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 [76]. **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 principals so that there are no covert channels. They do not suggest how this can be achieved [85].

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 [45], 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 [124] utilizes information functions to require that low level agents cannot deduce information about high level agents. Restrictiveness [82] requires that low level agents cannot differentiate between possible states after certain state transitions. Correctability [61] 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 [134] 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 [111]. 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 [116], 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 principals 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.

3. COMPONENT TYPES AND CONNECTION MECHANISMS

The word "Component" in software has a long history and is heavily overloaded. Different authors have used this word to designate various types of entities. This section gives a brief overview of the types of components that will be studied in this survey.

Components, whether they are the results of decomposition from a component of a higher level of abstraction, or they are the units to be composed into a composite component, need to interact with each other, possibly to achieve a common goal. Different types of components require different interaction and connection mechanisms. In addition to types of components, this section also discusses the mechanisms connecting components together.

When designing and analyzing security of a system consisting of its constituent parts, the modular structure brings up new challenges not encountered in a monolithic entity. Different types of components and different connection mechanisms may bring different types of security issues.

Components are classified into four types: abstract computation, module/object/component, CBSE (Component-based Software Engineering) Component, and COTS (Common-Off-The-Shelf) component. This catalog of component types, connection mechanisms, and security challenges does not intend to be exhaustive. It lists only representatives found in literatures and serves as the foundation for further discussion in this paper.

3.1 Abstract Computation

In the formal methods community, a component generally refers to some abstract computation. Correspondingly, the connecting mechanism is generally expressed as the input and output relationship between the computations.

For example, in the Abadi-Lamport framework [1] (see Section 5.1.1), a component is generally an abstraction of the underlying computation, likely described with various types of logic. Under the setting of the trace-based information flow security, such as MAKS [80] and Non-Interference [45](see Section 5.1.3), a component is a computation that is expressed as the set of input/output traces of the computation. A component in the process algebra approach [36, 37, 39] (see Section 5.1.3) is a process that changes its state depending on the input.

The connection mechanism for Abadi-Lamport specifications is the general logic conjunction, where the specifications of two smaller computations are juxtaposed and the composite system should satisfy both specifications. The composition between traces takes the form of outputs of one trace becoming inputs of another trace. The composition in process algebra is similar, where one process takes input from another process's output. The two processes share a common event.

As will be clear in Section 5.1, the composition utilized formal computations raises two types of challenges for security. The first is how to prove that one component will enforce proper access control for other components. This can be handled by the usual Abadi-Lamport framework. The second challenge is bigger: how to ensure one component cannot acquire information about other components. The usual Abadi-Lamport framework cannot handle this challenge very well, requiring new formalisms and techniques.

3.2 Module, Object, and Component

In the more common case, a component designates a smaller part of a larger system. In most imperative programming languages, the component is named as either a module, a function, or a procedure. The connection mechanisms are function calls between these procedures. In object-oriented programming languages, the basic components are objects, and these objects are connected together by sending and responding to messages between each other.

Software architecture is a more recent approach for developing large and complex software [119]. It views the constituent parts of a system as components and connectors. Components in this setting, which should not be confused with general uses of the term in other contexts, perform computation. Connectors are in charge of communication between components. Components can take many forms in this setting. It can be as small as a function or an object, or it can be as large as a complete application. There are also many types of connectors [89]. The dominant types are variants of procedure calls: local or distributed, synchronous or asynchronous. Some other common forms of connectors are blackboard data repository and event publication/subscription.

Modules, objects and components are the most common constituent parts of software systems. Most design and analysis techniques for security apply to these components. Some security questions that have been investigated are: where the security features should be allocated, how to assure policy enforcement, and what form will best facilitate the flexibility and evolution implementations.

When answering these questions, one issue relevant to software research is how to separate security from other functional and extra-functional concerns. Some promising techniques addressing this issue are meta-object protocols (see Section 5.4) and the aspect technology (see Section 0). These technologies extend the standard object-oriented technology. They utilize the connection mechanism, object method, in novel manners. They split objects into either base objects and meta-objects, or base objects and aspects. They connect the split objects through compile-time manipulation or run-time instrumentation. These technologies can handle security property such as access control and data encryption in a flexible manner.

3.3 CBSE Components

Component-based Software Engineering (CBSE) [125] takes a more specific view towards components. In this context, the components generally adhere to a component model, such as COM/COM+, JavaBeans, CORBA Component Model, and .NET components. The specific component model constrains the exposed form of a component, such as its external interfaces, syntactic and semantic descriptions, and deployment constraints.

The component model also provides the connection mechanism. The dominant mechanism is procedure calls. Two components are connected together when one component invokes a service provided by the other. The connection is generally facilitated by a broker supplied by defined in the component model. The broker masks all communication-related details, such as locating components and marshaling data. Developments that are more recent provide advanced brokers that support some advanced connection services like transactional methods and asynchronous communications.

CBSE components generally come without source code. To develop a complete application, components that come from different sources need to be connected together. These issues raise challenges for security design and analysis, as discussed in Section 3.4.

3.4 COTS Component

Like “components”, Common-Off-The-Shelf (COTS) is another overloaded term. In this classification, it is used to refer to software such as operating systems, databases, and word processors. Compared to Section 3.2, these components are not developed in-house, and source code is generally not available. Compared to Section 3.3, these components do not conform to any form of a component standard, and COTS components’ granularity is generally larger. As a result, integrating COTS components generally requires custom-made connection mechanisms.

Like CBSE components, COTS components generally come without source code. They may also come from different sources. The lack of source code and the heterogeneous origin of CBSE components and COTS components impose new challenges that do not exist when all components come from the same trusted origin with full source code. Lindqvist and Johnson identifies security risks present in the life cycle of using commercial off-the-shelf software products [78]. They divide the life cycle into the following phases: component design, component procurement, component integration, Internet connection of system, system use, and system maintenance.

In the component design phase, the design can be inadvertently flawed. Even worse, it might be intentionally flawed by malicious designers. The component might contain excessive functionality that is not necessary for a usage scenario. The design of the component might be open, or it can be widely spread, which gives adversaries precious information. The documentation for design might be insufficient or even incorrect.

In the component procurement phase, a procurer might make a decision before conducting sufficient validation for the component to be procured. During the delivery, a validated component might come through an insecure channel that might tamper the component.

During the component integration phase, the components integrated might not match with each other’s product security levels. The integrator might only have insufficient understanding of the integration requirements.

When a software system is connected to Internet, the external exposure is significantly increased, and easily available intrusion information and toolsets can be used against the connected system. The system can unintentionally execute malicious executable content downloaded from Internet. The Internet connection can also be used as an outward channel for stolen information.

During normal system use, the system can be used in unintended ways, and the user might not fully understand the functioning of the system.

Finally, in the system maintenance phase, fixes and updates to a system might be insecure, some side effects can happen due to maintenance, and there can exist maintenance backdoors that can be exploited maliciously.

To manage these security risks, Lindqvist and Johnson suggest that users should have a well-defined and relevant security policy[78]. Users should adopt a holistic perspective, partition the system into smaller parts shielded from each other, confine distrusted components, anticipate contingencies, and actively evolve the system and remedy defined flaws. The user organization should support secure operations from all levels of management. End users should also be aware of previous security breaches and learn from past mistakes.

4. SURVEY FRAMEWORK

This section introduces a framework that will be used to compare the techniques that have been proposed for designing and analyzing security for systems made of components. The focus is how each technique helps achieving modular. How the technique responds to the following questions is studied.

- **Security Model**: What kind of security model does the technique adopt or support? Does it support discretionary access control, mandatory access control, or information flow security?
- **Component Type**: What kind of components does the technique integrate? Is the component an abstract computation? Is the component general software? Is

the component compliant of some component models? Is the component a large COTS component?

- **Connection Mechanism**: What connection mechanism does the technique provide? What mechanism does it rely on? What are the security limitations of these connection mechanisms? What kind of dynamism will the technique address?
- **Approach**: Is the technique a top-down approach or a bottom-up approach? A top-down approach begins with a system wide security requirement and develops how security should be addressed at each level of abstraction to achieve the system wide security goal. A bottom-up approach will determine what security can be achieved from the constituent components and what changes need to be made to achieve full satisfaction if the initial result is not satisfactory. Some techniques can work in both manners.
- **Formalism/Tools**: What formalism does the technique use? How much automation can the formalism support? How much automation can be conducted at run-time? What tool does the technique employ for design and analysis?

The framework is illustrated in Figure 4.

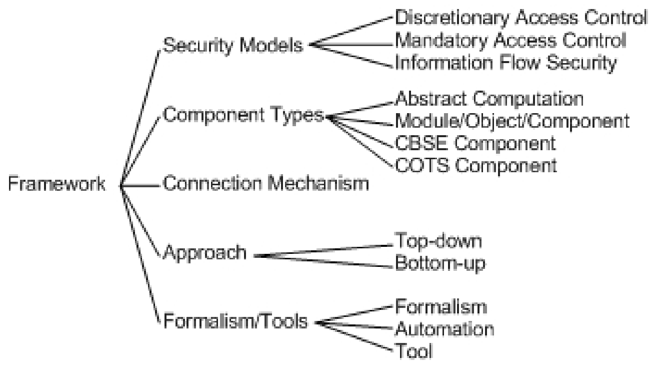


Figure 4, Framework of Survey

5. TECHNIQUES FOR DESIGN AND ANALYSYS OF MODULAR SECURITY

This section utilizes the framework developed in Section 4 to classify techniques proposed in literature. Since the framework tries to cover as many facets as possible, each technique does not always have every aspect.

Section 5.1 surveys the formal methods available for secure composition. Along with the formal models outlined in Section 2, they can be used as the formal foundations for other techniques.

From Section 5.2 to Section 5.4, some simpler techniques used to implement and compose real software are investigated. Based on its complexity and expressiveness, these techniques are categorized as following: wrapper, agent, and Meta Object Protocol.

Section 5.5 surveys techniques used to specify security requirements of components. A component in this section is more complex than components in the previous sections. They are deployable units, as those required by major commercial CBSE technologies.

Section 5.6 investigates some general composition frameworks that compose secure modular systems. Section 0 explores an especially powerful composition technology, the aspect technology. Finally, Section 5.8 surveys architectural techniques for security design and analysis.

5.1 Formal Foundations

5.1.1 Abadi-Lamport Composition in Alpern-Schneider Framework

In the formal method filed, 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 [4].

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 composite behaviors under this framework comprises of 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 pro0perties 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 [87]. The reason, suggested in [86], 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 [114], 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.

5.1.2 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 [49]. 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 [51]. 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) [99, 100] is another logic-based method to reason about security of components and their composition. It uses PVS [101] 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 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, it is more difficult 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 [27, 44].

The CSS framework is used to prove that a file manager always returns a secure file handle to a process manager [104]. 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 a small problem. These approaches also require highly skilled professionals with special expertise and training.

5.1.3 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 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.

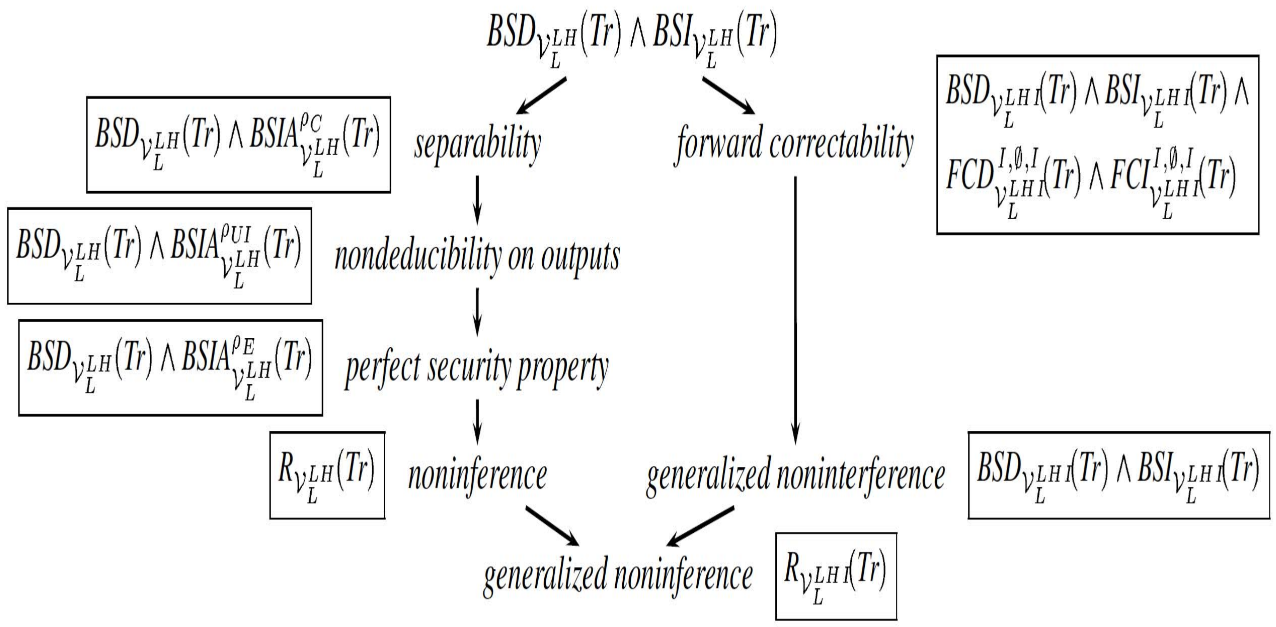
Naturally, most unifying frameworks are based on trace and logic because these are used for defining most of the properties originally. Four representative frameworks are outlined here. These efforts lay down the foundation to study securely composing abstract computations for confidentiality.

John McLean proposes the first such framework, Selective Interleaving Function (SIF) [84, 86]. 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 their equivalent functions.

Peri et al. suggest a simple unification framework based on the many-sorted logic [105]. 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 [80]. 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 **Error! Reference source not found.**.



$BSD_{\mathcal{V}_L^{LH}}(Tr) \wedge BSI_{\mathcal{V}_L^{LH}}(Tr)$
$BSD_{\mathcal{V}_L^{LH}}(Tr) \wedge BSIA^{\rho C}_{\mathcal{V}_L^{LH}}(Tr)$
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 5, Information Flow Properties, from [80]

Halpern and O'Neill uses a modal logic of knowledge to unify the various properties [47]. Their framework models states of both the agents and the environment. The framework extends the notion of Non-Deducibility on Input [124] in several aspects. First, 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. Second, 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. Third, 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.

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 other formalisms.

Security Process Algebra (SPA)[36, 37, 39] is a security extension to the process algebra Calculus of Communicating Systems (CCS) [91]. 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 [134], 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) [57] to unify information flow properties [110]. 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 [86]? 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 [86]. 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.

The three composition constructs are illustrated in Figure 6, Figure 7, and Figure 8, respectively. In these figures, σ_1 and σ_2 are the components, σ is the composed system, In_i and Out_i are input and output channels.

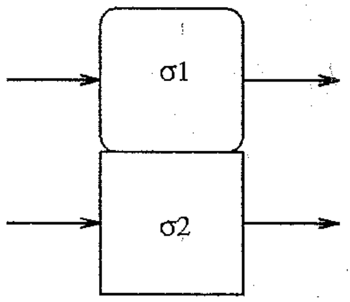


Figure 6, Product Composition, from [86]

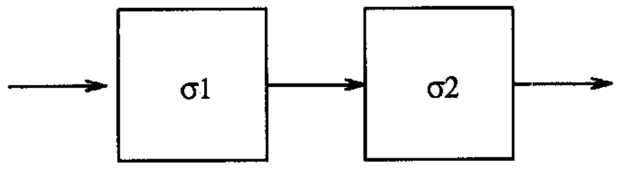


Figure 7, Cascade Composition, from [86]

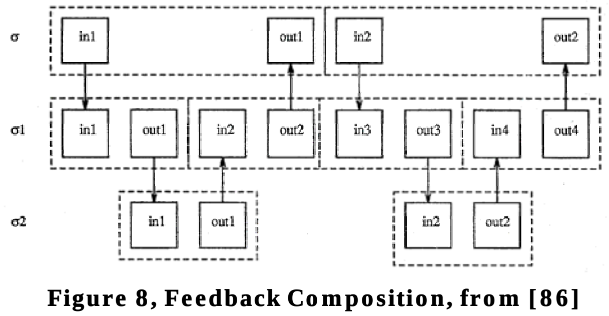


Figure 8, Feedback Composition, from [86]

Some representative results from studying composition under these constructs are summarized below. It is provided in [86] 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 [80]. 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 [136]. The framework studies both cascade and feedback and discovers that properties eliminating dependencies on inputs are preserved under feedback composition. Peri et al. [105] study the composition problem under the many-sorted logic and prove compositional properties in cascade and feedback composition using PVS [101].

Composition takes a different form in Security Process Algebra [37]. 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 [38], is used to check the compositionality of security. The power bisimulation proposed in [110] is also composable.

Santen et al. views the compositional problem under the refinement/composition perspective [114]. 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 [108]. 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 [85, 90]. As suggested in [108], "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.

5.2 Wrapper

After surveying techniques to design and analyze abstract computations, the study now turns to techniques dealing with concrete software components, beginning with the simple wrapper technology.

A wrapper is a standard technology to reuse existing software and probably extend it with more functionality. When available software provides useful capability for an environment but cannot be utilized in its current form, due to factors such as incompatible interfacing mechanisms and insufficient functionality, a wrapper is generally used to fill the gap. A wrapper is a layer of software that receives control from the invoking environment and performs additional processing before transferring control to the original software. After the original software finishes its activity, the wrapper gets back control conducts more processing before returning to the environment.

A wrapper can simply change the format of input or output parameters so that the parameters can meet the requirements of the outside environment and the wrapped software. A wrapper can also perform more complex checks and analyses that can be used to improve security.

The wrapper technology has many forms. Some agent technologies in Section 5.3 can be considered intelligent or data centric wrappers. Meta object protocols (Section 5.4) can also be viewed as object wrappers that utilize reflective and meta-level capabilities. This section focuses on simple procedural wrappers. It begins with application-level wrappers (Section 5.2.1), and then goes down to library function-level wrappers (Section 5.2.2), system library-level wrappers (Section 5.2.3), until it reaches system call-level wrappers (Section 5.2.4). This section illustrates how wrappers can be used to enhance security and survey techniques facilitating wrapper development and management.

5.2.1 Application-level Wrapper

Application-level wrappers are used to adapt existing application. Zhong and Edwards develop wrappers to make the most popular mail server, sendmail, a more secure application [140]. To tackle security risks such as accessing unauthorized resources, accessing resources in an unauthorized manner, or abusing execution privileges, they utilize the underlying mandatory access control (see Section 2.1) and least privilege execution support provided by the operating system. The architecture of the wrapped application is shown in Figure 9.

Sendmail needs to access some resources with special privileges during its execution, such as reading from configuration files, writing into user mail boxes, binding to specific network ports, and changing the security credential of processes. Due to its complexity in implementation and configuration, security breaches are occasionally reported.

After careful resource and privilege analysis, they separate the original sendmail application into two instances. Each runs in a separate security compartment with limited privilege. One runs in the "system outside" compartment. It can send emails to outside world using the credential of the original sender. It will not perform other functionalities of sendmail. The other sendmail instance runs in the "system inside" compartment. Its only responsibility is to write to users' mailboxes. These two instances communicate through each other via a newly-developed, simple, and trusted gateway called "relay". The relay redirects the output of the inside sendmail application to itself, and then sends the output to the outside sendmail application, with the credentials of the original sender.

Because the operating system enforces mandatory access control, there is no implicit communication path between the two compartments. Thus, even if the outside sendmail application is compromised, it cannot affect inside operations. And even if it is penetrated through the defense boundary, its limited privilege reduces the damage it can induce.

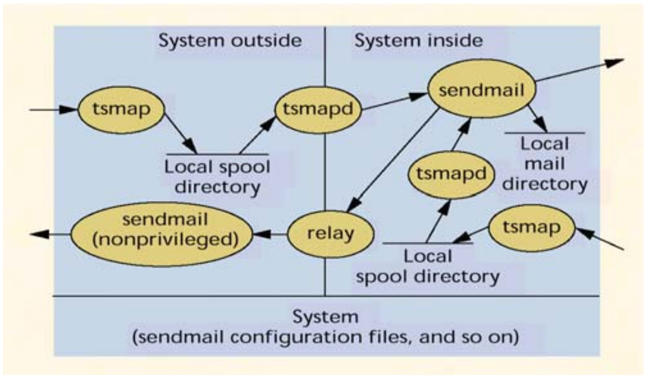


Figure 9, Wrapper of sendmail, from [140]

Another simple and trusted mail application, tsmap/tsmapd, is developed. One instance of this application is deployed in the "system outside" compartment, to receive mail from external network. Another instance is deployed in the "system inside" compartment, receiving outgoing mails from internal users. The two instances also execute with minimal privileges.

In summary, based on the mandatory access control and least privilege execution provided by the underlying operating system, with newly developed simple front-ends and a secure wrapper, the highly powerful yet very complex sendmail is reused in a secure manner to achieve the benefits of COTS software. This case study illustrates how wrappers can be used at the application level to secure applications.

5.2.2 Library function-level Wrapper

Balzer and Goldman propose a non-bypassable wrapper technology used to execute applications securely [8]. In this technology, a mediator mediates calling a usual library function by an application. A set of mediators comprises a wrapper. The wrappers can be stacked on top of each other, and each wrapper can be enabled and disabled independently.

The implementation of the technique adopts the following strategies. The application binary is patched to allow the mediating wrappers to intercept library function calls from the application. Certain parts of the memory image of the loaded application are write-protected so that the mediating mechanism cannot be bypassed by modifying memory tables. The wrappers are installed upon process creation, when the application is loaded.

Because mediators control calling library functions from the application, they can either allow or reject the call, based on security policy specified by the user. This can enforce the secure execution of any application, in addition to the protection provided by the operating system. Mediators are used to construct a safe execution environment that can securely execute ActiveX controls downloaded and Office documents with macros enabled.

In a word, mediators provide a usable mechanism that can control the execution of any application on Windows platform. They wrap function calls used by an application.

5.2.3 System Library-level Wrapper

A similar technology is used to implement the Interceptor/Enforcer that enforces access control policies in a coalition environment [118]. In such an environment, each resource of an organization can be accessed by principals from either within the organization or from an outside coalition organization. While enforcing policies can happen at the communication layer, Shands et al. argue that only the server on which the accessed resource resides has enough context and history information to enforce the full access control policy [118]. They thus choose to implement the Interceptor/Enforcer on the resource server. Implementing on the Java Web Server does not encounter many problems because Java Servlet provides flexible extension mechanisms. However, some issues must be resolved when implementing on Java Remote Method Invocation (RMI), because the reference RMI implementation does not provide any extensions allowing third-party software to augment the request handling process. Implementing on Microsoft DCOM and IIS is most challenging. They choose a simpler and less general approach over the mediator framework [8] (see Section 5.2.2) because they do not need the full flexibility. Still, they encounter numerous issues. The DCOM/IIS system dos not provide a convenient extension mechanism. The provided mechanism, custom filter, does not work as advertised in documentation. The system does not provide enough bridging support to integrate heterogeneous languages. These issues result in a wrapper that is a set of patches over several bridges. The wrapper enforces access control policies, but it lacks conceptual beauty and suffers significant performance penalties.

This approach demonstrates developing wrappers at certain level is necessary, but the effort can be hindered by the lack of suitable information and mechanisms from the existing infrastructure.

5.2.4 System Call-level Wrapper

Hypervisor. The Hypervisor [93, 94] is an early effort in providing system call-level wrappers that support more flexible security policies. A hypervisor is a loadable kernel module that can intercept systems calls and perform additional pre- and post-system call processing. Compared to non-kernel wrappers, they are non-bypassable. Since they are loadable modules, no kernel modification is needed. An application can be wrapped without any change to take advantages of the hypervisors. The policy enforced by the hypervisors is very flexible. The concept of a hypervisor can be applied to most mainstream modern operating systems that support kernel loadable modules.

The implemented Hypervisor architecture, illustrated in Figure 10, contains a master hypervisor, a set of client hypervisors, and client hypervisor management modules. The master hypervisor manages other hypervisors and provides facilities for monitoring and configuring other hypervisors. A client hypervisor implements a certain policy by injecting pre and post system call processing around standard system calls. Client hypervisors can be stacked upon each other to implement composite policies. A corresponding client hypervisor management module allows users to communicate with and configure policies for a client hypervisor.

Hypervisors support many different types of policies. They can be used for auditing. They can provide fine-grained access control. They can also enforce mandatory access controls.

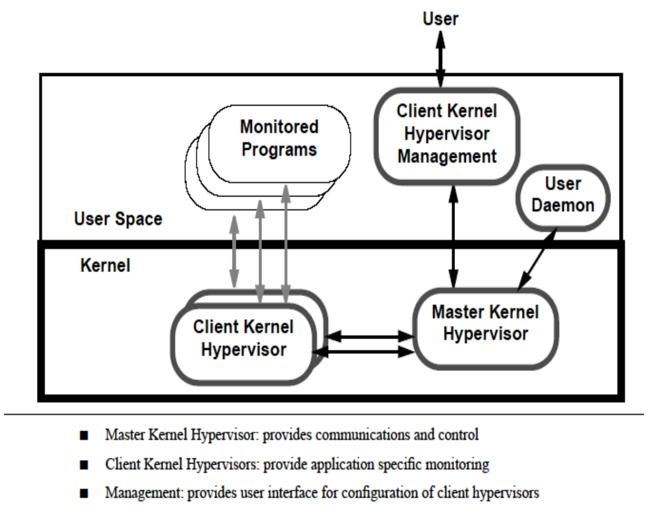

User
Monitored Programs
Client Kernel Hypervisor Management
User Daemon
User Space
Kernel
Client Kernel Hypervisor
Master Kernel Hypervisor
Master Kernel Hypervisor: provides communications and control
Client Kernel Hypervisors: provide application specific monitoring
Management: provides user interface for configuration of client hypervisors

Figure 10, Hypervisor Architecture, from [93]

Generic Software Wrapper. Generic Software Wrapper (GSW) provides more expressiveness, dynamism, and manageability than Hypervisor [40]. Its architecture is shown in Figure 11. It uses a C-based Wrapper Definition Language (WDL) to describe a wrapper as a basic state machine. The description specifies the system calls that the wrapper wraps, the patterns of events to which the wrapper will react, and the actions that the wrapper takes when these events happen. The actions can be augmenting the events, transforming the events, or simply denying the events. The Wrapper Definition Language hides the peculiarities of various flavors of operating systems by providing some common data types, so that the parameter types and return values of different but similar system calls of different systems can be described uniformly. State machine specifications and common data types achieve a degree of abstraction and enhance the portability of the technique.

The life cycle of the wrappers is as follows. At first, the system contains no wrapper. Then, the administrator installs a wrapper and specifies the conditions under which the wrapper will be activated. The wrapper will then receive `wr_install` events. When a process that satisfies the activation criteria is created, the wrapper is activated, and executes its `wr_activate` action. When the process exits, the wrapper is deactivated, and performs operations in `wr_deactivate`. Eventually, the wrapper will be uninstalled, and it has the opportunity to execute the `wr_uninstall` action. The life cycle and pluggable event processing provides flexibility in configuration.

A kernel-resident Wrapper Support Subsystem (WSS) is implemented using dynamically loadable kernel modules of common Unix/Linux operating systems. The WSS executes wrapper definitions generated by the WDL compiler, according to criteria specified through the activation criteria compiler. An administrator can communicate with WSS through a management GUI.

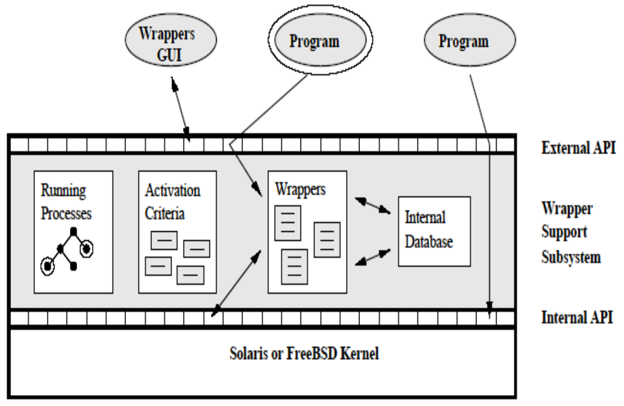


Figure 11, GSW Architecture, from [40]

Because the Generic Software Wrapper can monitor and augment each system call, it can harden COTS software in various ways. It can implement different access control models. The model can be either based on a rule set and subject/object labels, such as the Bell-LaPadula model [9] and the Biba model [10], or based on state and history, such as the Chinese Wall model [17] (see Section 2.1). The wrapper can record audit trails, analyze them in real time, and implement many schemes of intrusion detection based on the analysis. It can also be used to enhance other security features, such as transparent encryption and decryption.

The Generic Software Wrapper is not suitable for covert channel analysis [90]. Due to the limitation of the underlying Unix/Linux architecture, it cannot protect the system from compromised root programs. Since the Wrapper Support Subsystem operates in the kernel and utilizes knowledge about the kernel, rewriting it at the application-level will require much more knowledge about the application and impose significantly higher cost.

The Generic Software Wrapper technology is extended to work within a networked enterprise environment [34]. The challenges in such an environment are how to securely manage a heterogeneous environment over a network, how to manage data flow with push and pull models, and how to easily write wrappers. Extensions are added to the existing wrapper definition/query language and the wrapper database. Host and network controllers utilizing appropriate storage and communication protocols are constructed. The Windows mediator technology [8] (see Section 5.2.2) is also incorporated .

5.2.5 Discussion

The wrapper technology can be very useful to enhance the security of COTS software. As demonstrated above, it can be utilized in multiple manners to provide different forms of security, such as access control, confidentiality, and intrusion detection. Due to the unavailability of source code of COTS software, and the enormous cost of understanding and modifying COTS software by third-party developers even if the source code is available, arguably the wrapper technology is an essential constituent of secure component-based software engineering. However, before the wrapper technology can be more widely and effectively deployed, several key issues must be resolved:

The level at which the wrapper is applied. In [140], a set of customized wrappers (a simple mail server and a trusted gateway between security compartments) are developed to wrap sendmail and execute it more securely. While achieving the desired security, these application level wrappers are custom made, and can not be reused in other applications. This level of investment may be justified for reuse of popular and powerful COTS software such as sendmail. Generic Software Wrappers [40] provide a framework that can be used to describe wrappers on system calls, so each system call can be augmented with additional semantics processing. The technique employed in [118] wraps functions of a subsystem of the operating system, utilizing special knowledge of that subsystem. Mediators [8] applies to non-system call functions and implement a framework usable to wrap any function in a dynamic link library. When the abstraction level moves from the operating system to a subsystem, to general functionality, and eventually to an application, the application dependency rises, and the applicability of a wrapper technology operating at that level decreases. A technology and framework that can be applied to the widest situations is more desirable.

The information available to a wrapper. As the level of abstraction changes when moving between an operating system and an application, so is the kind of information available to a wrapper at each layer. Certain security properties can only be achieved at the application level, because they need application level knowledge, such as the mail configuration file in the sendmail application[140]. Others, like access control, are best enforced at where the access eventually happens, namely the operating system. As pointed out in [118], some information is not available at certain layers, so a wrapper technology that operates at the layer where the information is available is needed.

Security properties. As discussed in [40], wrappers are suitable for implementing access controls [8, 40, 140], audit, intrusion detection [40], and transparent encryption and decryption. They are not well suited for covert channel analysis. Further research is needed on what kind of security properties are best provided by the wrapper technology, and what kind of security needs other technologies. A related issue is the relationship between the security provided by a wrapper technology and the security provided by the wrapped software. Wrappers can augment the existing security support available from the wrapped software and provide complementary mechanisms and policies [8, 40]. A wrapper can use a different mechanism to enact its own policy, such as the Interceptor/Enforcer in [118] that does not make much use of the existing security features. A wrapper can also rely on the wrapped software, such as the wrapper's reliance on the mandatory access control support of the underlying operating system [140].

The extension mechanism provided by the underlying system. The extension mechanism that the wrapped system provides decides how easily the wrapper technology can be implemented. Generic Software Wrapper uses the dynamically Kernel Loadable Module capability available in Unix/Linux systems [40], thus its implementation is straightforward. Mediators do not have a suitable extension mechanism available on the platform they choose, so they need to resort to binary patch to make the wrapping work [8]. Implementing the Interceptor/Enforcer encounters even bigger obstacles, and the solution requires more engineering effort and is less reusable outside the application[118]. If sendmail does not provide a simple configuration mechanism to redirect the outbound messages to the secure relay, inserting the relay wrapper between the separate instances of sendmail executing in different security compartments will be very difficult. In general, if the system provides extension mechanisms in certain key decision points, inserting a wrapper at those points can be more easily accomplished.

Portability. Related to the previous issue, the wrapper technology may have a portability problem because of its dependency on system-specific extension mechanisms. Generic Software Wrapper relies on the Unix Kernel Loadable Module [40]. Mediators can only work on Windows [8]. Because of the system dependency, a certain level of system dependency is inevitable for the wrapper technology. However, research is still needed to find out to what extent a wrapper technology applicable to many systems can be developed.

Performance. Even with optimization, introducing a wrapper can still bring significant overhead to the normal execution of the system. A trust-based mechanism is proposed to reduce such overhead [55]. For more trustworthy components, the wrapper performs less work, or is deactivated at all, so the execution can finish at almost the native speed. In this approach, a trust manager controls how the wrapper should work, with the help of a trust information service, which stores trust values calculated from negative and positive events sent by wrappers during components execution.

To assure authentic trust and alleviate the problem of wrongful incrimination of components by malicious users or components, algorithms should compute trust value more reliably [54]. There are several strategies usable. The algorithm can be either more liberal or more conservative towards negative experiences with the user/component under investigation. An indirect trust based on recommendations takes the reputation of the recommender into consideration. A collective trust that is derived from consensus of components can also minimize the effect of a single malicious recommender. To further ensure the authenticity of the negative or positive events sent by a wrapper, an additional check tries to repeat the events logged and confirm their authenticity. A witness host, which receives every event that occurs in the wrapper, can also be set up to execute the composite system along with the original host. These strategies produce reliable trust-based wrappers that can reduce the execution overhead appropriately.

5.3 Agents

Agents are independent entities that augment security. Compared to the simple wrappers described in Section 5.2, they make more use of knowledge, perform more complex operations, cooperate more between each other, and possess less regular structure.

5.3.1 Gateway Agent

Bieber et al. describes an approach utilizing intelligent agents to achieve two goals [14]: introduce secure access control into a legacy application and extend access control to accommodate federated organizations.

The legacy application is a workflow system where different stakeholders in the air transportation industry, such as chiefs and pilots, use agents to access flight information. The agents are coordinated by a cooperator.

To introduce secure access, a layer of security agents that enforce a role-based access control policy is placed before the original workflow agents. The security agents authenticate the stakeholders and consult the policy before asking the original agents for information. Adding a layer at this level that utilizes the same communication pattern as the existing one minimizes the modification of the original system.

To introduce federated access that enables stakeholders from a different organization to access information that they are entitled to, a translator is placed between the two federated organizations. The translator is a special agent, operating at the same layer as original agents. It listens to requests for information about an external organization, and translates the subjects and objects involved in the request to a form that is understandable by the external organization.

All rules about access control and access translation are described using a Prolog-like language. The rules are stored in a knowledge base. The knowledge base can be updated by agents without interrupting the operation of other agents. The agent can reason about situations that are not directly visible from the rules.

The approach demonstrates that agents can be a flexible method to introduce powerful security features into applications. However, if the original communication pattern does not exhibit an agent-orientation flavor, the applicability of this approach is uncertain.

5.3.2 Secure Access Wrapper

Secure Access Wrapper (SAW) [24, 25] is a technique based on mediators that provide secure access to data but still retain autonomy of an organization when different organizations need to share information in their databases.

Previous approaches address this problem with a federated database. A federated database maintains a global schema that is an aggregation of schemas from member databases, and the federated database has a central enforcer to enforce the defined policy. This approach limits the autonomy available to each participating organization.

Lack of autonomy will make an organization less willing to share its information with collaborating organizations. To retain more autonomy in a fully distributed environment, a mediator-based approach is used in SAW. The mediator does not try to impose a global schema on the members. Instead it coordinates the access between members.

Secure Access Wrapper assumes that each organization has a Multi-level Secure (MLS) database and will define its own security lattice as the foundation for access control. The approach addresses two additional issues based on this assumption: 1) How can data be accessed by external organizations in a manner that is still considered secure by the local organization? 2) How can data be maximally shared between organizations without compromising security?

To answer the first question, a mapping between the lattices of collaborating organizations is established. Extra dominance relationships are defined between the external lattice and the local lattice. When an external subject is accessing local information, its lattice is used to decide what local lattice should be used in its capacity, based on the cross-lattice relationship. The resulting local lattice decides the access level that the external subject will be granted. When establishing cross-lattice relationships, care should be taken to avoid inconsistency, ambiguity, and redundancy.

Figure 12 illustrates the mapping with an example from the health care domain. The original lattices, from Clinic, Medline, and Hospital, are depicted by solid lines. The dotted lines establish the cross lattice mapping. For example, the "cli" from Medline is mapped to "med" in Clinic. When a subject of Medline with "cli" level accesses data of Clinic, the level it acquires in Clinic will dominate "unc", but will be dominated by "sys".

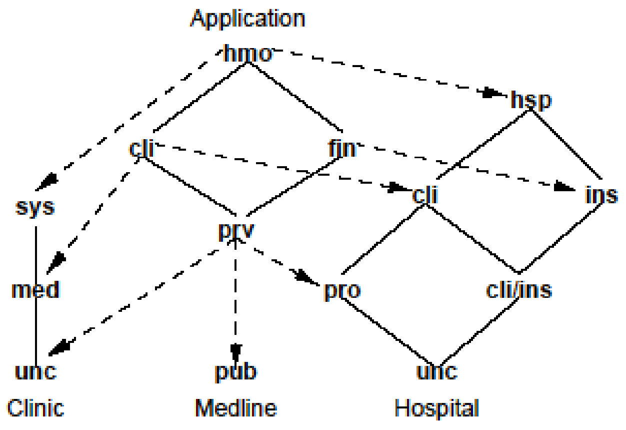


Figure 12, Lattice Mapping, from [25]

The answer to the second question comes from appropriate classification of attributes of a database relation. While assigning an attribute a higher level of classification will increase security, it will reduce the chance of sharing and may not be desirable in certain situations. The key for maximal sharing of information is to find the minimal classifications of attributes that still satisfy the classification constraints. Secure Access Wrapper expresses classification constraints in a constraint graph. After finding the upper bound of classifications, SAW searches for the lower bound. These bounds are used to assign secure levels.

Architecturally a mediator is placed between the consumer of data and the data itself. The consumer and the data might belong to different organizations. When the consumer tries to access data, it identifies itself. The mediator accesses the data on behalf of the consumer, based on the established cross organization mapping and sharing relationships. The mediator performs further sanitization before finally releasing data back to the consumer. It also records audit trails of data access.

The mediator performs the restriction, sharing, and sanitization based on rules in a rule system that is independent of the mediator software. The rules describe the security policy and primitives for enforcement. The mediator can also interact with a security officer if it cannot finish a task automatically.

In summary, the mediator of Secure Access Wrapper is agent-like software that achieves autonomous secure sharing of information between heterogeneous organizations by mapping lattices and labeling attributes appropriately.

5.3.3 NRL Pump

The NRL Network Pump [65] is a device that enables secure communication between components that run at different security levels in a multilevel secure (MLS) environment. It can be used for fast and secure communication between components operating at different security levels.

A component at a lower level needs to send data to a component at a high level. To increase the reliability of communication, an acknowledgement signal is sent back from the high level component to the low level component after a successful communication.

The acknowledgement can be used to exercise flow control so the low level sender will not receive an acknowledgement if its continuous sending data will hit a full buffer. Using acknowledgements to prevent the buffer from getting full or staying full is very important, because the event that a buffer becomes or stays full can be used as a storage covert channel.

However, using an acknowledgement itself creates a timing covert channel, because the timing when a low level component receives the acknowledgement can be used by a malicious high level component to send information secretively.

To reduce the danger of such a timing covert channel but still provide reliable communication, the Data Pump [66] regulates the delivering of acknowledgements. Its architecture is shown in Figure 13. A pump is a communication device between a low level component and a high level component. It contains separate buffers for the low level component and the high level component. It uses separate trusted processes to communicate with the low level component and the high level component. After it receives the acknowledgement from the high level component, it does not deliver the signal back to the low level component immediately. Instead, it inserts a random delay that matches the statistical average delay of acknowledgements before sending the acknowledgement back. Since statistically the delayed acknowledgements have the same delay as the original acknowledgements, the technique does not affect the reliability and throughput of the pump. However, the statistical noise thus introduced can effectively reduce the capacity of the timing channel, and the degree of reduction can be controlled by the designer in tradeoff with other design factors.

A network version of Pump has also been developed [67]. The Network Pump supports multiple communication sessions. This makes the arrival of acknowledgements at a low level component even more difficult to predict, effectively increasing the statistical noise of the covert channel and achieving better security. The Network Pump also prevents denial of service by monitoring the acknowledgement rate so that no low level component can send faster than what can be handled by the high level component.

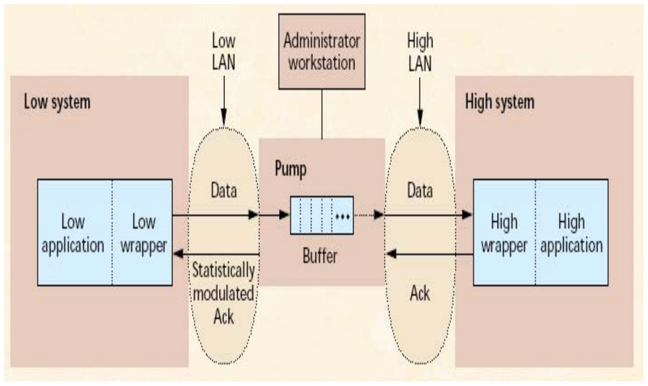


Figure 13, NRL Pump, from [67]

The NRL Pump facilitates integration of multiple MLS and non-MLS systems. Simply integrating several multilevel secure systems may not result in a satisfactory solution. A straightforward integration solution is connecting different MLS systems together and allowing communications only between senders and receivers of the same level. This solution suffers from following limitations: 1) If the systems do not have the same level of assurance, the total assurance level is the same as the weakest one among the system. 2) The approach cannot be used to integrate systems that have only one level of security. 3) Because of the high cost and slow pace in MLS development, the approach cannot utilize available non-MLS technologies and applications. An easy to use and flexible alternative is using the NRL Pump together with multilevel workstation (a workstation trusted to securely manipulate information from multiple levels) and downgrader (devices lowering the security level of data). These devices assure that "no higher level information should pass to lower level users/processes and lower level information should be available to higher level users/processes" [64].

5.3.4 MLS METEOR

Kang and Frosher develop MLS METEOR, a multilevel security (MLS) extension to a traditional workflow management system (WFMS) METEOR [62, 63]. It allows workflows executing at different security levels to aggregate into a composite workflow according to a defined policy.

In contrast to the approach integrating multiple MLS systems into a composite MLS system using the NRL Pump (see Section 5.3.3), MLS METEOR argues for using single level workflow system as much as possible and integrating multilevel components only when absolutely necessary. They believe that composing an MLS WFMS out of multiple single level WFMS is the only practical solution.

They adopt a layered method to construct the architecture of the workflow management system. A task in the higher layer workflow is implemented as a complete workflow at a lower layer. The tasks in the lower layer workflow might not be dominated by the security level of the higher layer task. This situation requires repartitioning the implementing task into several subtasks.

A transition from a task in one workflow to another task in another workflow is called an external transition. External transitions are the cornerstones of MLS workflows. Each transition crossing security levels is an external transition because a workflow can only contain tasks belonging to the same security level.

To communicate crossing different security levels, a one way communication device (like the Network Pump, see Section 5.3.3) is used, and release policy servers in the sending domain and receive policy servers in the receiving domain assures the proper security policy. These policy servers reside on synchronization nodes in each domain that serve as both the entry/exit points for information passing and proxies for tasks of different domains.

A supporting environment supports designing and executing MLS workflows. An editor allows the designer to design the domains and roles of the MLS workflows, in addition to standard workflow artifacts such as networks, tasks, arcs, and data. The resulting design can be compiled into executable code, if actions for each task are available. The code executes in a standard single level workflow management system. The code for the synchronization nodes, generated by a compiler of the design environment, ensures proper multilevel security semantics during MLS workflow execution.

In summary, this methodology intends to maximize reuse of existing single level software components in providing multilevel security capability. The key connection mechanism is a one-way communication device implementing secure information release. The structure of a workflow can be modified to meet the MLS security requirements. A supporting environment consisting of an integrated designer, a compiler and a run time system supports security design and execution.

5.3.5 Workflow Partition

Atluri et al. propose a novel system for securely executing workflows in a distributed environment where nodes participating in the execution do not necessarily trust each other [7]. Because of the performance gain and the distributed nature of certain applications, a workflow management system executing steps of a workflow in different nodes is necessary and desirable. The description of a distributed workflow contains data and control dependency among the steps of the workflow. This dependency can be utilized by malicious workflow execution agents to manipulate the result to suit the purpose of those agents. For example, if a ticket agent of the company *A* knows the booking agent of a client will purchase ticket from the company *B* if the quote received from the company *A* is higher than a certain limit, the agent of the company A can propose just below the limit, and direct the flow to satisfy the interest of the company.

To solve this problem, the starting agent divides the workflow into smaller *restrictive partitions*. Each smaller workflow partition can be executed on agents belonging to the same class of interest (the company *A* and the company *B* of the above example belong to different classes), because no sensitive information is contained in the smaller workflows. Certain neutral agents that have no conflict of interest with existing classes of competing agents are added. These neutral agents collect results of current workflow execution steps, direct executing remaining steps, and further partition the remaining steps if necessary. Since this is a distributed environment, the starting agent only bootstraps the executing process. It will not serve as a central monitor or coordinator. The neutral agents cooperate to complete the workflow.

This approach achieves confidentiality. It prevents explicit information flow among potentially antagonistic agents. No information will be leaked to malicious agents. The approach accomplishes this through using neutral agents that save data and control execution for the competing agents. A drawback of the approach is that it requires complete knowledge about the classes of conflict of interests and dependency of data and control before partitioning the workflow and executing the resulting steps.

5.3.6 JIF/Split

JIF/Split [139] is a system that partitions a program into components such that no invalid information is passed between components during the program execution. This is a programming system that enforces information flow security.

Since information flow security is a property for all possible program executions, it is not well suited for run-time access control where only information available to the current execution can be used to make decisions. A more static approach enumerating all possibilities is superior.

The input to JIF/Split is an annotated program and a trust declaration. The declaration specifies what agents (hosts) are trusted by others so that the agents can receive information necessary for computation from the trusting hosts. JIF/Split outputs a split program, if there is a possible splitting. The execution of the split program conforms to the trust declaration and ensures secure information flow. Figure 14 depict the JIF/Split architecture.

JIF/Split adopts the decentralized label model proposed in [97]. The model gives each data a confidentiality label and an integrity label. The confidentiality label specifies what principals are allowed access, and the integrity label expresses how much trust is placed on the validity of the data. Declassification on confidentiality and endorsement on integrity are used to loosen confidentiality and integrity requirements for meeting some realistic needs.

JIF/Split analyzes the implicit data flow that comes along with the control flow. It tracks where a field is defined or used in a program and checks whether the flow is allowed by the trust declaration. A set of primitives is developed so agents can pass data and control between each other without revealing information to distrusted agents. The agent partitioning the program must be trusted by each participating agent. It ensures that all participating agents execute the same split program, and optimize data access without violating information flow security.

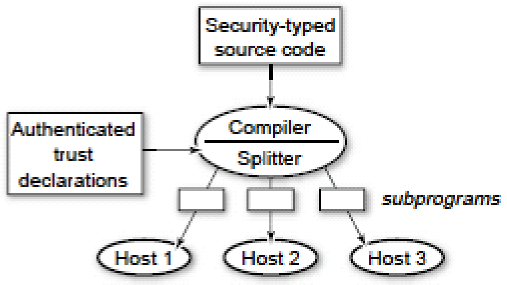


Figure 14, Secure Program Partitioning, from [139]

JIF/Split has a number of advantages. It can enforce a stronger security than simple access control. It enables partially trusted agents to cooperate for a computation. It is fully automated in assuring security. It supports explicit trust declarations. Its drawbacks are that it can only work when program source code is available, and the policy to enforce must be known prior to the splitting.

The workflow partition technique (see Section 5.3.5) can work on a different format (workflow description) and support more dynamism because it allows dynamic partitioning even though it still requires complete prior knowledge. JIF/Split approach could be extended to a general binary-only dynamic situation if enough metadata is available for binary components.

5.3.7 SafeBot

Filman and Linden proposes an intelligent agent-based approach to attack the security problem [35]. Such an intelligent agent is named SafeBot. SafeBots are ubiquitous, communicating, and dynamically confederating agents that monitor and control the execution of components of existing applications.

Simple SafeBots are wrappers around existing software components. They monitor the incoming and outgoing traffic of the wrapped component. A SafeBot can require further authentication, reject inappropriate access, detect suspicious activities (probably with help from other SafeBots), audit user and component actions, randomize duration of component invocation to frustrate covert timing channels, and thwart leaking sensitive information.

SafeBot Agencies are more powerful agents that do not wrap existing software components. They authenticate users and services, monitor security status, profile behaviors, facilitate information exchange between simple SafeBots, reason about trust relationships, and support human security officers and administrators.

The proposed SafeBot framework consists of a language, a tool, and a library, shown in Figure 15. The framework aims at generating deployable powerful SafeBot wrappers around existing components and avoiding labor intensive manual wrapping. The three constituent parts are: 1) The OntoSec language. This language describes properties and communications of SafeBots. It is expressive, including security ontology that can specify goal, action, event, knowledge, policy, status, belief, and other concerns. It is directly computable within a reasonable amount of time. The language unifies programming with reasoning. 2) The Swathe compiler. The compiler automatically compiles OntoSec specifications into deployable SafeBot wrappers. To be wrapped by a SafeBot wrapper, the original component should be specified (with a formal specification facilitating the automatic generation of the SafeBot), sequestered (not directly invocable from intruders), and substitutable (so the SafeBot wrapper can substitute it). 3) The SecLib library. The library contains algorithms, mechanisms, and existing SafeBots that understand the OntoSec language and can be assembled into a new SafeBot. When the Swathe compiler generates new SafeBots based on an OntoSec specification, it deals with the syntactic issues of wrapping, leaving the real semantics to be handled by items in the SecLib library. SafeBots generated from these items can understand the environment, reason about threats, and plan possible actions.

SafeBots are important security infrastructure, so protecting themselves is an important goal. Possible protections include using cryptography in communication, running SafeBots in dedicated hardware, reasoning trust of SafeBots, and isolating rogue SafeBots.

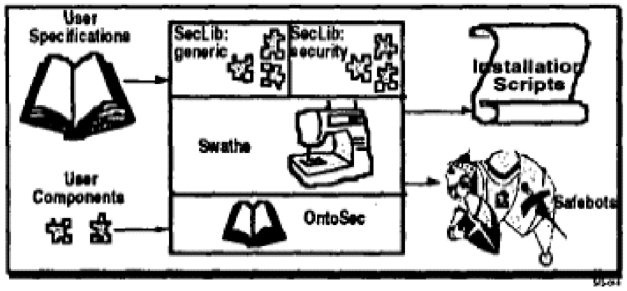


Figure 15, SafeBot Framework, from [35]

Compared to general wrapper mechanisms (see Section 5.2), SafeBot stands out as communicating intelligent wrappers. The SafeBots maintain knowledge through security ontology, and they reason before taking possible actions. Other wrapper technologies are straightforward wrappers that do not communicate with each other, and do not exploit reasoning. This could give SafeBots more power.

However, the framework proposal has several severe limitations that hinder its wider applicability. First, it requires that the wrapped software component has a formal specification, is invocable by a SafeBot wrapper, but is not accessible by intruders. Few components can satisfy these stringent requirements. Second, to facilitate automatic generation of wrappers so that no manual wrapping is needed, an ontology language covering a significant part of security is proposed as the foundation for specifying behaviors and communications of SafeBots. Given the evolving nature of the security domain, an extensible language is essential, and a reasonably large core of the ontology must be available before the approach could describe any real applications. Finally, a relatively mature library is required before any meaningful wrapper can be generated, although this problem can be mitigated as the adoption process moves forward.

5.3.8 Discussion

Agent techniques provide security for modular software by either connecting software components together or partitioning a system into appropriate components.

Gateway Agents (Section 5.3.1), Secure Access Wrappers (Section 5.3.2), and NRL Pumps (Section 5.3.3) are connection mechanisms when different organizations need to share data. The Gateway Agents approach can be adopted in more general cases. The NRL pump facilitates data exchange between different levels in a single MLS environment. Secure Access Wrappers are used when several MLS organizations need to share data with each other.

Techniques outlined in Sections 5.3.4, 5.3.5, and 5.3.6 are all top-down approaches for securing a modular system by partitioning it appropriately. They all need prior complete knowledge about the system to securely partition it. They differ in the formalism that they work on (workflow description in Section 5.3.4 and Section 5.3.5, source code in Section 5.3.6), and what extra support they need (network pump in Section 5.3.4, neutral agents in Section 5.3.5, and data/control transfer support in Section 5.3.6).

The SafeBot framework is the most ambitious, and theoretically can be applied to both situations. But significant obstacles in available library and suitable components prevent its full implementation.

Table 1 summarizes available agent techniques.

Table 1, Summary of Agent Techniques

Technique	*Connect / Partition*	*Additional Feature*
Gateway Agent	Connect	General
Secure Access Wrapper	Connect	Multi MLS
NRL Pump	Connect	Multi level in Single MLS
MLS METEOR	Partition	Workflow, use NRL Pump
Workflow Partition	Partition	Workflow, use neutral agent
JIF/Split	Partition	Source code, use data/control transfer
SafeBot	Both	Ambitious

5.4 Meta Object Protocols

This section surveys another wrapper like mechanism, meta-object protocols, which uses entities of a special type (meta-objects) and a wrapping mechanism (object methods interception) to augment base entities (objects).

Meta Object Protocols (MOP) come from the Object-Orientation technology. The Object-Orientation technology has become the dominant development paradigm in the past decade. Security is getting more and more attention in mainstream object-oriented programming languages [46]. However, standard security programming techniques have several drawbacks. Developers have to develop new security policies, and manually insert system calls enforcing these policies into proper places at the application [131]. This mixes the security processing with the general functionality, which hinders maintenance and evolution of both functionality and security. When components coming from different sources are combined together, it is difficult to reason about the composite security properties and to reconcile potentially conflicting security policies. After deployment, it is difficult to enact new security policies without major redevelopment effort. These drawbacks come from the intertwining of functionality and security.

To tackle these problems, two advanced separation of concerns technologies have been developed. They are Meta Object Protocol and aspect-oriented software development. This section studies security enhancement techniques that adopt Meta Object Protocol, the simpler of the two. Aspect-oriented security technologies, which are more powerful in composing concerns, are surveyed in Section 0.

Meta Object Protocols are tightly related with reflection. Both concepts have a wide theory foundation. Their use in security engineering can be described as follows. Reflection [79] is a technique for introspecting the implementation in a controlled process. A meta-level programming abstraction is provided to examine and change the underlying structure and behavior of a system. The abstraction with its manipulation is called reification. The meta-level abstraction is causally connected to the underlying system, so that any change at the meta-level will be reflected back to the underlying system. Meta Object Protocol (MOP) [73] is an Object-Orientation reification, where the abstractions in the meta level are expressed as a set of meta-objects. The execution of Meta Object Protocol is summarized in Figure 16. When one object sends a message to another object (the solid lines in Figure 16), the message will not reach the destination object directly. Instead, the connecting mechanism redirects the message to a meta-object that is associated with the destination object (dashed line 1 in Figure 16). After some pre processing by the meta-object, the message is eventually routed to the destination object (dashed line 2 in Figure 16). Similarly, if there is a response from the destination object to the original source object, before the response can reach back the source object (dashed line 4 in Figure 16), it goes back to the meta-object first (dashed line 3 in Figure 16), probably for some additional post processing. The rest of this section describes how this general pattern is used in security engineering.

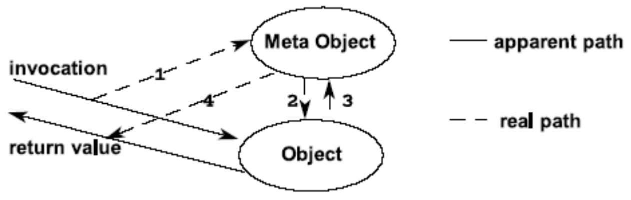


Figure 16, Meta Object Protocol, from [131]

5.4.1 *Actor*

The discussion of meta-object protocols begins with the Actor model. The Actor model is a general and flexible model of concurrent and distributed computation [2]. Each actor has a unique name and an associated behavior. This behavior describes the states of the actor and how the actor manipulates them. Actors communicate with each other by sending messages asynchronously. Each actor serially processes messages it receives. When processing a message, an actor can perform one of three actions: send messages asynchronously, create a new actor with a specified behavior, and become ready for next message.

The Actor model is a general model, so it can be used to model many computation systems. It also has a formal semantics defining how a system of actors can evolve when they send messages to each other.

To support extra-functionality, such as security, the original Actor model is extended with a meta layer [6]. In such a model, components at the base layer handle the functionality of the system, and components at the meta layer address extra-functionality, such as security, performance, and coordination. Additional meta layers can be stacked further onto existing meta layers to provide more extra functionality. For example, the composition of security can be handled in a meta-meta layer.

In an Actor system, there are three types of events: “message sent” event, “actor created” event, and “next message requested” event, corresponding to the three actions that an actor can perform. Events have a causal relationship among them, and they are atomic. To connect the base layer and the meta layer, an event is sent from the base layer to the meta layer whenever any event happens. This event is a meta-message, containing relevant information such as the sender of the original message, the receiver, and the content. After an actor at the meta layer (named a meta actor henceforth) receives the message, it can process the message to achieve the desired extra-functionality. During the meta processing, an actor at the meta layer can send notifications back to actors at the base layer, either unblocking them from the current sending operation, delivering the message, or returning a newly created actor. The process is illustrated in Figure 17.

The Actor model with a meta layer is used to enhance the data secrecy of the system [3]. A pair of meta actors is inserted between a sending actor and a receiving actor. One meta actor, the Encryptor, listens to the messages from the sending actor. For every message the sender sends, the encryptor encrypts the message and delivers it to the original receiving base actor. The other meta actor, the Decryptor, listens to the “next message requested” event from the receiving actor. After receiving the event, the

Decryptor receives the encrypted message before the receiving actor. It then decrypts the message and delivers it to the receiving actor. Because of the generality and flexibility of the meta-layer framework, the outlined security can be imposed on existing actors with minimal effort.

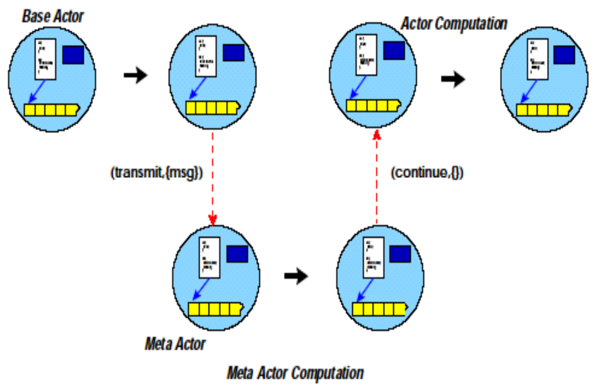


Figure 17, Meta processing of the Actor model, [3]

Because the Actor model provides a formal semantics for communicating distributed systems [2], it can be used to model and verify secure authentication protocols. An approach uses the Actor model to model the behaviors of not only the communicating parties but also the medium (called routers) and adversaries as actors [3]. After modeling the message exchanges between the parties, verifying the security protocol depends on showing that the parties can achieve the desired end result even in the presence of the adversaries. Currently the verification procedure is still mostly manual.

The actor model supports meta-object protocols naturally. However, its full implementation and application for security still waits for further investigation.

5.4.2 Security Meta Object

Security Meta Object [106, 107] is an early approach using meta-objects to enforce proper access controls.

In standard Object-Orientation technology, a reference to an object can be considered a capability (see Section 2.1) that enables access to the full functionality of that object. If a malicious object holds a reference to a high-privilege object, it can invoke that object to perform a dangerous activity.

A security meta-object is a meta-object that holds a reference to an object and enforces proper access control before the original object can be invoked. When a function call is made through the meta-object, the object checks whether the call is allowed. If the call can be dangerous, the security meta-object rejects the call and throws an exception. Otherwise, the call progresses as a normal function call.

To make security meta-objects effective, referencing a privileged object must go through a meta-object. This is the non-bypassability property of the reference monitor [5]. Otherwise, a malicious user can try to get a raw reference to the privileged object and bypass the access control enforced by meta-objects. Thus, a meta-object should also monitor the creation of any new outgoing reference for its underlying object and attach itself to the new reference.

Similarly, since an incoming reference may point to a malicious object, an invocation on that reference can result in security breach. A security meta-object solves this problem by reversely attaching itself to the incoming reference and enforcing proper check before invocations through the incoming reference.

The basic security meta-object approach is extended in [107]. Each meta-object is assigned a role representing the principal for which the meta-object acts. As in the role-based access control model [113], the role makes the access control policies more explicit and natural. Another extension is forming a domain for a principal and the detaching security meta-objects when they act in the home domain of the protected raw objects. This reduces security overhead. The problem of proper attachment and detachment of security meta-objects when entering and leaving domains has also been formalized and studied [107].

The Security Meta Object approach uses meta-objects to control access and attach meta-objects to all possible references. While using a separate meta-object to enforce security is attractive, attaching a meta-object to every possible reference and preventing all inappropriate access at all possible paths is difficult, giving the complexity of control flows and data flows in a program [42]. If a single reference is returned without a properly attached security meta-object, there exists a chance of security breach. Probably as a result of this, the Security Meta Object approach has not been fully implemented in its current form.

5.4.3 Types of Java Meta Object Protocol

Since Java has quickly become a mainstream language and it provides built-in support for security and reflection, several attempts have been made to use a Meta Object Protocol approach in Java to implement security.

Caromel and Vayssiere classifies Java MOP efforts into four categories, depending on when meta-level code is executed [21]: compile-time, load-time, VM-based run-time, and proxy-based run-time. They have studied the impact on security permission sets by different types of meta-level code. The solid lines in Figure 18 depict when meta-objects are in use.

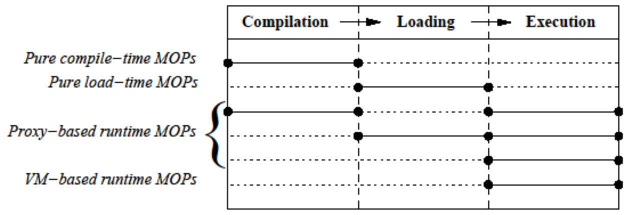


Figure 18, Types of Java MOP, from [20]

In the compile-time MOP approach, the source code is manipulated and translated to generate new source code for meta level classes. These classes are compiled together with the original classes. As a result, new permissions for these classes are needed to execute the translated program.

In the load-time MOP approach, the meta-level code begins execution at class loading time. It might end at load time (pure load time), after modification of the byte code. The meta-level code can also span into run time, where meta-objects created at load time are used to implement run-time meta-object behavior. The load time approach operates on bytecode, instead of source code. As a result, it cannot change the classes already loaded through the non-MOP class loader. Permissions for bytecode manipulation are required in addition to the normal operation permissions.

The VM-based run-time MOP approach uses a non-standard Java Virtual Machine to execute a program. The virtual machine can be either a modification of an original one or an extension to a standard VM with native library. The VM-based approach does not impose additional permissions, because all meta-objects are in the trusted computing base, along with the virtual machine. The virtual machine must be properly designed, implemented, and verified. Otherwise, malicious Java code may breach the architecture. This approach was only taken by early MOP efforts.

The proxy-based run-time MOP approach operates within an environment of a standard virtual machine. It is not as powerful as the VM-based approach because some events are not visible or controllable to it, but it can coexist with the standard environment and benefit from the engineering efforts invested on that environment. The approach uses a meta-object as a proxy of a base object. The meta-object exposes the same interface as the base object, intercept method calls for the base object, and perform additional processing around these method calls. The meta-objects require additional permissions to execute.

5.4.4 A Proxy-based Run-Time MOP

Because proxy-based run-time MOP achieves balance between expressive power and coexistence with accepted standards, some recent MOP efforts have taken this route. In [20], a simple proxy-based MOP was developed for Java. The approach aims at minimizing change of existing code when using meta-objects.

The MOP system intercepts two types of events during run-time: method invocations and instance creations. A meta-level object is created when a special instance creation function is invoked. The created meta-level object handles future method invocations.

The MOP system associates a base class with its meta class. The base class implements a marker interface and stores the meta class name as a static class member. When an instance of the base class is instantiated, the marker interface of the class notifies the MOP system for meta processing. The MOP system then retrieves the meta class name and creates a meta-object for the base object.

Each meta class implements an interface MetaObject. The interface contains one method MethodCall. This method implements the standard meta processing for the base class. The MOP system provides necessary information to this method. The meta classes can be organized into their own hierarchy independent of the base class hierarchy.

To meet the type compatibility requirements of the base class, a stub class is created for each base class during run time. The stub class inherits from the base class. Each inherited method of the stub class packages available information and invokes the MethodCall on the meta class. Transparently creating the stub class enables both meta processing and independent development of base classes and meta classes. This helps relieve one of the inheritance problems identified in [129], where the meta class has to replicate all implemented interfaces of the base class but still cannot support casting the meta class to super classes of the base class. Another problem identified in [129], the meta constraint problem, where a derived base class might be bound to a meta class significantly different than the meta class bound to the super base class, still remains a challenge.

To avoid introducing any further requirements for security permission by the meta processing, two techniques are used. First, before entering the meta processing, the current security context is captured. After the meta processing, the captured security context is restored before returning to normal processing. Second, the complete meta object protocol subsystem is granted full permission, essentially being put into the trusted computing base.

This approach provides a practical solution to add security capability onto base functionality. However, some problems should be further investigated. As pointed out in Section 5.4.2, to prevent security breach, each possible reference to a base object should be made through the meta-object. This approach does not address this problem, relying completely on calling the special instance creation function. This approach also grants full permissions to the meta object protocol subsystem, enlarging the trusted computing base. Further investigation on granting less privilege to the subsystem is worthwhile. Finally, the binding between a base object and a meta-object is achieved through programmatic declaration of the base class, which is different from binding through a separate file. More experimental results are needed for what will be a best binding mechanism.

5.4.5 Kava

Kava is a load-time MOP for Java [129, 131]. It uses a bytecode toolkit to rewrite the bytecode during class loading time. It uses behavior reflection to inspect the bytecode without resorting to source code. Kava does not blindly rewrite bytecode. It only injects meta-objects when they are necessary for implementing the policy, minimizing the performance impact. Kava also rewrites the bytecode in a type-safe manner, limiting potential problems brought by binary code operation.

Kava provides more capability than the simple MOP described in Section 5.4.5 [20]. In addition to method invocation and instance construction, Kava also provides control over field access and exception handling. For method invocations, Kava differentiates between two cases. The first case is called method execution, where the execution of a method of a base class is augmented. All calls on the method are automatically handled through the meta-object. The second case is called method invocation, where all invocations of a method are augmented to go through the meta-object. In this case, only invocations on classes already loaded can be augmented, but the classes themselves cannot be augmented. The two cases are named callee-side translation and caller-side translation in [21], respectively.

Instead of specifying the binding between base objects and meta-objects in the source code, as adopted in the simple MOP [20], Kava uses a separate specification file declaring the binding. The specification file also supports meta-objects parameter to further enhance flexibility. A separate binding file taking effect at binding time increases flexibility, because a developer can develop new security features and bind them with the base functionality without touching the original source program. A separate file is also necessary in the case of bytecode rewriting, because no source code is available to inject meta-objects.

Kava's architecture is shown in Figure 19. The shaded parts are provided by Kava. The figure clearly illustrates Kava's use of byte code rewriting during class loading time and its utilization of a separate binding file.

A novel feature in Kava is that it implements stronger non-bypassability. Its secure class loader employs standard Java security mechanism, monitors other class loaders and brings classes loaded by those loaders under Kava's control. For an application-level class loader, Kava monitors its activity and uses bytecode rewriting of method execution to inject Kava's control. For a system class loader, Kava cannot rewrite the byte code for method execution, but it can rewrite places of method invocation to enable Kava's control. Also, since Kava adds hooks to bytecode directly, the separate proxy problem identified in [106] (see also Section 5.4.4), where a base object might be accessed without the provision of a meta-object, can be greatly reduced. Bytecode rewriting effectively merges the two objects into one binary entity [129].

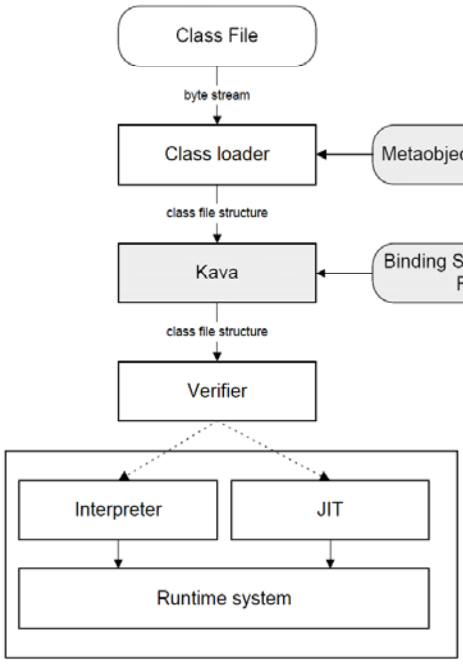


Figure 19, Kava, from [131]

Kava can support many types of security policies. It can support the standard access control policy, where the access to a resource is granted or rejected based on principals. It can support a resource-consumption access control policy, where the access to a resource is granted to a principal, but only to the extent that the resource is used within the a specified limit, such as the amount of bytes that is written over a network connection [131].

Kava can also support a complex integrity model like the Clark-Wilson model [127] (see Section 2.1). The constrained data items and unconstrained data items are modeled as Kava fields, and the transformation procedures and integration validation procedures are modeled as Kava methods. Principals are authenticated in the usual manner. Kava monitors field access on data items and allows only trusted transformation procedures to perform these accesses. Validation procedures are invoked after modification methods. Audit logs are written after any access methods. Since the Clark-Wilson model is an abstract model that embodies abstract concepts and operations, it is appropriate to use a meta-level mechanism to enforce the rules stipulated by the model.

In summary, Kava is a powerful and flexible MOP implementation. It coexists harmoniously with current Java infrastructure and applications. It can implement several types of security models. Moreover, its enforcement of security policies can be composed with base functionality flexibly.

5.4.6 Discussions

Meta Object Protocols can separate the security concern from the general functionality. As a result, these two aspects can be developed independently by domain experts each with the required expertise and appropriate techniques. Combining security and functionality can be performed at the deployment and customized to the special needs of the site. The evolution of functionality, security, and their integration is easier than the case when they are intertwined.

The Meta Object Protocol approach can be used to implement various kinds of security policies, as demonstrated in [127, 131]. Some arguments hold that it is superior to a standard container-based approach where only access control policies can be enforced [130]. What differentiates these two approaches more is probably not what policies they can enforce [115], but how flexible the enforcement can be. Meta Object Protocol shows promise in this aspect, but this should be investigated further under a comprehensive perspective with more experiments.

The Meta Object Protocol approach does have some disadvantages. The performance of a reflective based system can be unsatisfactory [131]. A reflective based approach can make reasoning about the system behaviors more difficult, because the possibility opened by a dynamic reflective execution is not analyzable by static techniques. However, this problem can be mitigated if the use of reflection by the base program is limited to a certain extent. A more serious problem is the challenge on comprehensive security bought by the extra MOP subsystem. Since the subsystem is quite powerful, abusing it can lead to severe damage. Current approaches put the subsystem into the trusted computing base [5] and make the subsystem part of the reference monitor. This can only be justified and trusted only after careful design, implementation, and strict verification of the subsystem [128] [132].

While many commercial platforms have not supported easy extensions to utilize the Meta Object Protocol approach, it is expected that, with the evolution of the technologies, MOP will be incorporated into these platforms to achieve “reflection at large” [133].

From the viewpoint of separating of concerns, the Meta Object Protocol approach still has limitations. Compared to aspect technologies discussed in Section 0, this approach uses the same language for implementing both the base objects and the meta-objects, where an aspect technology can use a more declarative language than the base language, and support security more effectively. The binding mechanism provided by this approach is generally less powerful and flexible than the weaving language implemented by aspect technologies. The two approaches can be combined, where meta object protocols can provide the necessary infrastructure to support a powerful aspect weaving [128].

5.5 Component Specifications

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.

5.5.1 *Computer Security Contract*

Computer Security Contract (CSC) addresses how to disclose the security property of a component to others [68, 69]. 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, 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.

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 20.

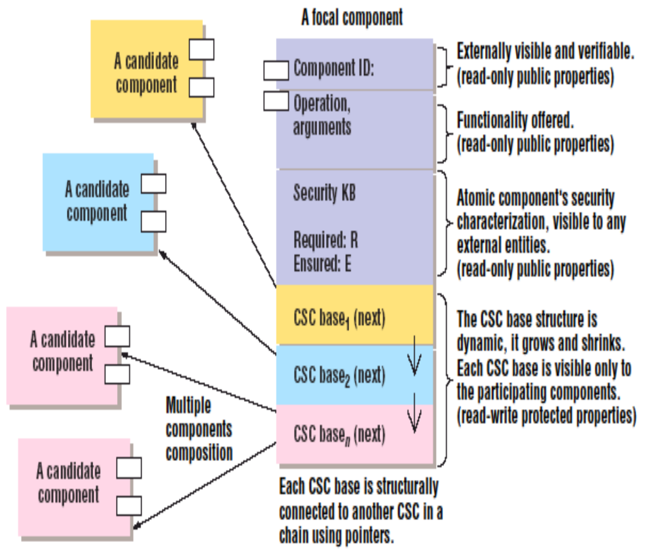

A focal component
A candidate component
A candidate component
A candidate component
A candidate component
Component ID:
Operation, arguments
Security KB
Required: R
Ensured: E
CSC base$_1$ (next)
CSC base$_2$ (next)
CSC base$_n$ (next)
Externally visible and verifiable. (read-only public properties)
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 20, Active Interface, from [69]

The logic-based contract is expressed with a Prolog-like form of logic programming [70]. 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 [18]. 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. First, 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. Second, the current composition mechanism is still very simple, mirroring a functional 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 [71]. Third, 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 (see Section 5.7.1). Other forms of composition might choose different places to store security contracts.

5.5.2 cTLA Contract

Hermann proposes a more elaborate component specification to describe and verify security properties of component-based systems [53]. Instead of the simple first order predicate logic used in the Computer Security Contract, a compositional extension to the Temporal Logic of Actions (TLA)[75], 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 5.1.1). 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 [115].

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 [69] (see Section 5.5.1), 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 [32, 123]. 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 experts.

5.5.3 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, while other implicit forms utilizing other description mechanisms are discussed in following sections. 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 underspecification of the components or implicit assumptions made by the components. Specifications of components should be complete so no undesirable properties will emerge during composition [56, 137]. 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 [29] 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 [43]. 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.

5.6 Composition Framework

A composition framework provides base components and composition mechanisms for composing secure application modularly. Compared to other techniques, a framework provides an explicit repository of components and connection mechanisms. This section surveys some general composition frameworks proposed in literature.

5.6.1 Infrastructure for Composability at Runtime of Internet Services

Infrastructure for Composability at Runtime of Internet Services (ICARIS) [23] is an environment that permits dynamic composition of services to form composite services. Three strategies are outlined for constructing new services. The first one composes the composite service as a virtual interface for its constituent service. The second strategy constructs a new container containing the constituent services. The last strategy extracts related components and re-assembles them into a composite service.

The framework is used to augment a client and server so that they can communicate securely using the cryptography technology [23]. After deciding the composite system needs a symmetric encryption of the application data and an asymmetric encryption of the symmetric encryption key, those components are selected, composed and deployed to the client and the server. The composition should be careful about the correct order, because the corresponding decryption components should used in a reverse order on the server than that of the encryption components on the client.

Composing components dynamically and securely poses greater challenges than doing it statically [14]. The ICARIS approach does not provide convincing answers to many challenges. It is not clear how the components are described, how new requirements are introduced, how the composition is negotiated and decided to meet the requirements, and how the correct order can be persevered automatically.

5.6.2 Composable Replaceable Security Service

Composable Replaceable Security Service (CRSS) [32] is a framework to support fault-tolerant and composable security services.

The CRSS framework classifies services into high-level services and low-level services. The high-level services include a connection service (adding confidentiality and integrity to a connection between two applications), a transaction/exchange service (providing security enhancement to data in a single transaction), a retrieval and storage service (allowing secure retrieval and storage of named objects), a remote execution service (executing mobile code), and an authentication service (associating active entities with their identities, authorizations, certificates, and credentials).

Low-level services include a cryptographic service, a database service (a highly secure repository for critical data), a key/credential/certificate service, a trust/authorization service, and an audit service.

The CRSS framework has four components: a provider registry, a provider manager, a provider switch, and a survivability manager. The provider registry keeps information about each available security service provider. The provider manager selects providers to fulfill requests from applications. The manager can choose different providers as long as they all provide the same service. The provider switch facilitates transparent execution of remote providers. If a service can only be accomplished by a provider not locally available, the manager asks the switch to launch the execution and returns the result back when the execution is complete. Finally, a survivability manager enhances survivability by using several potentially different implementations of the same service.

The composition of services in the CRSS framework is rudimentary. It is limited to selecting compatible service providers when fulfilling an application request. Even the straightforward issue of composing high-level services from available low-level services is not addressed by the current CRSS framework.

5.6.3 Intrusion Detection Inter-component Adaptation Negotiation

Intrusion Detection Inter-component Adaptation Negotiation (IDIAN) [33] is a system to support dynamic communication of intrusion detection components. The dynamic communication can be used to introduce new components or new capability of old components into the comprehensive intrusion detection system, and it can also be used to balance load among available components.

The intrusion detection components operate under the Common Intrusion Detection Framework, which contains monitors to monitor events, analyzers to analyze information, responsers to respond to actual intrusions, and a database to store information. These comp0nents communicate with each other by passing General Intrusion Detection Objects, which describe events that occurred in the system, such as possible attacks. Filters can be applied on these objects to decide which objects a component will receive.

The components form a producer and consumer agreement between them after participating in a negotiation protocol. The protocol specifies how a component advertises its capability, how a component proposes, how a component counter-proposes, how a component rejects or cancels a proposal, and how a component accepts a proposal and seals the agreement. The protocol is specified in a formal language. A component can behave as both a consumer and a producer when it engages in multiple agreements.

In summary, this approach sketches how multiple secure components coordinate to provide even more security, participating in a common, formally specified protocol and exchanging information through a well-defined format. Due to its limited objective in application domain, it does not support composition using dynamic protocols and non-intrusion detection related information.

5.6.4 Partitionable Services Framework

Partitionable Services Framework (PSF) [58] is a framework that supports dynamic assembly and deployment of components to adapt to heterogeneous environments where each administrative domain maintains its own security policies.

The PSF framework has four elements: a declarative specification of the application and its environment, a monitoring module, a planning module, and a deployment infrastructure. The monitoring module provides dynamic information. Using this information and specification for the components, the application and the environment, the planning module produces a sequence of component deployment plans. The infrastructure implements these deployment plans.

A view is an object that either implements a subset of the functionality of the original object, or works with a subset of the data of the original object. A view provides greater flexibility under application and network constraints, and it enables a finer level granularity in access control.

The access control model in PSF is a decentralized Role Based Access Control model. The model encodes properties of applications and resources into credentials. To make an access control decision, the model seeks an answer to a question of whether X has the role of Y. The decentralized nature of the model lies in that the model permits use of names local to each autonomous domain and depends on role mapping delegations to translate local names.

A view is the atom of operation and access control. It is generated from component specifications and credentials available at the generation time, so it can fit the dynamic security constraints presented. These credentials enable the views to operate across various autonomous domains.

One important part of the monitoring module is the SwitchBoard. The SwitchBoard establishes a secure, authenticated, and continuously authorized and monitored connection between two components. During its operation, when the SwitchBoard detects a change in credentials of two components, it can take actions, such as asking new credentials, to keep the secure connection alive.

In summary, PSF uses a credential-based formalism to express security properties, generates views dynamically from component specifications to accommodate constraints presented at component composition time, and provides a monitoring mechanism to support the continuous secure interoperation between components.

5.6.5 Discussion

Table 2 summarizes current composition frameworks.

Table 2, Summary of Composition Frameworks

Technique	*Component*	*Composition*	*Other feature*
ICARIS	General	Virtual Interface, New Container, Re-Assembly	
CRSS	Low-level services, High-level services	Selection of service providers	Remote provider, Survivability
IDIAN	Intrusion Detection Components	Events exchange, Producer-Consumer negotiation	Formally described negotiation protocol
PSF	View with declarative specification	Dynamic composition	Monitoring module for secure session

The notion of composition framework is appealing. This approach constructs a secure application from a collection of available components, using appropriate composition mechanisms, to achieve desired security with assurance. However, the state of art in composing securely is far behind what has been accomplished in general functional composition. There is no consensus in what secure components are or should be. Current composition mechanisms utilized are insufficient, either lacking dynamic features or failing to address special security requirements. Finally, there is no mechanism assuring the result of composition.

5.7 Aspect

Aspect technology can be considered as a special composition framework to compose secure and modular applications.

Aspect-Oriented Programming (AOP), exemplified by AspectJ, is an extension to Object-Oriented Programming (OOP) to address the cross cutting concern problem more effectively [72]. It cleanly captures each of these crosscutting concerns in one self-contained aspect. Each *aspect* contains two types of information. One is called advice, which defines how the crosscutting concern should be implemented. The other is called pointcuts, which are places where the advice should be applied to the OOP base code. A special tool, the weaver, is used to combine (weave) the aspect and the base code together. The system resulted from this weaving process will contain appropriate links inserted in the base code. These links are defined by the pointcuts of the aspect, and they reference the advices of the aspect.

Because AOP separates concerns explicitly and models them directly, it has received much interest since its inception and has been extended to other phases of software development. This section surveys how the security aspect is addressed by various aspect technologies. Section 5.7.1 discusses an alternative AOP system, A-TOS/JAC. How aspect technologies can be applied to security issues in traditional procedural language environments and middleware settings are discussed in Section 5.7.2 and Section 5.7.3, respectively. Section 5.7.4 turns to advanced composition techniques provided by Lasagne. Section 5.7.5 discusses Component Virtual Machine. Section 5.7.6 investigates whether the aspect technology can be applied to early security design stage.

5.7.1 A-TOS/JAC

A-TOS [102] is an aspect-oriented reflexive middleware for distributed environment. Its core concept is an aspect component implementing global transversal properties including security. Aspect components are used to achieve separation of concerns in a distributed environment.

Aspect components utilize two approaches to achieve separation of concerns, meta-objects and meta-classes. Meta objects provide adaptability and distribution. At run-time when objects exchange messages, meta-objects can intercept the message and perform additional processing before and after message delivery. The order of the extra processing can be flexibly specified.

Meta classes enable powerful reflexive features. The class definitions are readable and writable, so wrapping classes that provide extra-functionality, such as security, can be inserted into the original classes at run time.

Each aspect component class specifies what it does and how it should be applied to the base classes. Under A-TOS, security can be handled by invoking appropriate operations before regular functions are performed.

These ideas have been evolved into Java Aspect Components (JAC) [103]. This approach can be considered as using Meta Object Protocol (Section 5.4) to implement aspect technology (see Section 5.4.6).

A-TOS/JAC demonstrates that the aspect technology can solve some simple security problems by collecting functions handling security concerns into a single aspect and invoking these functions with the standard aspect-oriented programming facility.

5.7.2 Aspect-Oriented Security Framework

The Aspect-Oriented Security Framework (AOSF) [117] is a source code translation framework that applies aspect technology to address the security concern. Its current implementation works on programs written in C.

Aspects in this framework are considered as code transformation templates, specifying where, why, and how the code should be translated. The aspects are defined in an aspect language, which is a superset of the application language. The framework works along with the normal build process. The application and associated aspects are pre-processed. The weaver then weaves the pre-processed code together into woven pre-processed code. Finally, the woven code is compiled and linked into a complete application.

The framework has been initially used to solve implementation security issues, such as buffer overruns and time-of-check-versus-time-of-use bugs. Aspects in these situations are simple one-to-one syntactic transformations, needing only local context information. Architectural security issues such as communication channels and event ordering will require aspects embodying more logic, context, and customization.

In summary, AOSF provides a simple approach that uses the notion of aspect to solve some common security implementation problems. Its applicability to more complex situations has not been proven yet.

5.7.3 DADO

DADO [135] standards for Distributed Adaptlets for Distributed Objects. It extends support of cross cutting concerns to a distributed, heterogeneous environment. Most aspect technologies are based on a single language, and they operate on a single computer. DADO is a middleware supporting aspect technology. It uses a language to model the concerns in the system, but the implementations of the model can be achieved through different languages, and they can reside on different computers.

DADO extends CORBA, the accepted distributed object-oriented architecture standard. In DADO, a client and a server form a pair of adaptlets. An adaptlet is described using an extended interface definition language (IDL). In additional to standard method declarations, the interface of an adaptlet can have two more kinds of methods. The first kind of methods is called advice. An advice method implements concerns such as security. It is executed every time some other base methods are executed, subject to a binding specification. The second kind of methods is called request. A request is an asynchronous message sent by a client or a server during the execution of an advice method. The client and the server in an adaptlet can use the keyword “that” to refer to the other side of the pair, similar to the use of the keyword “this” in standard object-oriented programming.

DADO extends the AspectJ [72] pointcut language to express pointcuts in a client or server adaptlet. A compiler compiles an adaptlet IDL description, which specifies all IDL level events needing adaptlet processing and the information needed for the processing. To trigger the appropriate processing logic when certain events happen, DADO provides a variety of strategies. These strategies maintain the language independence of middleware. Both source code instrumentation and binary-only instrumentation are used so that the trigger process is transparent to the application and security development. DADO packages needed information into the per-invocation service context, and sends returned information back along with the standard CORBA message flow.

DADO supports a limited form of dynamic service discovery and composition. It encodes available services in a server reference. When a client retrieves the reference, it decodes and discovers the available services from the server. The server pointcuts are assumed mostly static. When the available services change, an exception is thrown to the client, and the client is expected to retrieve a new reference.

DADO stresses type-safety and uses strong typing in compilation and marshalling. This is not a problem in a single language environment and it is sometimes ignored by other component composition work.

DADO can be used to support security concerns. Like other aspect technologies, the basic mechanism is the injection of proper security method calls along the message flow of base functionality. Due to classifying methods into advice and request, the communication pattern is richer and clearer. For example, to support the authentication of a client, the client can execute the `contactAuthentic` advice before it contacts the server, and the server can execute the `check` advice before it allows the client to proceed. During the execution of `contactAuthentic`, the client can issue a `register` request to the server, so the following check by the server allows the client to proceed.

DADO adopts standard aspect technology in a distributed and heterogeneous environment. As an extension to CORBA, DADO uses generated stubs/skeletons to support heterogeneity. This CORBA influence is novel. It makes DADO suitable for situations where traditional single language-based aspect technologies are insufficient. However, DADO is also limited by its CORBA origin. First, it is heavily dependent on client/server architecture. Its basic entity, adaptlet, is a pair of client and server. This limits its use in other situations where the roles of a client and a server can change or the roles do not exist at all. Second, its composition mechanism is static. Its current form relies on compiling IDL descriptions. Relying on IDLs imposes challenges for providing the flexibility similar to Meta Object Protocols (Section 5.4) and dynamic context-sensitive compositions (Section 5.7.4).

5.7.4 Lasagne

When combining functional components together with other non-functional aspects, the combination should be modular so that only necessary components and aspects specific to the requirements are integrated. All components and aspects thus integrated should interact in a consistent manner, exposing no potentially incompatible behavior. The integration should also be dynamic and adaptive to evolving situations.

Composition techniques based on simple class wrappers have two limitations. First, they suffer the identify management problem, where the wrapper has to maintain proper reference and encapsulation over the wrapped object (see Section 5.4.2). Second, the wrapping is static. Only wrappers can access wrapped classes, and they cannot adapt the access to dynamic context by freely integrating with other wrappers.

Lasagne is an integration solution that separates the functionality of wrappers from how the wrappers are combined and used [126]. It has four basic concepts. First, Lasagne uses a component identity to unite and hide the component instance and the wrapper around it. Second, it introduces the concept of extension, which is the context or service where components are composed. Third, it introduces a composition policy external to the components and extensions. The composition policy specifies how the components and extensions should be composed together in a collaboration. Finally, an interceptor can intercept message exchanges, inspect contextual properties, and dynamically modify compositions to achieve system wide consistency.

A wrapper, illustrated below in Figure 21, can be a decorator wrapper performing additional pre and post processing or a role wrapper extending the original interface. Each component provides a generic dispatch mechanism called variation points to support dynamic composition of wrappers. During deployment, an extension can specify how different wrappers are to be weaved together by delineating the method invocation sequence for the wrappers. At run time, the interceptor dynamically decides the combination of extensions based on requirements and contexts.

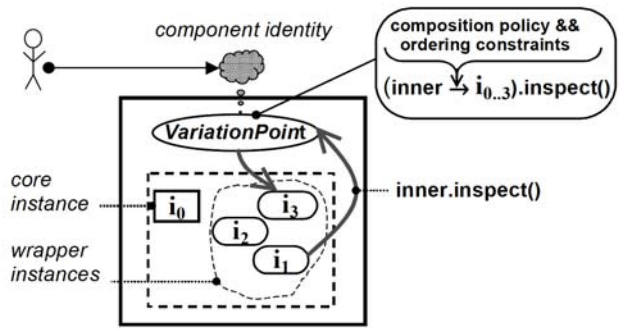


Figure 21, Wrapper in Lasagne, from [126]

Lasagne supports applying a consistent policy to a collaboration where several components are composed together. This can be used to enforce a dynamic security policy tailored for the collaboration. In [59], a dynamic monitor is created for each task to enforce unique policies coming from potentially conflicting sources. The context-specific dynamic composition, along with the merging of component identities, is illustrated in Figure 22, in comparison against traditional static composition.

In summary, Lasagne merges wrappers into united components, separates composition from encapsulation, and supports context-specific composition. It uses a powerful dispatching mechanism to support flexible composition.

5.7.5 Component Virtual Machine

Component Virtual Machine (CVM)[30] is another novel approach that treats security as a general aspect of software and leverages Meta Object Protocol in its solution. It tries to overcome limitations of current security technologies based on containers and aspects.

A component container provides infrastructure for component-based application, such as location, resolution, invocation, and transaction. The current generation of component containers, like those available in CORBA, COM+, .NET, and Enterprise JavaBeans, imposes several limitations on components developed for them: 1) The component designer choose the specific targeting environment, making it very difficult, if not impossible, for a component user to retarget a component to a different environment. 2) Components have to implement callback functions defined by the environment. 3) Partly because of the above two issues, component users do not have enough flexibility in changing components. 4) Most importantly, the security features of the environment are predefined, and component users cannot define new security services.

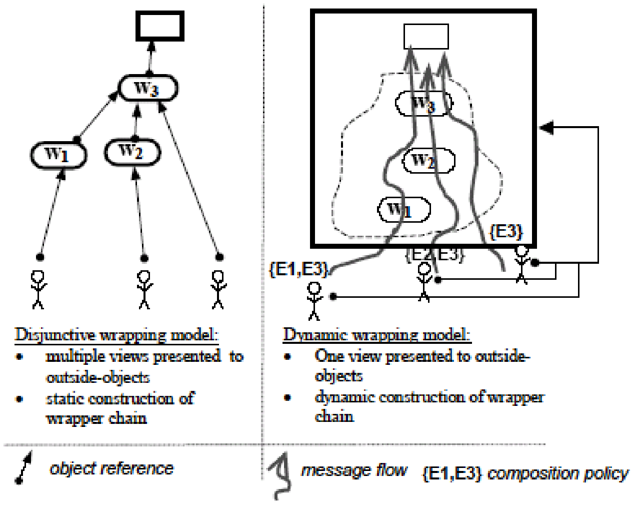

W3
W1
W2
{E3}
{E1,E3}
{E2,E3}
Disjunctive wrapping model:
multiple views presented to outside-objects
static construction of wrapper chain
Dynamic wrapping model:
One view presented to outside-objects
dynamic construction of wrapper chain
object reference
message flow
{E1,E3} composition policy

Figure 22, Composition of Lasagne, from [126]

Current aspect-oriented programming techniques do not provide flexibility. They operate on methods of source code. The focus is the transformation of code, not the interpretation of code. Decisions about the location and the content of those transformations are mostly made at compile time, leaving little flexibility for later time customization.

To tack these problems, Duclos et al. propose Component Virtual Machine (CVM), an approach combining the component container technology with aspect-oriented programming. The architecture is shown in Figure 23. They still adopt a container based on Enterprise JavaBeans, but eliminate the rigidity of services as much as possible. The container intercepts invocations on components so that all invocations are regulated by the container. Thus the container provides a virtual machine for components, from which the components receive services needed for their functionality. The AOP approach they adopt operates on the component level, instead of the method level used in traditional AOP. The actions of each aspect also affects the execution environment (the virtual machine), in addition to the component itself.

The Component Virtual Machine utilizes Meta Object Protocol (see Section 5.4). It provides mechanisms through which the component user can define new policies for component execution, so adding extra functionality to the base capability of the component is easy and flexible.

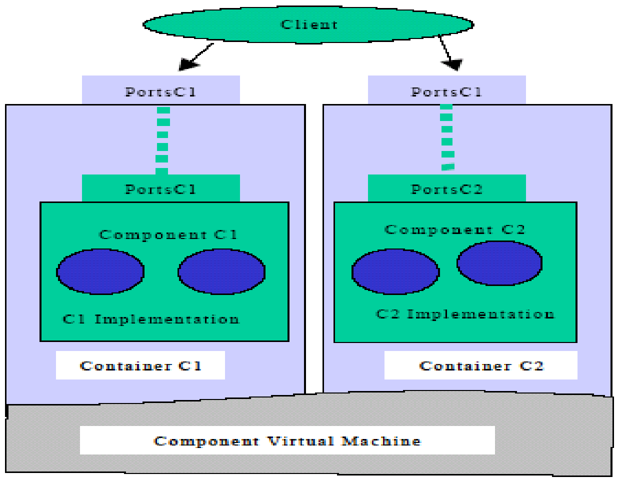


Figure 23, Component Virtual Machine, from [30]

Two languages, an Aspect Description Language and an Aspect User Language, are defined. The former allows the component designer to describe new aspects, specifying where the aspect should be applied, what actions should be executed when the aspect applies, and how these aspects can be generated and weaved. The latter permits the component user to express how the aspect should be applied and supply the context and information needed in applying the aspect. While as many callbacks as possible are implemented outside components, components still have to implement most of the callbacks because only the component designer has the full knowledge to correctly accomplish this task. Figure 24 describes the meta models for components and aspects.

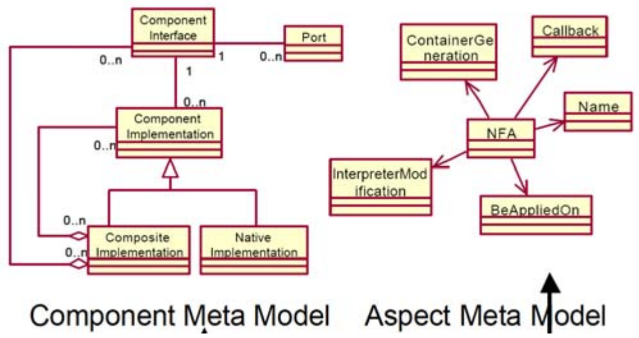


Figure 24, Meta Models of CVM, from [30]

An access controller can be applied as an aspect on a component so each invocation on the component will only succeed when the invoker presents satisfactory credentials. An aspect as well as its generation is defined with the Aspect Definition Language. Applying the aspect before each invocation of the component is described using the Aspect User Language. The Component Virtual Machine generates the necessary code and inserts it at appropriate places to enforce the access control policy.

Component Virtual Machine is a combination of Meta Object Protocol, the aspect technology, and container approaches. It shows great promise in handling security concern modularly and flexibly. While its intention to eliminate callbacks might not be fully fulfilled, its adoption of a deployment mechanism (the container/virtual machine) and use of aspect definitions by end users can improve the flexibility in security. More experiments are needed to evaluate its capability.

5.7.6 Feature Solution

Feature-Solution [26] graph is a graph that links requirements (features) to possible solutions. It captures design knowledge. The graph can be used in a design process named top-down composition. When following the general top-down decomposition approach to decompose the system from the most abstract level down to the most concrete level, at certain points where the decomposition can be achieved by reusing existing designs, those designs are integrated, so the system is at least partially composed out of available solutions, like the bottom-up composition approach.

When reusing existing solutions to satisfy a feature requirement, the solutions should provide variation points accommodating customizations for the new feature. In certain cases, more than one variation points are touched to accommodate a single feature, like security. For example, to add encrypted communication between a client and a server, not only is the client modified to include an encryption but the server is also modified to include a compatible decryption. This design approach is named Aspect-Oriented Programming at the Architectural Level, and is claimed usable as an effective approach to tackle the problem of adding cross cutting aspects like security into applications.

This approach raises interesting questions that should be answered before its claim can be fulfilled. First, larger and deeper knowledge about design should be captured in Feature-Solution graphs before its treatment of cross-cutting aspects can be evaluated with more assurance. Second, the issue of automation support should be explored. The current approach includes a significant portion of manual work. Finally, how well the approach suits security and what kind of security can be effectively treated remains to be seen.

5.7.7 Discussion

Separation of concerns is an important theme of software engineering. The aspect technology provides a natural mechanism to automate reasoning and constructing concerns and their separation. It shows promise for modular and secure software, as demonstrated by applications in previous sections. However, the current approaches are still limited to some simple situations such as access control and encryption/decryption. How the aspect technology can be extended to solve large-scale problems in a systematic fashion still remains a research question.

JAC (Section 5.7.1) provides an option for Aspect-Oriented Programming other than AspectJ. AOSF (Section 5.7.1) retrospectively applies the notion of aspect to traditional procedural languages which are still used for a large fraction of secure software development. DADO (Section 5.7.3) extends aspects from a single programming language to a heterogeneous environment using middleware technologies. All these efforts expand the applicability of aspect technology.

Lasagne (Section 5.7.3) proposes a different mechanism that combines separate concerns dynamically to suite context-specific needs. Component Virtual Machine (Section 5.7.5) uses a container to provide support for concerns. Both techniques open new possibility to introduce additional concerns into the base functionality.

The Feature Solution (Section 5.7.6) approach is still very rudimentary. How concerns can be expressed more explicitly during the analysis and design stages and how they can be effectively enforced in the implementation and deployment stages is still open for further research.

5.8 Architectural Approaches

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 5.8.1). It then turns to approaches without an explicit notion of connectors (Section 5.8.2, 5.8.3, and 5.8.4). The next discussion is about architectural models supporting explicit connectors (Section 5.8.5 and 5.8.6). The issue of architecture evolution is discussed in Section 5.8.7.

5.8.1 Object-Oriented Labeling

Like modeling software architecture with standard object-oriented notations [88], some design techniques extend object-orientated methodologies to support security. Herrmann introduces a methodology to analyze information flow security [52]. The theoretical foundation of the methodology is a decentralized labeling model. The meta-model used in the methodology is the Common Criteria [19]. To facilitate the adoption of the methodology, a tool based on graph rewrite system is also developed.

A label in the decentralized labeling model [97] 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 [19] defines a set of classes for concepts 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 [81], 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.

5.8.2 ASTER

Bidan and Issamy 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 [138] and composes a customized connector out of base connectors and system-provided connectors to connect the components and meet those requirements.

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 [121]. 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 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 5.5). The approach is supported by a configuration-based design environment. The approach 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.

5.8.3 System Architecture Model

System Architecture Model (SAM) is a methodology that can be used to model and analyze security of system architectures [27]. 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 [27]:

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 to 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 resulting larger and executable Petri net can be used to verify the consistency between constraint patterns at two levels.

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

The overall methodology is illustrated in Figure 25, 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.

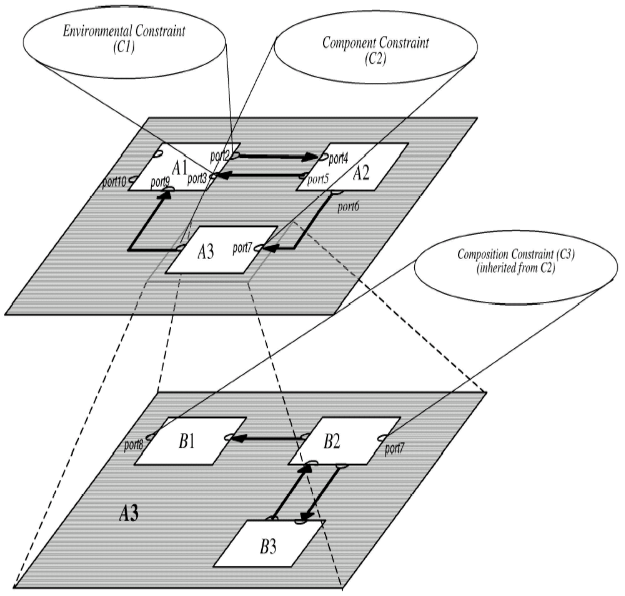

Environmental Constraint (C1)
Component Constraint (C2)
port2
port4
A1
A2
port10
port9
port3
port5
port6
A3
port7
Composition Constraint (C3) (inherited from C2)
port8
B1
B2
port7
A3
B3

Figure 25, System Architecture Model, from [27]

The SAM methodology is a top-down approach. It starts with the security requirement 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 5.1.1 and 5.1.2).

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 trade offs of the allocations 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 trade-off analysis becomes even more difficult.

Since the System Architecture Model is based on Petri nets, its notion of connector is different than a canonical one. 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.

5.8.4 Colored Petri Net

A special type of Petri Nets, Colored Petri Nets, is also used to analyze security [41]. A Colored Petri Net associates a type (its "color") with each place of a Petri Net. It also uses guards to specify conditions for firing a transition. Expressions can also be attached to transitions to describe further actions.

A software architecture containing components and connectors is mapped into places and transitions of Petri Nets. More information is captured through colors of places and guards and expressions on transitions. A Colored Petri Net is executable, so a simulator can be used to simulate the architecture and collect execution information.

An example given in [41] is a simple model for security of a network. The network is modeled as a transition. The weakness of cryptography, the importance of information, and the ease of wiretapping are modeled as expressions on the transition and used as parameters to calculate a value designating security on the network.

Other quantitative approaches construct a queuing model or a Markov Chain Model and have to use different models for different systems and different types of quality. Compared to those models, the Colored Petri Net approach can handle different types of quality of different systems in a uniform manner. However, this approach is not suitable for general security analysis. As pointed out in [41], the approach is best suited for qualities with the following properties: 1) The quality can be calculated using numerical factors 2) the numerical factors can be assigned to components and connectors, and 3) the calculation can be performed along the execution of the architecture. Reliability is a good example of such qualities. Security, on the other hand, does not belong to this category.

5.8.5 Connector Transformation

Given the importance of connectors in architectural development [89], constructing them effectively is of great importance. Handcrafting each connector can be very expensive. Existing connectors do not always provide all required qualities. Like composing general application using existing components, connector composition is becoming indispensable to software development. 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 [120].

The motivating problem of the approach is to add security property to a generic communication mechanism. In the example given in [120], 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 very expensive, and the result is not maintainable. A second possibility is to modify the generator generating stubs for the communication mechanisms so it 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 mechanism. The general transformational method could be applied again on the resulted connector when the mechanism needs to provide other qualities.

The transformation method is outlined in Figure 26, 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 parts, and w collects the formal specification describing the connector's proper behavior.

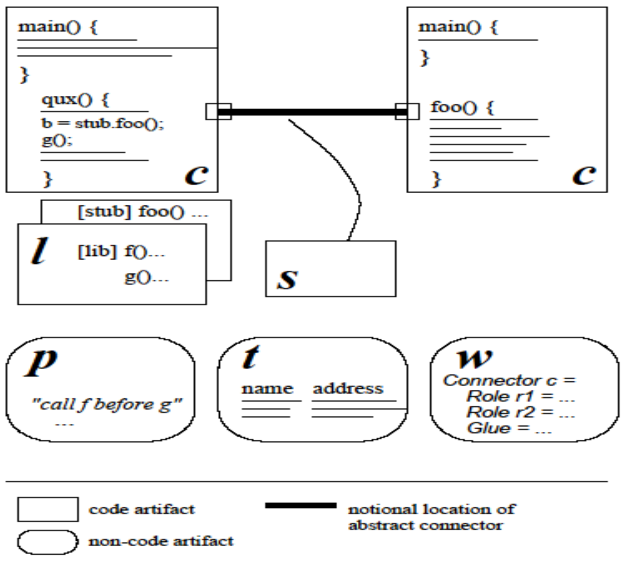


Figure 26, Connector Transformation, from [120]

They 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 aforementioned aspect methodologies (see Section 0) provide a general framework that can handle many different aspects, but not much support specific for security is provided. 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.

5.8.6 SADL

Architecture Proof. Secure Software Architecture [96] is one of the few approaches that directly deal with security at the architectural level. Based on the correct refinement approach presented in [95], 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 supporting property refinement. The approach is illustrated in Figure 27.

They use the approach to prove the Bell-LaPadula [9] 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) Put all three components into a single security level. 2) Put the application and the resource manager at different levels, connect them through a MLS 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.

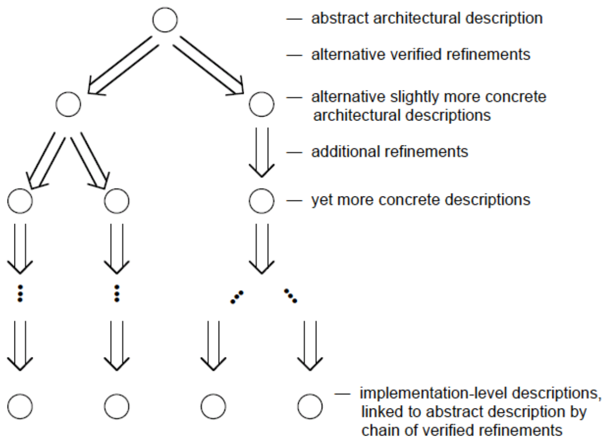

— abstract architectural description
— alternative verified refinements
— alternative slightly more concrete architectural descriptions
— additional refinements
— yet more concrete descriptions
— implementation-level descriptions, linked to abstract description by chain of verified refinements

Figure 27, Secure Software Architecture, from [44]

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 that assumes 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 [44]. 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 they have to introduce transformations that do not always preserve security, and they will check to assure that such transformations 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. 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 is 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, they 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 very effective in verifying the security of the architecture. This is also demonstrated in [27], 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 5.1.2). 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,.

They plan to use light weight formal approach, design a lot, specify some, and prove just a little [122]. This is more practical than a formal method that requires great efforts from methodology experts.

5.8.7 Law-Governed Architecture

Law-Governed Architecture [92] 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. First, it can bridge the gap between a descriptive architecture and the system, enabling reliable reasoning about the system. Second, 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 manger 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. Or the laws can 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. Or 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, among other aspects.

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. This issue remains an open research problem

5.8.8 Discussion

This section discusses several software architecture-related solutions for the modular secure software problem.

The simple extension of standard object-oriented notions with security information (Section 5.8.1) can be very 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 (Section 5.3.6), whose information flow security requirements on programs can derive from the secure object-orientated design models.

However, security should be addressed as early as possible. This naturally leads to an architecture-based approach. Simple extensions to module interconnection models (Section 5.8.2) do not provide a formalism rich enough to express and reason about architectural security concerns. Even models with a formal underpinning (Section 5.8.2) 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 5.8.5 and 5.8.6) 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 5.8.7). The model can serve as a basis to prevent the system from degenerating into insecure variants. This remains a big challenge for researchers.

6. CONCLUSION

The surveyed technologies are summarized in Table 3, using the framework developed in Section 4. Shades are used to separate categories of techniques from each other.

A rather coarse rating (fair, good and excellent) is given to each technique, based on subjective judgment of its expressiveness, applicability, flexibility, maturity, and potential for solving the modular security problem.

Table 3, Summary of Surveyed Techniques

Technology	*Security Model*	*Component Type*	*Connection Mechanism*	*Approach*	*Formalism & Tools*	*Rating*
Integrity Verification, like CSS [99, 100]	Access Control	Logic Formula	Refinement	Top-down	Logic + PVS	Good (Applicable, proof intensive)
Trace-based Information Flow, like SIF [84, 86], [80], [47]	Information Flow Security	Trace	Product, Cascade, Feedback	Bottom-up	Trace	Good (Theoretically appealing, few applications)
Process Algebra-based Information	Information Flow Security	Process	Parallel execution	Bottom-up	Process Algebra + Model Checking	Good (Theoretically appealing,

Technology	*Security Model*	*Component Type*	*Connection Mechanism*	*Approach*	*Formalism & Tools*	*Rating*
Flow, like SPA [36, 37, 39], [110]						few applications)
Application-level Wrapper, like [140]	Access Control	Application	Wrapped application	Top-down		Fair (Ad hoc)
Library function-level Wrapper, [8]	Access Control	Function Call	Function call interception	Bottom-up	Mediator	Excellent (Practical solution)
System Call-level Wrapper, Hypervisor [93, 94] and GSW [40]	Access Control	System Call	System call interception, Kernel Loadable Module	Bottom-up	Wrapper Definition Language	Excellent (Practical solution)
Gateway Agent [14]	Access Control		Gateway Agent	Bottom-up	Prolog-like knowledge base	Fair (Ad hoc)
SAW [24, 25]	Mandatory Access Control	Database	Secure Access Wrapper	Top-down	Lattice mapping and labeling graph	Excellent (Practical solution)
MLS METEOR [62, 63]	Mandatory Access Control	Single-level Workflow	Pump [65], Policy servers	Top-down	Design Environment	Good
Workflow Partition [7]	Information Flow Security	Conflict-free Workflow	Neutral Agent	Top-down	Workflow trust relationships	Good
JIF/Split [97]	Information Flow Security	Conflict-free subprogram		Top-down	Hosts trust relationships	Good
SafeBot [35]		Wrapper Agent	Knowledge base	Bottom-up	Ontology language, compiler, and library	Fair (Over Ambitious)
Actor [6]	Access Control	Actor	Meta level events	Top-down, Bottom-up		Good (Has potential)
Security Meta Object [106, 107]	Access Control	Object	Security Meta Objects	Top-down Bottom-up		Fair (Limited)
Simple MOP [107]	Access Control	Object	Compile time tagging + Stub class	Top-down Bottom-up	Tagged Java source	Good (Practical)
Kava [129, 131]	Access Control, Clark-Wilson	Object	Bytecode rewriting	Top-down Bottom-up	Kava Class Loader, Binding specification	Excellent (Practical, Flexible)
Computer Security Contract [68, 69]	Access Control	Component	Contract Negotiation	Bottom-up	Logic, Active Interface	Good (Has potential)
cTLA Contract [53]	Access Control	Logic Formula		Top-down	Temporal Logic	Good (Has potential)
ICARIS [23]				Bottom-up		Fair (Limited)
CRSS [32]		Low-level service	Select low-level services for high-level service	Bottom-up		Fair (Limited)

Technology	*Security Model*	*Component Type*	*Connection Mechanism*	*Approach*	*Formalism & Tools*	*Rating*
IDIAN [33]	Intrusion Detection	Intrusion Detection Component	Events, Negotiation	Bottom-up	Formally specified components and negotiation protocol	Good (Domain is limited)
PSF [58]	Role-based Access Control	Views generated from object	Dynamic composition, monitored connection	Top-down Bottom-up	Logic-support credential	Good
A-TOS/JAC [102] [103]	Access Control	Base + Aspect	Meta Object, Meta Class	Top-down, Bottom-up		Excellent (Practical)
AOSF [117]		Base + Aspect	Weave	Top-down	Weaver	Fair (Limited)
DADO [135]	Access Control	Adaptlet	Extended CORBA	Top-down Bottom-up	Extended IDL; service and request	Excellent (Applies to middleware)
Lasagne [126]	Access Control	Wrapped Component	Dynamic, context-specific composition; Dispatching	Top-down		Excellent (Powerful composition)
CVM [30]	Access Control	Deployable Component	Container-based interception; dynamic composition	Bottom-up	Aspect Description Language and Aspect User Language	Excellent (Very Flexible)
Object-Oriented Labeling [52]	Information Flow Security	Object		Top-down	Decentralized Labeling; Graph Rewrite	Good
ASTER [12]	Access Control	Component	Component selection	Bottom-up	Logic	Fair (Limited)
SAM [27]	Access Control	Petri net	Petri-net composition	Top-down	Petri net and Temporal Logic	Excellent (Practical approach)
Connector Transformation [120]	Secure Communication	Regular component	Transformed secure connector	Top-down	Transformations	Excellent (Has potential)
SADL [96]	Mandatory Access Control	Component	Security-preserving Transformation	Top-down	Logic, PVS	Good (Powerful, but proof intensive)

The following observations are drawn from the survey:

Foundations. Security is an extra-functional property. It is not always preserved by standard notion of refinement or composition. This implies that the assurance gained by formal proofs of a higher abstraction level cannot be necessarily transferred down to a lower abstraction level. This presents a big challenge to applying traditional abstraction and reasoning mechanisms for security design and analysis [87].

A refinement and composition approach can help establish certain security properties so that the cost of compromising them is higher than an adversary can afford [83]. To assure these properties, development methods that can yield such systems are preferred. Refinement methods can lead to more secure systems than developing a system first and then analyzing its security. Trusted components and methods of reasoning about the security of the composite system should be developed to complement the refinement effort. The composition logic should resemble the refinement logic.

Security Model Support. The most common form of security addressed by surveyed technologies is integrity, taking the form of access control. Its enforcement typically relies on inserting checks at appropriate places. Another security issue that is similarly enforced is encryption and decryption. Exemplar enforcing techniques are wrappers (Section 5.2), meta-object protocols (Section 5.4), and aspects (Section 5.8).

Confidentiality, in the form of information flow security, along with its composition has been researched extensively in an abstract manner, but few applications addressing information flow security have adopted those research results. It might be worthwhile to investigate the applicability of the various models and techniques proposed in the literature for an environment where modern component technologies are deployed. However, this is a very difficult problem, due to the enormous gap existing between the theoretical models and the reality.

Availability has received much less attention from researchers. It deservers further investigation, and might be well suited for architectural level analysis.

Connector-oriented Architecture. There exists a dichotomy between abstract formal models about security and real practical systems to which the models are supposed to apply. A bridge should be found to cover the gap between the two worlds[98, 122]. The bridge should be formal enough to support modeling and analysis, but it should also be reasonably easy to relate to real systems. Software architecture can be a well-suited bridge, due to its ease of analysis and link to final implementation.

Architecture can be used to specify where security functionality is allocated and what security mechanisms are used to achieve it. An explicit architecture enables the designer to identify security critical areas: places where attacks have happened, places where security functionalities are deployed, and places whose compromise can lead to severe security problems. Then, high assurance but high cost engineering methods, such as formal analysis, can be directed to these areas. A balance has to be achieved between security architecture analysis and functionality and availability of existing components, which usually take precedence over security requirements [98].

In an architecture-based approach, connectors, as the loci for communication, are appropriate to enforce extra functionality when components are connected together. Efforts have been made to construct secure connectors [120]. Recursively applying a compositional approach for the construction of secure connectors seems quite natural. However, current theories on connector compositions [121], like most other composition theories, address security insufficiently, and are still not easy to use. Some work, like [50], addresses security, but the notion of composition, compared to similar notions used in most other formal works, tends to be rather abstract and primitive. Theories, techniques, and tools that can support design and analysis of secure connectors are very much needed.

Description and Enactment mechanisms. Describing security properties of each component (see Section 5.5) still remains a research problem. Current proposals based on logic [68, 69] have not fully demonstrated their power yet. Other forms wait for exploration.

There have been many different types of enactment mechanisms that support augmenting systems with security. Simple wrappers (see Section 5.2) are a suitable choice to handle low-level security, because there is little information available at this abstraction level. Flexible extension mechanisms provided by the original infrastructure can greatly facilitate the development of wrapper support. Agents (see Section 5.3) can serve as high-level security enablers, when each situation will probably ask for a unique solution.

The aspect technology (see Section 0) can be used as a powerful mechanism to enact security. Its generality and expressiveness lies between wrappers and agents. Its support of modularity and flexibility is very desirable. Due to its newness, many more experiments should be conducted to demonstrate its applicability to the security aspect. The Meta Object Protocol (see Section 5.4), in addition to being an enactment mechanism, can also be used as an implementation facility for the aspect technology.

General composition frameworks (see Section 5.6) have not demonstrated much success so far. While the notion is appealing, the content of the frameworks is not rich enough, and the frameworks have not proven that their coverage can meet most security needs. Research on composition mechanisms, especially those on dynamic composition, can be beneficial for other enactment mechanisms.

Software engineering research generally favors flexibility and generality. Whether this will come at the cost of security, due to the complexity and difficulty incurred in design and analysis, should be carefully evaluated. A delicate and calculated balance in tradeoffs must be maintained.

Research Methodology and Plan. The problem of security design and analysis in modular software has been studied from different perspectives. While some have produced many useful results, others are still in an immature age. This survey holds the opinion that a comprehensive methodology to solve the modular security problem is necessary. Here the components of the methodology and the future research plan are outlined.

In summary, the methodology will be architecture-centered and connector-oriented. It can support both top-down refinement and bottom-up composition. It will employ lightweight formal methods to a reasonable extent, not relying on knowledge and labor intensive proofs. It will integrate practical security models beyond simple access control. An architecture model will guide the development of comprehensive security. The component specifications will be extended to support flexible security requirements. Their compositions will be handled by connectors, which impose flexible security modularly and non-intrusively, applying injection and composition technologies such as wrappers, meta-object protocols, and aspects. Usable automatic tools will be developed to support the practice of this methodology.

The details of the methodology are as follows. The methodology will support both top down refinement and bottom up composition. It can be used to design a secure system and its constituent parts, as well as assess the security of an assembly composed from existing components.

The methodology is architecture centered and connector oriented. It is based on a connector-centric architectural model. It employs formal methods for description and analysis, but uses them only to a reasonable extent, not relying on knowledge and labor intensive proof.

An architecture model guides the development of comprehensive security. An architecture model provides a complete picture, which is essential in ensuring each component has received proper attention for security assessment. An architecture model enables the designer to allocate security enforcement into appropriate places. The combination of software architecture and software security will provide novel insights in attaining secure software.

Future research will begin with using an architecture-based model for expressing, reasoning about, and enforcing access control. Access control is the dominant security enforcement mechanism. A solution that can assure its fulfillment at an early stage of software development will facilitate the overall security of the system.

Supporting practical security models other than access control will also be investigated. More specifically, trust and availability might be appropriate issues that can be handled at the software architecture level.

Component specifications will be extended to support security requirements flexibly. Logic or process algebra will be investigated as unifying description mechanisms that can describe several security properties. The possibility of automatic reasoning and analysis based on these descriptions will be explored at both the design time and run-time settings.

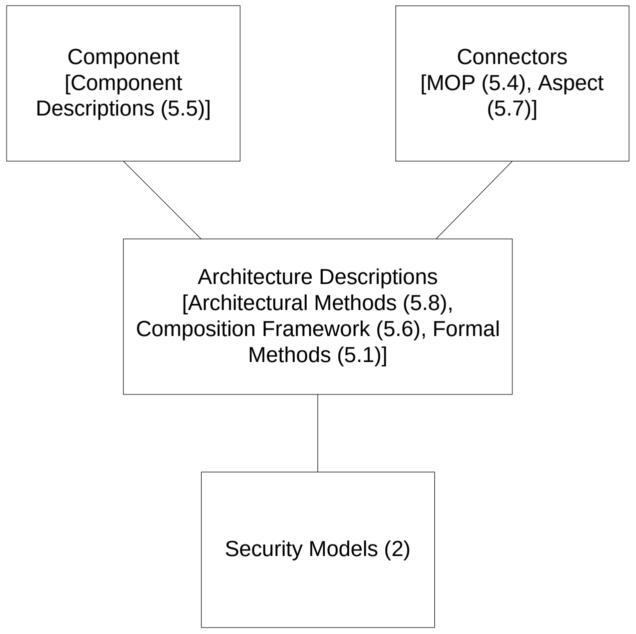

Component [Component Descriptions (5.5)]
Connectors [MOP (5.4), Aspect (5.7)]
Architecture Descriptions [Architectural Methods (5.8), Composition Framework (5.6), Formal Methods (5.1)]
Security Models (2)

Figure 28, Research Plan

The composition of the components will be handled by connectors, which impose flexible security modularly and non-intrusively. The reasoning and analysis aforementioned might take place at connectors. A connector can apply injection and composition technologies to introduce security onto base communication capabilities. The aspect technology, combined with meta-object protocols, can be a promising mechanism. More experiments using them for connector implementation will be conducted.

The methodology will be supported by an architecture description language. The language enables security design and analysis for systems made of components and connectors. The design-time and run-time support for the language will be developed. Automatic tools to support the practices and activities of this methodology will be developed.

Case studies will be performed to evaluate the effectiveness of the methodology. Significant examples will be developed to demonstrate the applicability of the methodology. Representative applications from research literature might be rearchitected using the proposed methodology to assess improvements brought about by the methodology.

The relationship between the constituent parts of the proposed methodology and the techniques surveyed in this paper is depicted in Figure 28 (the numbers are the numbering of the sections surveying the techniques).

7. ACKNOELEDGEMENTS

The author would like to thank helpful comments and discussions from Yuzo Kanomata, Justin Erenkrantz, Eric Dashofy, and Kari Nies.

The author is supported partly by the National Science Foundation under grant number 0326105 and by the Intel Corporation. The content of the work should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of either organization.

8. REFERENCES

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

[2] Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems*. 1986: MIT Press.

[3] Agha, G.A. and R. Ziaei. *Security and Fault-Tolerance in Distributed Systems: An Actor-Based Approach*. in Proceedings of Computer Security, Dependability and Assurance: From Needs to Solutions p.72-88, 1998.

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

[5] Anderson, J.P., *Computer Security Technology Planning Study*. 1972, ESD/AFSC, Hanscom AFB: Bedford, MA.

[6] Astley, M. and G.A. Agha. *Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management*. in Proceedings of 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, p.1-9, 1998.

[7] Atluri, V., S.A. Chun, and P. Mazzoleni. *A Chinese Wall Security Model for Decentralized Workflow Systems*. in Proceedings of 8th ACM Conference on Computer and Communications Security, p.48-57, 2001.

[8] Balzer, R.M. and N.M. Goldman. *Mediating Connectors: A Non-Bypassable Process Wrapping Technology*. in Proceedings of DARPA Information Survivability Conference & Exposition, p.361-368 vol.2, 2000.

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

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

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

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

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

[14] Bieber, P., D. Raujol, and P. Siron. *Security Architecture for Federated Cooperative Information Systems*. in Proceedings of 16th Annual Computer Security Applications Conference, p.208-216, 2000.

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

[16] Bonatti, P. and Sabrina, *An Algebra for Composing Access Control Policies*. ACM Transactions on Information and System Security 2002. **5**(1): p. 1-35.

[17] Brewer, D.F.C. and M.J. Nash. *The Chinese Wall Security Policy*. in Proceedings of 1989 IEEE Symposium on Security and Privacy, p.206-214, 1989.

[18] Burrows, M., A. Abadi, and R. Needham, *A Logic of Authentication*. ACM Transactions on Computer Systems, 1990. **8**(1): p. 18-36.

[19] Caplan, K. and J.L. Sanders, *Building an International Security Standard*. IT Professional, 1999. **1**(2): p. 29-34.

[20] Caromel, D., F. Huet, and J. Vayssière. *A Simple Security-Aware Mop for Java*. in Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, p.118-125, 2001.

[21] Caromel, D. and J. Vayssiere. *Reflections on Mops, Components, and Java Security*. in Proceedings of 15th European Conference on Object-Oriented Programming, p.256-74, 2001.

[22] Clark, D.D. and D.R. Wilson. *A Comparison of Commercial and Military Computer Security Policies*. in Proceedings of 1987 IEEE Symposium on Security and Privacy, p.184-94, 1987.

[23] David, M. and B. Pagurek. *A Runtime Composite Service Creation and Deployment Infrastructure and Its Applications in Internet Security, E-Commerce, and Software Provisioning*. in Proceedings of 25th Annual International Computer Software and Applications Conference, p.371-376, 2001.

[24] Dawson, S., S. Qian, and P. Samarati. *Secure Interoperation of Heterogeneous Systems: A Mediator-Based Approach*. in Proceedings of 14th IFIP TC-11 International Conference on Information Security, 1998.

[25] Dawson, S., et al. *Secure Access Wrapper: Mediating Security between Heterogeneous Databases*. in Proceedings of DARPA Information Survivability Conference & Exposition, p.308-322 vol.2, 2000.

[26] de Bruin, H. and H. van Vliet. *Top-Down Composition of Software Architectures*. in Proceedings of 9th IEEE International Conference and Workshop on the Engineering of Computer Based Systems, p.147-156, 2002.

[27] Deng, Y., et al., *An Approach for Modeling and Analysis of Security System Architectures*. IEEE Transactions on Knowledge and Data Engineering, 2003. **15**(5): p. 1099-1119.

[28] Denning, D.E., *A Lattice Model of Secure Information Flow*. Communications of the ACM, 1976. **19**(5): p. 236-43.

[29] Dobson, J.E. and B. Randell. *Building Reliable Secure Computing Systems out of Unreliable Insecure Components*. in Proceedings of 17th Annual Computer Security Applications Conference, p.164-173, 2001.

[30] Duclos, F., J. Estublier, and P. Morat. *Describing and Using Non Functional Aspects in Component Based Applications*. in Proceedings of 1st International Conference on Aspect-Oriented Software Development, p.65-75, 2002.

[31] Emmerich, W. *Distributed Component Technologies and Their Software Engineering Implications*. in Proceedings of 24th International Conference on Software Engineering, p.537-546, 2002.

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

[33] Feiertag, R.J., et al. *Intrusion Detection Inter-Component Adaptive Negotiation*. in Proceedings of 2nd International Workshop on Recent Advances in Intrusion Detection, 1999.

[34] Feldman, M. *Enterprise Wrappers for Information Assurance*. in Proceedings of DARPA Information Survivability Conference & Exposition III, p.120-122, 2003.

[35] Filman, R. and T. Linden. *Safebots: A Paradigm for Software Security Controls*. in Proceedings of 1996 New Security Paradigms Workshop, p.45-51, 1996.

[36] Focardi, R., *Analysis and Automatic Detection of Information Flows in Systems and Networks*, in *Department of Computer Science*. 1998, University of Bologna, Italy.

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

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

[39] Focardi, R. and R. Gorrieri, *Classification of Security Properties: (Part I: Information Flow)*, in *Foundations of Security Analysis and Design : Tutorial Lectures*. 2001, Springer-Verlag Heidelberg. p. 331-396.

[40] Fraser, T., L. Badger, and M. Feldman. *Hardening Cots Software with Generic Software Wrappers*. in Proceedings of DARPA Information Survivability Conference & Exposition, p.323-337 vol.2, 2000.

[41] Fukuzawa, K. and M. Saeki. *Evaluating Software Architectures by Coloured Petri Nets*. in Proceedings of 14th International Conference on Software Engineering and Knowledge Engineering, p.263-270, 2002.

[42] Garfinkel, T. *Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools*. in Proceedings of 10th Annual Network and Distributed System Security Symposium, 2003.

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

[44] Gilham, F., R.A. Riemenschneider, and V. Stavridou. *Secure Interoperation of Secure Distributed Databases: An Architecture Verification Case Study*. in Proceedings of FM'99 - Formal Methods, Wold Congress on Formal Methods in the Development of Computing Systems, Vol. I, p.701-717, 1999.

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

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

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

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

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

[50] Heintze, N. and J.D. Tygar, *A Model for Secure Protocols and Their Compositions*. IEEE Transactions on Software Engineering, 1996. **22**(1): p. 16-30.

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

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

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

[54] Herrmann, P. *Trust-Based Protection of Software Component Users and Designers*. in Proceedings of 1st International Conference on Trust Management, p.75-90, 2003.

[55] Herrmann, P. and H. Krumm. *Trust-Adapted Enforcement of Security Policies in Distributed Component-Structured Applications*. in Proceedings of 6th IEEE Symposium on Computers and Communications, p.2-8, 2001.

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

[57] Hoare, C.A.R., *Communicating Sequential Processes*. 1985: Prentice-Hall. viii+256.

[58] Ivan, A.-A. and V. Karamcheti. *Using Views for Customizing Reusable Components in Component-Based Frameworks*. in Proceedings of 12th IEEE International Symposium on High Performance Distributed Computing, p.194-204, 2003.

[59] Jaeger, T., et al. *Security Architecture for Component-Based Operating Systems*. in Proceedings of 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications, p.222-228, 1998.

[60] Jaeger, T., X. Zhang, and F. Cacheda, *Policy Management Using Access Control Spaces*. ACM Transactions on Information and System Security 2003. **6**(3): p. 327-364.

[61] Johnson, D.M. and F.J. Thayer. *Security and the Composition of Machines*. in Proceedings of 1st IEEE Computer Security Foundations Workshop, p.72-89, 1988.

[62] Kang, M.H., B.J. Eppinger, and J.N. Froscher. *Tools to Support Secure Enterprise Computing*. in Proceedings of 15th Annual Computer Security Applications Conference, p.143-152, 1999.

[63] Kang, M.H. and J.N. Froscher. *Software Architecture and Logic for Secure Applications*. in Proceedings of DARPA Information Survivability Conference & Exposition, p.391-405 vol.1, 2000.

[64] Kang, M.H., J.N. Froscher, and I.S. Moskowtiz. *A Framework for Mls Interoperability*. in Proceedings of 1st IEEE High-Assurance Systems Engineering Workshop, p.198-205, 1996.

[65] Kang, M.H., A.P. Moore, and I.S. Moskowitz, *Design and Assurance Strategy for the Nrl Pump*. Computer, 1998. **31**(4): p. 56-64.

[66] Kang, M.H. and I.S. Moskowitz, *A Data Pump for Communication*. 1995, Naval Research Laboratory.

[67] Kang, M.H., I.S. Moskowitz, and D.C. Lee, *A Network Pump*. IEEE Transactions on Software Engineering, 1996. **22**(5): p. 329-338.

[68] Khan, K., J. Han, and Y. Zheng. *A Framework for an Active Interface to Characterise Compositional Security Contracts of Software Components*. in Proceedings of 2001 Australian Software Engineering Conference, p.117-126, 2001.

[69] Khan, K.M. and J. Han, *Composing Security-Aware Software*. IEEE Software, 2002. **19**(1): p. 34-41.

[70] Khan, K.M. and J. Han. *A Security Characterisation Framework for Trustworthy Component Based Software Systems*. in Proceedings of 27th Annual International Computer Software and Applications Conference, p.164-169, 2003.

[71] Khan, K.M., J. Han, and Y. Zheng. *Security Characterisation of Software Components and Their Composition*. in Proceedings of 36th International Conference on Technology of Object-Oriented Languages and Systems, p.240-249, 2000.

[72] Kiczales, G., et al. *Aspect-Oriented Programming*. in Proceedings of 11th European Conference on Object-Oriented Programming, p.220-42, 1997.

[73] Kiczales, G., J.d. Rivieres, and D.G. Bobrow, *The Art of the Metaobject Protocol*. 1991: MIT Press.

[74] Kotonya, G. and N. Maiden, *Editorial Component-Based Software Engineering*. IEE Proceedings-Software, 2000. **147**(6): p. 201.

[75] Lamport, L., *The Temporal Logic of Actions*. ACM Transactions on Programming Languages and Systems, 1994. **16**(3): p. 872-923.

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

[77] Lampson, B.W., *Protection*. ACM SIGOPS Operating Systems Review, 1974. **8**(1): p. 18-24.

[78] Lindqvist, U. and E. Jonsson. *A Map of Security Risks Associated with Using Cots*. in Proceedings of Computer, p.60-66, 1998.

[79] Maes, P. *Concepts and Experiments in Computational Reflection*. in Proceedings of 2nd ACM SIGPLAN Conference on Object-oriented programming systems, languages and applications, p.147-155, 1987.

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

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

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

[83] McLean, J. *New Paradigms for High Assurance Software*. in Proceedings of 1992-1993 New Security Paradigms Workshop, p.42-47, 1993.

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

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

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

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

[88] Medvidovic, N., et al., *Modeling Software Architectures in the Unified Modeling Language*. ACM Transactions on Software Engineering and Methodology, 2002. **11**(1): p. 2-57.

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

[90] Millen, J., *20 Years of Covert Channel Modeling and Analysis*, in *1999 IEEE Symposium on Security and Privacy*. 1999. p. 113-114.

[91] Milner, R., *Communication and Concurrency*. 1989: Prentice Hall. xi+260.

[92] Minsky, N.H. *Should Architectural Principles Be Enforced?* in Proceedings of Computer Security, Dependability and Assurance: From Needs to Solutions, p.89-102, 1998.

[93] Mitchem, T., R. Lu, and R. O'Brien. *Using Kernel Hypervisors to Secure Applications*. in Proceedings of 13th Annual Computer Security Applications Conference, p.175-181, 1997.

[94] Mitchem, T., et al. *Linux Kernel Loadable Wrappers*. in Proceedings of DARPA Information Survivability Conference & Exposition, p.296-307 vol.2, 2000.

[95] Moriconi, M., X. Qian, and R.A. Riemenschneider, *Correct Architecture Refinement*. IEEE Transactions on Software Engineering, 1995. **21**(4): p. 356-372.

[96] Moriconi, M., et al. *Secure Software Architectures*. in Proceedings of 1997 IEEE Symposium on Security and Privacy, p.84-93, 1997.

[97] Myers, A.C. and B. Liskov, *Protecting Privacy Using the Decentralized Label Model*. ACM Transactions on Software Engineering & Methodology, 2000. **9**(4): p. 410-42.

[98] Nelson, R. *Integrating Formalism and Pragmatism: Architectural Security*. in Proceedings of 1997 New Security Paradigms Workshop, p.1-4, 1997.

[99] Olawsky, D., et al. *Using Composition to Design Secure, Fault-Tolerant Systems*. in Proceedings of 3rd IEEE International High-Assurance Systems Engineering Symposium, p.29-32, 1998.

[100] Olawsky, D., et al., *Using Composition to Design Secure, Fault-Tolerant Systems*, in *DARPA Information Survivability Conference & Exposition*. 2000. p. 380-390 vol.2.

[101] Owre, S., J.M. Rushby, and N. Shankar. *Pvs: A Prototype Verification System*. in Proceedings of 11th International Conference on Automated Deduction, p.748-52, 1992.

[102] Pawlak, R., et al. *Distributed Separation of Concerns with Aspect Components*. in Proceedings of 33rd International Conference on Technology of Object-Oriented Languages and Systems, p.276-287, 2000.

[103] Pawlak, R., et al. *Jac: A Flexible Solution for Aspect-Oriented Programming in Java*. in Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, p.1-24, 2001.

[104] Payne, C.N., Jr., *Using Composition and Refinement to Support Security Architecture Trade-Off Analysis*, in *22nd National Information Systems Security Conference*. 1999: Gaithersburg, MD. p. 238-44.

[105] Peri, R.V., W.A. Wulf, and D.M. Kienzle. *A Logic of Composition for Information Flow Predicates*. in Proceedings of 9th IEEE Computer Security Foundations Workshop, p.82-94, 1996.

[106] Riechmann, T. and F.J. Hauck. *Meta Objects for Access Control: Extending Capability-Based Security*. in Proceedings of 1997 New Security Paradigms Workshop, p.17-22, 1997.

[107] Riechmann, T. and F.J. Hauck. *Meta Objects for Access Control: A Formal Model for Role-Based*

Principals. in Proceedings of 1998 New Security Paradigms Workshop, p.30-38, 1998.

[108] Ryan, P., et al. *Non-Interference, Who Needs It?* in Proceedings of 14th IEEE Computer Security Foundations Workshop, p.237-238, 2001.

[109] Ryan, P.Y.A., *Mathematical Models of Computer Security*, in *Foundations of Security Analysis and Design : Tutorial Lectures*. 2001, Springer-Verlag Heidelberg. p. 1-62.

[110] Ryan, P.Y.A. and S.A. Schneider. *Process Algebra and Non-Interference*. in Proceedings of 12th IEEE Computer Security Foundations Workshop, p.214-227, 1999.

[111] Sabelfeld, A. and A.C. Myers, *Language-Based Information-Flow Security*. IEEE Journal on Selected Areas in Communications, 2003. **21**(1): p. 5-19.

[112] Samarati, P. and S.d.C.d. Vimercati, *Access Control: Policies, Models, and Mechanisms*, in *Foundations of Security Analysis and Design : Tutorial Lectures*. 2001, Springer-Verlag Heidelberg. p. 137-196.

[113] Sandhu, R.S., et al., *Role-Based Access Control Models*. Computer, 1996. **29**(2): p. 38-47.

[114] Santen, T., M. Heisel, and A. Pfitzmann. *Confidentiality-Preserving Refinement Is Compositional - Sometimes*. in Proceedings of 7th European Symposium on Research in Computer Security, p.194-211, 2002.

[115] Schneider, F.B., *Enforceable Security Policies*. ACM Transactions on Information and System Security 2000. **3**(1): p. 30-50.

[116] Sewell, P. and J. Vitek. *Secure Composition of Untrusted Code: Wrappers and Causality Types*. in Proceedings of 13th IEEE Computer Security Foundations Workshop, p.269-284, 2000.

[117] Shah, V. and F. Hill. *An Aspect-Oriented Security Framework*. in Proceedings of DARPA Information Survivability Conference & Exposition III, p.143-145, 2003.

[118] Shands, D., et al., *Secure Virtual Enclaves: Supporting Coalition Use of Distributed Application Technologies*. ACM Transactions on Information and System Security, 2001. **4**(2): p. 103-133.

[119] Shaw, M. *The Coming-of-Age of Software Architecture Research*. in Proceedings of 23rd International Conference on Software Engineering, p.656-664, 2001.

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

[121] Spitznagel, B. and D. Garlan. *A Compositional Formalization of Connector Wrappers*. in Proceedings of 25th International Conference on Software Engineering, p.374-384, 2003.

[122] Stavridou, V., et al. *Intrusion Tolerant Software Architectures*. in Proceedings of DARPA Information Survivability Conference & Exposition II, p.230-241 vol.2, 2001.

[123] Stavridou, V., R.A. Riemenschneider, and F. Gilham. *Sdtp: A Verified Architecture for Secure Distributed Transaction Processing*. in Proceedings of DARPA Information Survivability Conference & Exposition, p.369-379 vol.2, 2000.

[124] Sutherland, D. *A Model of Information*. in Proceedings of 9th National Computer Security Conference, p.175-183, 1986.

[125] Szyperski, C., *Component Software - Beyond Object-Oriented Programming*. Second Edition ed. 2002: Addison-Wesley.

[126] Truyen, E., et al. *Dynamic and Selective Combination of Extensions in Component-Based Applications*. in Proceedings of 23rd International Conference on Software Engineering, p.233-242, 2001.

[127] Welch, I. *Reflective Enforcement of the Clark-Wilson Integrity Model*. in Proceedings of 2nd Workshop on Distributed Object Security, 1999.

[128] Welch, I. and Robert Stroud. *Security and Aspects: A Metaobjects Protocol Viewpoint*. in Proceedings of Workshop on Aspects, Components and Patterns for Infrastructure Software at the 1st International Conference on Aspect-Oriented Software Development, 2002.

[129] Welch, I. and R. Stroud. *From Dalang to Kava-the Evolution of a Reflective Java Extension*. in Proceedings of 2nd International Conference Meta-Level Architectures and Reflection, p.2-21, 1999.

[130] Welch, I. and R. Stroud. *Supporting Real World Security Models in Java*. in Proceedings of 7th IEEE Workshop on Future Trends of Distributed Computing Systems, p.155-9, 1999.

[131] Welch, I. and R.J. Stroud, *Using Reflection as a Mechanism for Enforcing Security Policies on Compiled Code*. Journal of Computer Security, 2002. **10**(4): p. 399-432.

[132] Welch, I. and R.J. Stroud. *Dynamic Adaptation of the Security Properties of Applications and Components*. in Proceedings of ECOOP'98 Workshop Reader, p.282, 1998.

[133] Welch., I. *Adding Security to Commercial-Off-the-Shelf Software*. in Proceedings of 1997 European Research Seminar on Advances in Distributed Systems, 1997.

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

[135] Wohlstadter, E., S. Jackson, and P. Devanbu. *Dado: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems*. in Proceedings of 25th International Conference on Software Engineering, p.174-186, 2003.

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

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

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

[139] Zdancewic, S., et al., *Secure Program Partitioning.* ACM Transactions on Computer Systems, 2002. **20**(3): p. 283-328.

[140] Zhong, Q. and N. Edwards, *Security Control for Cots Components*. Computer, 1998. **31**(6): p. 67-73.