AN ALGORITHM FOR CHECKING SOFTWARE ARCHITECTURE DESCRIPTIONS FOR CONFIDENTIALITY

Technical Report

Ulu, Cemil and Oğuztüzün, Halit
Department of Computer Engineering
Middle East Technical University
January 2004

Abstract

This report addresses the confidentiality aspect of the information security problem from the viewpoint of the software architecture. It presents a new approach to secure system design in which the desired security properties, in particular, confidentiality, of the system are proven to hold at the architectural level. A software architecture configuration described in the extended architecture description language, Wright/c, is statically analyzed to verify that confidentiality requirements as per Bell LaPadula principles are satisfied. A verification algorithm that takes the Wright/c description and the access control lattice model as inputs, and that checks if there is a potential violation of the Bell-LaPadula principles based on data flow analysis is developed. The algorithm also detects excess privileges. A software tool, which features an XML-based front-end to the algorithm is constructed. Finally, the algorithm is analyzed for its soundness, completeness and computational complexity.

Keywords: Lattice-based access control, clearance, data flow, confidentiality, software architecture, architecture description language, privilege, Bell-LaPadula.

1. Introduction

As technology advances and information management systems become more powerful, the problem of enforcing information security also becomes more critical. Thus, the problem of protecting information has been an important issue since the increasing development of

information technology in the past few years, which has led to the widespread use of computer systems to store and manipulate information. Recent security compromises to widely distributed software such as various web browsers, operating systems, and the application software have caused widespread enterprisewide outages and are closely monitored. The majority of the security compromises can be attributed to one or more weaknesses within the integral components that make up the software. Therefore, a computer and network system must be protected in terms of *availability, confidentiality, and integrity* These three information security objectives are separate but interrelated:

- *Confidentiality* (or secrecy) is related to disclosure of information.
- *Integrity* is related to modification of information.
- *Availability* is related to denial of access to information.

Security issues have many concerns at different steps in an information flow and different levels of abstraction in an information system. Higher level of abstraction for a system leads to powerful expressiveness in constructing the system. *Software architecture* is emerging as an important discipline for engineers of software. It has emerged over time as a natural evolution of design abstractions, as engineers have searched for better ways to understand their software and new ways to build larger and more complex software systems.

In this report, we relate studies in two distinct realms, namely description of software architectures, and access control models, particularly the lattice-based access control model, to lay a foundation for secure software architectures (figure 1). Our concern is the confidentiality and the information flow in a system at architectural level. It is a new approach to a secure system design in which the various representations of the architecture of a software system are described formally and the desired security properties, in particular confidentiality, of the system are proven to hold at the architectural level. We focus on a static analysis of a software architecture during the design phase to assure end-to-end secure

information flow. The confidentiality properties defined by Bell and LaPadula, namely the 'simple security property' and the '* property', are incorporated into the analysis.

For the architectural description, we adopt the Wright architecture description language in which component and connector, port and role, style and configuration aspects are clearly identified. Other architecture description languages such as UniCon [40] and particularly Rapide [27, 24] were also good candidates. Wright is chosen because it supports the formal specification and analysis of interactions between architectural components, and it has already been well studied. Wright is based on a process algebra called CSP (Communicating Sequential Processes) [16,15] to describe the behaviour of entities such as port, component, role, and connector. In order to enhance the architectural description of a software system to address end-to-end security issues, Wright is annotated, called Wright/c [43], by labeling its constructs with sensitivity levels, and authorizations.

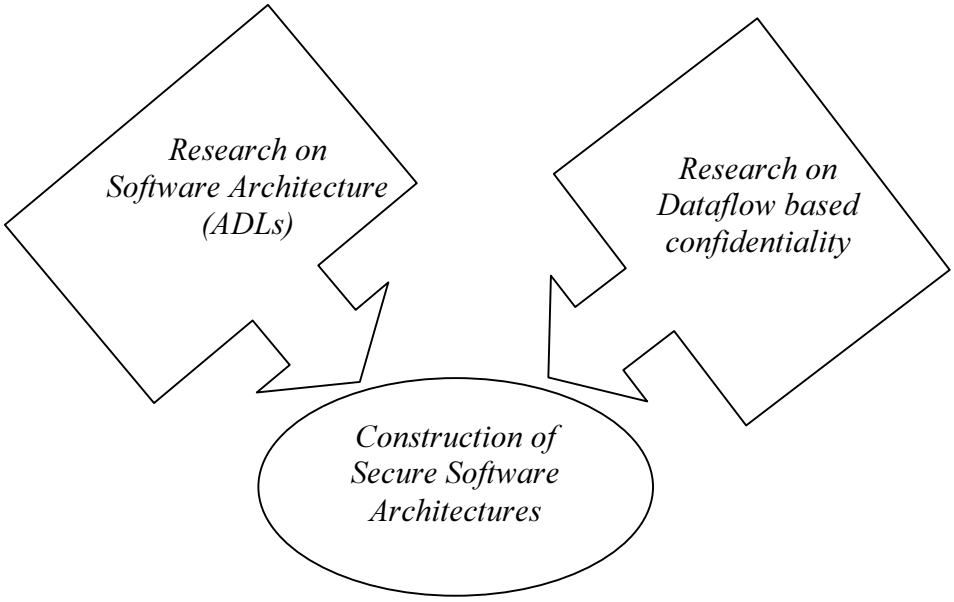

Figure 1: General view of the study

This report is structured as follows: relevant background is summarized in section 2; section 3 contains the approach and the verification algorithm developed to statically verify a Wright/c description including the correctness (completeness and the soundness) of the

algorithm with its computational time and space complexity. Lastly, section 4 discusses the approach and its potential.

2. Background

This section summarizes the relevant background on software architecture, particularly architecture description languages, and access control models.

2.1. Software Architectures

As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerge as a new kind of problem. Structural issues include gross organization and global control structure, protocols for communication, synchronization, and data access, assignment of functionality to design elements, physical distribution, composition of design elements, scaling and performance, and selection among design alternatives [14]. This is the *software architecture* level of design. An *architecture* is a specification of the components of a system and the interactions between them [1, 5, 41]. Systems are constrained to conform to an architecture. An architecture should guarantee certain behavioral properties of a conforming system, i.e. one whose components are configured according to the architecture. An architecture should also be useful in various ways during the process of building a system [23,13,12,42,5,28,40].

Software architectures are intended to describe essential high level structural and behavioral characteristics of software-intensive systems. They allow programmers to compose an application from a mixture of existing software modules, third party libraries, legacy programs and a minimal amount of code developed for that particular application. A software architecture defines applications in terms of components, connectors and configurations [18,12,5,8]:

- A *component* is a computational unit written in some programming language,

- A *connection* defines the type of interactions among components (e.g. remote procedure calls) via *connectors* that transport data between components and perform the necessary transformations on that data, so that the interfaces of the components are respected.
- A *configuration* defines the application structure through components and their interconnections. In addition to specifying the structure and topology of the system, the configuration shows the intended correspondence between system requirements and the elements of the constructed system.

2.2. Architecture Description Languages

A*rchitecture Description Languages (ADL)* [8, 9] provide a conceptual framework and a concrete syntax for characterizing and describing software architectures. They describe essential high level structural and behavioral characteristics in a way that can be analyzed and manipulated algorithmically. These languages are used to specify architectural elements, such as components, connectors and constraints, and how these elements fit together. CMU Software Engineering Institute maintains links to the Web pages of number of ADLs; see http://www.sei.cmu.edu/architecture/adl.html.

To demonstrate our ideas we have chosen Wright as a vehicle due to its suitability for static analysis.

2.2.1. Wright Architecture Description Language and Wright/c

Wright is introduced in R. Allen's PhD thesis [1]. The language provides a practical formal basis for the description of both architectural configurations and architectural styles. It has the ability to describe the abstract behaviour of the components and the connectors using CSP (Communicating Sequential Processes) notation [2]. The CSP theory is an attractive base for the analysis of high level system description because the theory provides an expressive process-algebraic programming notation, a range of semantic models of varying degrees of

abstraction, powerful notions of refinement and abstraction, and a useful set of equality and refinement laws.

In Wright, a *component type* is described as a set of *ports* and a *computation* that specifies the component's abstract behavior. Each port defines a logical point of interaction between component and its environment. Ports allow a component to define multiple interfaces to other parts of a system.

A connector type is defined by a set of *roles* and a *glue* specification. The roles describe the expected local behavior of each of the interesting parties. That is, they act as a specification that determines the obligations of each component participating in the interaction.

In order to describe a complete system architecture, the components and connectors of a Wright description must be combined into a *configuration*. A configuration is a collection of component instances combined via connectors. If an architect is dealing with a single system, (s)he introduces component and connector types and then uses these to create instances of components and connectors. Then, topology of the configuration is specified through attachments of ports to roles. Often, however, an architect is concerned not with a single system in isolation, but rather with a system in the context of entire family of systems. So, he seeks not merely to develop an arbitrary architecture, but to select that architecture from a particular *style,* i.e. a family of architectures. A style defines a set of properties that are shared by the configurations that are members of the style. These properties can include a common vocabulary and restrictions on the ways this vocabulary can be used in configurations.

In [43], Wright is extended so that confidentiality authorizations can be specified. An architectural description in Wright/c, the extended language, assigns clearance to the ports of the components and treats security labels as a part of data type information. As dictated by the

security policy, the security labels are declared along with clearance assignments in an access control lattice model, also expressed in Wright/c.

2.3 Access Control Policies

A set of restrictions on who can read/write which kind of data make the access control policy of a system. The access control models, which include lattice-based models (Bell-LaPadula Model [20,37,6], the Biba integrity model [7,29,37], Denning's information flow model [10,37]), the access matrix model [38], logic-based models [3,4], certificate-based models [33] etc., depend on the traditional access control policies given below [39,35,17]:

i. *Discretionary Access Control (DAC)* [34,33,36,20] is based on the idea that the owner of data should determine who has access to it. DAC allows data to be freely copied from an object to another object, so even if access to the original data is denied, access to a copy can be obtained. The controls are discretionary in the sense that a subject with certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject. Individual users are owners of objects and therefore have complete discretion over who should be authorized to access the object and in which mode (e.g. read or write). Ownership is usually acquired as a consequence of creating the object.

DAC, which is the most common type of control mechanism implemented in information systems today, has an inherent weakness that information can be copied from one object to another, so access to a copy is possible even if the owner of the original does not provide access to the original. Moreover, such copies can be propagated by Trojan Horse software without explicit cooperation of users who are allowed to access to the original. For example, assume that Alice owns an object and that she decides to grant Bob access to it. Later on she changes her decision and revoke Bob's access. Now a question that arises is whether or not Bob can further grant access to Charlie. In turn this causes the problems of cascading revoke. Suppose also that Alice grants a permission X to Bob with the grant option.

Bob then grants X to Charlie, followed by a grant X from Alice to Charlie. Now Alice revokes X from Charlie. Now the question arises: “Should Alice’s revoke override Bob’s grant or should Bob’s grant override Alice’s revoke?”. Therefore, discretionary policies do not enforce any control on the flow of information once this information is acquired by a process. This makes it possible for processes to leak information to users not allowed to read it.

ii. *Mandatory Access Controls (MAC)* [37,20,19], confine the transfer of information to one direction in a lattice of security labels (for example, low to high but not high to low). MAC emerged from confidentiality requirements of the military but has broad applications for integrity and separation objectives. MAC provides a restriction to access to objects based on the sensitivity (represented by *a label*) of the information contained in the objects and the formal authorization (i.e. *clearance*) of subjects to access information of such sensitivity. For MAC, all the resources in the computer (OS files, processes, users, tables, etc.) are labeled with a tag which indicates the sensitivity of that object or row of data. Data is marked by at least its classification (e.g. unclassified, restricted, secret, top secret) and possibly by compartments that designate specific subject area and restrict the data further to certain groups (e.g. a confidential project).

MAC ensures that a user can only gain access to an object or data if the relationship between the user’s clearance and the object’s or data’s label should permit the access. So, a user who has logged into a system at the level ‘secret’ may be able to read secret and unclassified data, but he cannot read top secret data and he can update only secret data. This form of access control is called mandatory because it is always enforced by the operating system and the database server automatically and cannot, in general, be changed at the discretion of the owners of the data, unlike discretionary access control provided by ‘ordinary’ operating system and database servers.

For mandatory access control, data have associated sensitivity labels that also apply to

copies and derivatives of the data, so as to prohibit downward information flow. For DAC, data are not labeled and copies of the same information can have independent access control lists.

Throughout the report, sensitivity label and secrecy label are used interchangeably.

Having given the traditional access control policies, in the next subsection, lattice based access control model is presented.

2.3.1 Lattice-based Access Control Models

In this section, we recall basic notions on orders and lattices. The relationship of the information flow policy proposed by Denning [37], and the lattices also is presented. Next, the Bell LaPadula confidentiality model, which is used in our study and based on the lattice-based access control models, is given.

Definition (*Partial order)*: A relation R on a set is called *a partial order* if it is reflexive, antisymmetric and transitive. A set S together with a partial ordering R is called a *partially ordered set, or poset,* and it is denoted by (S,R). [31]

For example, let $\rho(A)=2^A = X$ be the power set of a set A. That is, X is the set of subsets of A. The relation of set inclusion ($\subseteq$) on X is a partial ordering, because, it is:

- reflexive : $a \subseteq a$ for every subset a of X,
- antisymmetric : if $a \subseteq b$ and $b \subseteq a$, then $a = b$ for every subsets a, b of X,
- transitive: $a \subseteq b$ and $b \subseteq c$ implies $a \subseteq c$, where a, b and c are subsets of X.

Definition (*Hasse diagram*): An undirected graph, which is derived after the following reductions are applied to a directed graph for a finite poset, is called a *Hasse Diagram*:

- since a loop is present for each node (reflexivity), these loops are removed,
- all edges present because of transitivity are removed,
- all the arrows on the directed edges are removed .

For example, figure 2 illustrates the construction of a hasse diagram for ($\{1,2,3,4\},\leq$).

Definition (*upper bound, lower bound, the least upper bound, the greatest lower bound)*: For a subset A of a poset $(S, \preceq)$, if u is an element of S such that $a \preceq u$ for all $a \in A$, then u is called an *upper bound of A*. Likewise, if $u \preceq a$ for all elements of $a \in A$, then u is called a *lower bound* of A. An element x is called the *least upper bound* of a subset A if x is an upper bound of A and $x \preceq z$ for all z, where z is an upper bound of A. Similarly, an element y is called the *greatest lower bound* of a subset A if y is a lower bound of A and $z \preceq y$ whenever z is a lower bound of A.

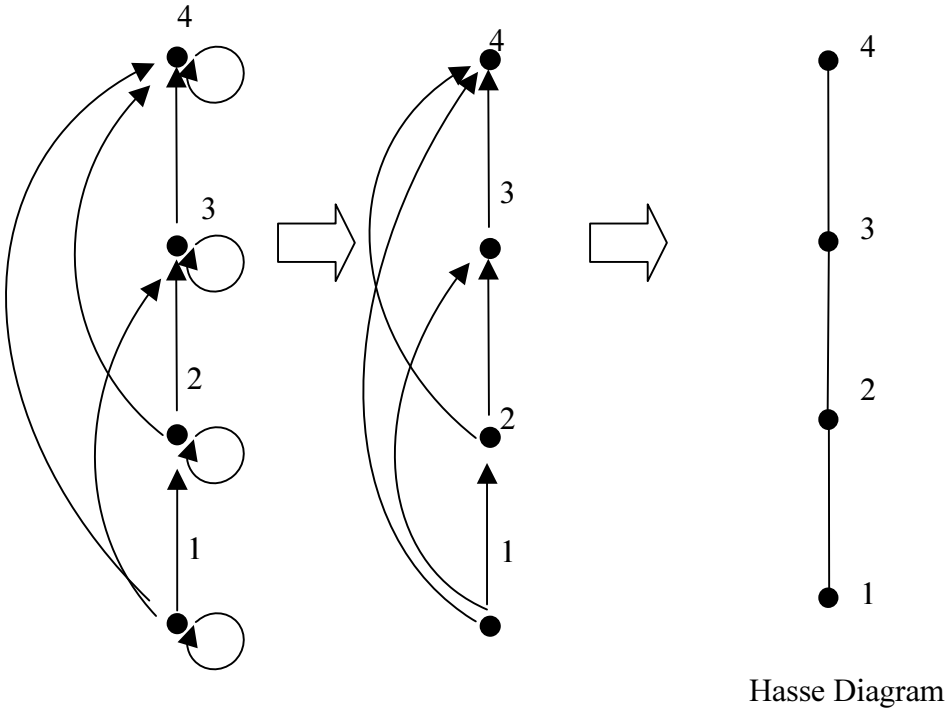

Figure 2: Construction of a Hasse diagram

Definition (*Directed set)*: If $(S, \preceq)$ is a partial order, then a subset $D \subseteq S$ is *directed* if every finite $S_0 \subseteq D$ has an upper bound in D; in other words, there is $y \in D$ such that $x \preceq y$ for all $x \in S_0$ [30].

Definition (*lattice*): A partially ordered set in which every pair of elements has both a least upper bound and a greatest lower bound. For example, the poset diagram in figure 3a is a lattice whereas figure 3b is not.

The poset depicted in Figure 3b fails to be a lattice since the elements b and c have no

least upper bound. Although *d, e, and f* are upper bounds, none of them precede the other two with respect to the ordering of the poset.

Definition (*the principle (order) ideal)*: Let $\mathcal{L}$ be a lattice, and let *a* be an element of the set $\mathcal{L}$. The largest set of elements whose least upper bound is *a* is called the principle (order) ideal of $\mathcal{L}$ generated by *a.*

$$(a] = \{x \in \mathcal{L} : x \leq a \}$$

Definition (*the principle (order) filter)*: Let $\mathcal{L}$ be a lattice, and let *a* be an element of the set $\mathcal{L}$. The largest set of elements whose the greatest lower bound is *a* is called principle (order) filter of $\mathcal{L}$ generated by *a*.

$$[a) = \{x \in \mathcal{L} : a \leq x \}$$

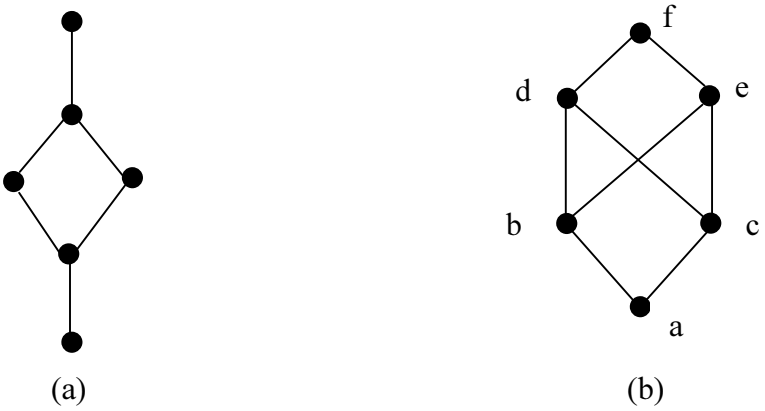

Figure 3: Example Poset diagrams

A lattice model can be used to represent different *information flow* and access control policies. The Bell and LaPadula (BLP) model [20,37,6,26,21] which is the first security model based on the MAC policy is a lattice-based access control model to deal with information flow in computer systems. Information flow is clearly central to confidentiality; this applies to integrity as well.

Information flow policies are concerned with the flow of information from one *security*

class to another [11, 32, 10]. In a system, information actually flows from one object to another. Information flow is usually controlled by *security classes* which are disjoint classes of information. Each object x is bound to a security class which specifies the security class associated with the information stored in x. Whenever information flows from an object x to an object y, there is an accompanying information flow from the security class of x to the security class of y. There are two methods for binding objects to security classes: *static binding,* where the security class of an object is constant, and *dynamic binding,* where the security class of an object varies with its contents. Users may be bound, usually statically, to security classes referred to as *clearance.*

Denning defined the concept of an information flow policy as follows [37]:

A triple $<SC, \rightarrow, \oplus>$ where SC is a set of *security classes*, $\rightarrow \subseteq SC \times SC$ is a binary *can-flow* relation on SC and $\oplus : SC \times SC \rightarrow SC$ is a binary *class-combining* or *join* operator on SC.

Denning also showed that an information flow policy forms a *finite lattice*, that is, with the sensitivity levels as nodes of a graph, the graph formed by the information flow relationship should be acyclic (Denning's axioms):

- The set of security classes SC is *finite*.
- The *can-flow* relation $\rightarrow$ is a *partial order* on SC. That is reflexive, transitive and antisymmetric binary relation. $A \rightarrow B$ denotes that information can flow from the security class A to the security class B.
- SC has a unique *lower bound* L with respect to $\rightarrow$, that is, $L \rightarrow A$ for any $A \in SC$. It acknowledges the existence of public information (i.e. the least sensitive information) in the system.

The *join operator* $\oplus$ is a totally defined least upper bound operator. Join operator combines the information from any two security classes and gives the result a label. It can be

applied to any number of security classes. Thus, least upper bound of $\{A_1, A_2, \ldots, A_n\}$ can be computed by applying the join operator's associativity property.

Similar to *can-flow* relation, a *dominance relation* was defined as:

$A \geq B$ (read as "*A dominates B"*) if and only if $B \rightarrow A$. The *strictly dominates* relation $>$ is defined by $A > B$ if and only if $A \geq B$ and $A \neq B$. A and B are *comparable* if $A \geq B$ or $B \geq A$; otherwise they are *incomparable.*

The key idea in Bell LaPadula (BLP) is to augment discretionary access controls with mandatory access controls to enforce information flow policies. Each subject and object has an attribute associated with it to indicate its clearance and sensitivity level, respectively. The information flow among these sensitivity levels forms a lattice. All authorizations are controlled by a reference monitor which enforces two rules for information flow:

1. *Simply security property:* No subject may read information classified above its clearance ("No read up").
2. *- *property:* No subject may lower the classification of information ("No write down"), that means, a subject *s* can write to an object *o* only if the clearance of *s* is dominated by the security label of *o*.

These rules ensure that information can only flow from a lower sensitivity level to a higher sensitivity level and prevent information flows from high sensitivity level to low sensitivity level.

3. A Confidentiality Verifier

3.1 Approach

Confidential information flow in a software system composed of a number of interacting modules requires an end-to-end behaviour analysis. An *end-to-end* behaviour of a computing system reflects the flow of information between endpoints in the system [32]. This end-to-end information flow also affects the security requirements of the system that we refer to as *end-*

to-end security. Standard security mechanisms such as access control (ACL or capabilities), and firewalls provided by individual modules (components) are inadequate to assure end-to-end security when the system is composed of a number of interacting modules. A firewall, for example, protects confidential information by preventing communication with outside in both directions. Whether this communication violates confidentiality lies outside the scope of firewall mechanism. Access control, on the other hand, does not control how data is used after, for example, it is read from a file. To enforce confidentiality using this access control policy, it is necessary to grant the file access privilege only to processes that will not improperly transmit or leak the confidential data. However, access control mechanisms can not identify these processes; therefore, access control, while useful, can not substitute for information flow control.

Thus, we propose that software architectures seem useful as an abstraction for structuring secure systems that provide end-to-end-security. In a software system described in Wright, the ports of component instances can be regarded as the endpoints in a configuration.

In order to enhance the architectural description of a software system to address end-to-end security issues, we extend the Wright ADL, namely Wright/c, by labeling its constructs with sensitivity levels [43]. The extended description is used in a static verification of a software architecture configuration to assure end-to-end confidentiality of data flow by applying a data flow analysis in port-to-port basis.

Figure 4 illustrates the data flow diagram of the specification and the data flow analysis included in the verification process taking place within a software developer's organization.

The verification process requires two inputs: an access control lattice model that includes security label and authorization level relations, and the Wright/c description of the

software configuration that also includes the confidentiality authorizations in a suitable format for the processing.

The verification procedure can start once the parser constructs the abstract syntax of the configuration, generates the data structures and initializes them. Before detailing the steps of the procedure, some definitions and notations are introduced. The definitions in the sequel refer to a fixed (but arbitrary) configuration under consideration, thus the definitive article in "the configuration".

The construction of the access control lattice model is carried out separately from architectural description regarding the safety requirements that include confidentiality of the information manipulated by the software.

The ADL description of the configuration, on the other hand, is produced using Wright/c language. The lattice model and the description are represented in XML (eXtended Meta Language) to leverage tool support [44].

The verification process performs two main tasks: a data flow analysis process, and an anomaly and excess privilege detection processes. The processes operate on the abstract form of the configuration with respect to the lattice model.

3.2. The Verification Model

The verification procedure can start once the parser (in the front-end) constructs the abstract syntax of the configuration and initializes the generated data structures. Before detailing the steps of the procedure, some definitions and notations are introduced. The definitions in the sequel refer to a fixed (but arbitrary) configuration under consideration, thus the definitive article in "the configuration".

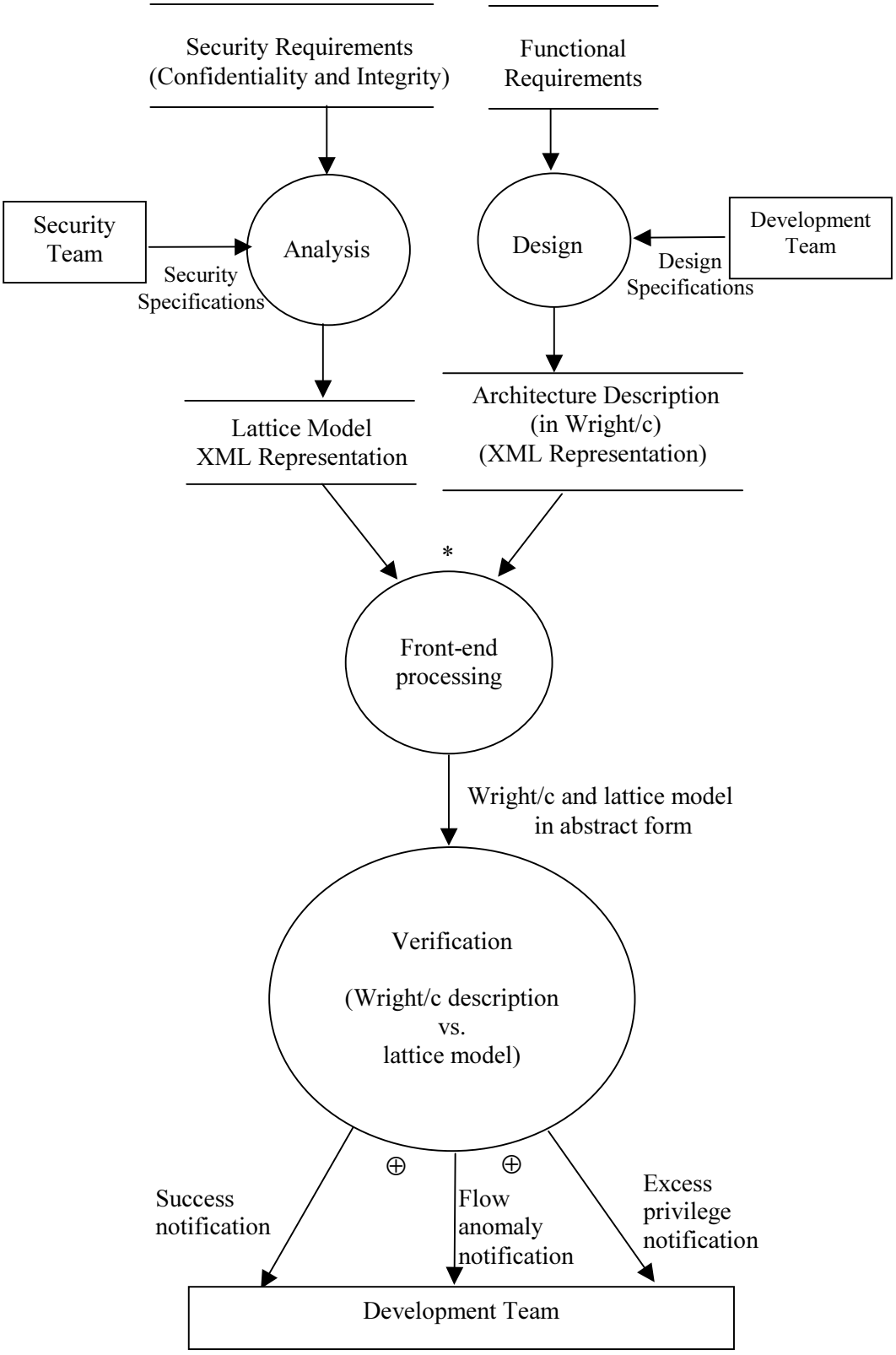

Figure 4: Data flow diagram for the specification and one iteration of verification procedure

Notation (*Components, ports*): The *Components* stands for the set of all component instances in the configuration. The $Ports^c$ is the set of all ports that a component instance $c \in Components$ has.

Notation (*Connectors, roles*): The *Connectors* stands for the set of all connector instances in the configuration. The $Roles^c$ is the set of all roles that the connector instance $c \in Connectors$ has.

Notation: ca_p^c denotes the clearance assigned to the port $p \in Ports^c$ of a component instance $c \in Components$.

Notation: $ReceivedSet_p^c$ [$SentSet_p^c$] denotes a set of security labels associated with the input [output] port $p \in Ports^c$ of a component instance $c \in Components$.

Definition (*Conforming received (sent) set*): A *conforming received [sent] set* associated with the input [output] port $p \in Ports^c$ of a component instance $c \in Components$ is the set of labels (lc], the principle order ideal generated by lc ([lc), the principle order filter generated by lc), where lc is the label which ca_p^c is associated with. A member of a conforming received [sent] set is called a *conforming label*.

Definition (*Port event*): Let L be $ReceivedSet_p^c$ [$SentSet_p^c$] for an input [output] port $p \in Ports^c$ of a component instance $c \in Components$. Then for each $\ell \in L$ we associate an input [output] event type (action) on port p where c receives [sends] a datum with label ℓ through the port p. An input (output) event type on port p with label ℓ is called a *conforming input [output] event type* if its associated label is conforming (i.e. $ca_p^c \geq \ell$ [$ca_p^c \leq \ell$]). An input [output] event type on a port p with label ℓ is an action expressed in CSP as *p?x [p!e]* where p is regarded a channel, and x is a variable whose contents have label ℓ (e is an expression

whose value has label ℓ). An (conforming) input or output event type is called a *(conforming) port event type*.

Note that we distinguish between event types (actions as a part of a CSP term) and events, which are instances of event types. An event refers to an occurrence of an action, such as input or output by a port, while an event type refers to a class of events, such as those involving a particular port and a datum with a particular security label. Thus, corresponding to an action *p?x (p!e)* there can be many input [output] events on port *p* in a trace.

Notation: $\ell(e)$ denotes the label associated with a port event (type) *e*. More specifically, $\ell(e)$ is the security label of the input [output] datum if *e* is an input [output] event (type).

Definition (*received label set assignment, rlsa)*: A received label set assignment (*rlsa*) of a configuration is an indexed collection of *conforming received sets* of each input port of every component instance in the configuration.

Definition (*sent label set assignment, slsa)*: A sent label set assignment (*slsa*) of a configuration is an indexed collection of *conforming sent sets* of each output port of every component instance in the configuration.

A pair of *rlsa* and *slsa* constitutes a *label set assignment (lsa)* for the configuration: *lsa=(rlsa, slsa).*

Definition (*GRLSA*): GRLSA (Global Received Label Set Assignment) is a set of all received label set assignments (*rlsa*s) for the configuration. Note that GRLSA is a finite set.

Notation (*Projection*): The projection of a received [sent] label assignment *rlsa* [*slsa*] on a component instance $c \in Components$ is denoted $rlsa^c$ [$slsa^c$].

Thus, $Proj^c\ rlsa = rlsa^c$, where $Proj^c$ is the projection operator. Similarly, $Proj^c\ slsa = slsa^c$. Hence, the product of received [sent] label set assignments is a collection of $rlsa^c$ ($slsa^c$) sets indexed by $c \in Components$.

$slsa = \prod_{c \in Components} slsa^c$, and

$rlsa = \prod_{c \in Components} rlsa^c$, where $\prod$ is the product operator.

Definition (*Data source ports)* : The set of *data source ports* of a port $p \in Ports^c$ of a component instance $c \in Components$, denoted DSP_p^c, is a set of ports that potentially supply data to the port p (directly through some connection in which p plays a role). More formally, $q \in DSP_p^c$ if and only if both p and q play roles in some connection. By considering the attachment entries of the configuration, the parser determines the data source ports for every port of each component instance, based on *port adjacency*.

To illustrate the port adjacency concept, consider the example configuration in Figure 2. A pair p_i,r_j in the figure pair indicates that port p_i plays the role r_j in some connection it is attached. The port adjacency in this configuration has four entries, one for each port. Note that the connector N_1 can be viewed as a "hyper-edge" with three nodes, where each node is a port playing a role in N_1 . The $DSP_{p_1}^{c_1}$ includes p_1, p_2 and p_3 (through connection N_1), and, p_1 and p_4 (through connection N_2). Port p_1 plays the role r_1 in the connection N_1 and the role r_5 in the connection N_2. Note that a port that is both an input and output port is a data source for itself in any connection.

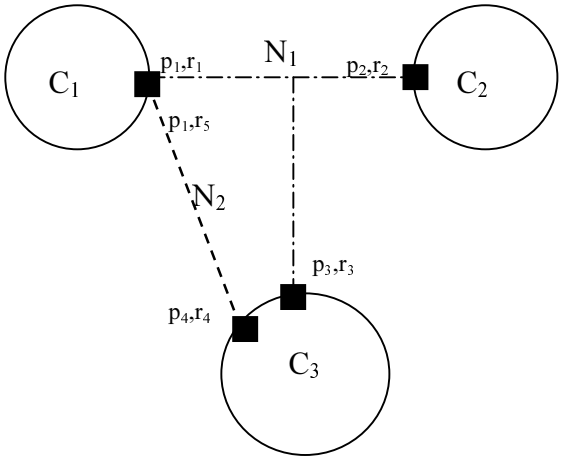

Figure 2: Port Adjacency in a Wright/c configuration

Port p_4, on the other hand, has two data source ports, p_1 and p_4, which plays the role r_5 and r_4, respectively, in the connection N_2. Note that we do not consider at this stage, whether input or output are realized on the related ports, rather, consider the *potential* for each port.

Definition *(Flow Anomalies):* Two types of flow anomalies are defined in regards to the BLP principles:

i. Through some connector a datum with security label ℓ is sent to an input port p with clearance not dominating ℓ. (Hence if the datum were to be received by the port p, that would be a violation of the BLP 'no-read up' principle).
ii. Some component attempts to send a datum with security label ℓ through one of its ports, say p, with clearance not dominated by ℓ. (Hence if the port p were to send the datum, that would be a violation of the BLP's 'no-write down' principle.)

Note that type (i) anomalies arise from the glue of some connector, and type (ii) anomalies arise from the computation of some component.

3.3 Static Analysis of CSP Expressions

Wright configurations (thus, Wright/c configurations as well) involve CSP expressions for the behavioral aspect of their description. Therefore, the analysis of data flow (strictly speaking, the flow of security labels of data) through the ports boils down to an analysis of these CSP expressions. A function called *CSPAnalysis* performs this analysis by extracting the data flow dependencies among the channels of the CSP expression.

The *CSPAnalysis* function takes a CSP expression (computation or glue) along with a list of input and output channels (ports) that appear in it, and the (received or sent) label set assignments for the (input or output) channels.

- In case the *computation* of a component c is analyzed in the function, the input label set assignment is the *received label set assignments* of the ports of the component c.

Then the output is $SentSet^c$.

- In case the *glue* of a connector is analyzed in the function, the input label set assignment is the *sent label set assignments* of the ports playing some role in the connection. Then the output is a *ReceivedSet* for each of these ports.

We need to discuss the analysis of a Computation (*i.e. rlsa* computation) only, as the Glue has the same structure as a CSP expression.

CSPAnalysis : $rlsa^c \rightarrow SentSet^c$, for $c \in Components$

The domain and codomain are both collections of security label sets indexed by the ports of the component instance.

CSPAnalysis, in addition to *SentSets*^{c}, provides information about the trustworthiness of the component $c \in Components$. An output of *'the component must be trusted'* is produced if some data with security label, say ℓ_l, is output through a port and these data is obtained from some input data with security label ℓ_h such that $\ell_l < \ell_h$. If this is the case, the component is assumed to lower the secrecy level of data, for example by applying an encryption, and it must be trusted.

3.4 The Verification Procedure

The verification procedure consists of three phases that are executed sequentially: preprocessing and initialization, data flow analysis, and postprocessing. A data flow diagram of the verification procedure is presented in figure 5.

3.4.1 Preprocessing and Initializing

Taking the abstract syntax of the Wright/c description and the access control lattice as inputs, the verification procedure starts by initializing the *label set assignment (lsa)* of the configuration (although logically included in the verification procedure, it is implemented as a part of the parsing process).

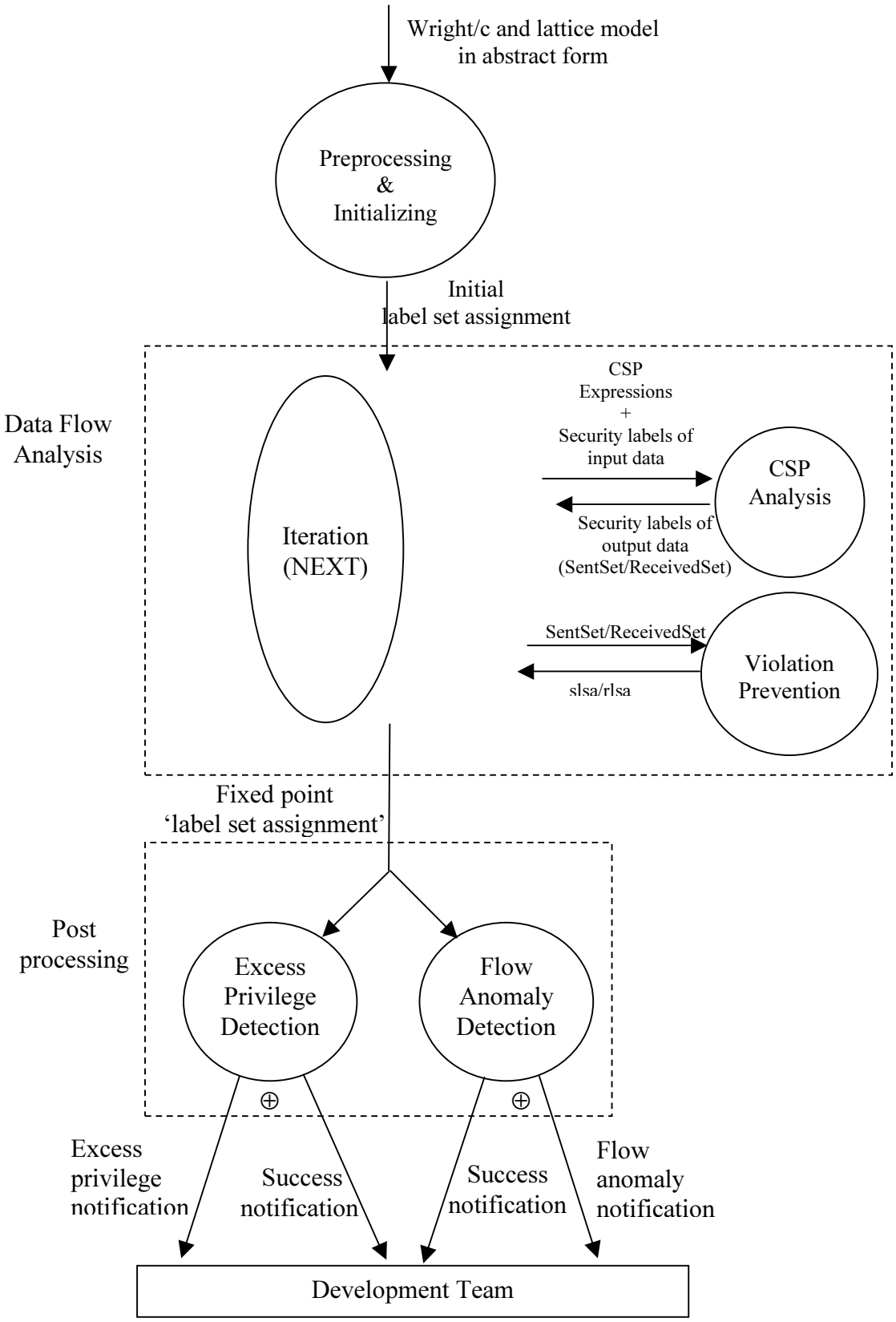

Figure 5: Data flow diagram of the Verification Procedure

The initial *rlsa* is set up by "flooding": All conforming labels are offered to all input ports.

Another task of preprocessing is to determine the *data source ports* for each port of every component instance in the configuration.

The next step, data flow analysis, is a fixed point computation. It can be viewed as an iterative improvement on the initial *lsa.*

3.4.2 Data Flow Analysis

The data flow analysis repeatedly executes the NEXT function until two successive *rlsa*s coincide (i.e. the fixed point *rlsa* is obtained). The *NEXT* function maps an *rlsa A* to a new *rlsa A′* in an iteration step, i.e. *A′=NEXT (A).*

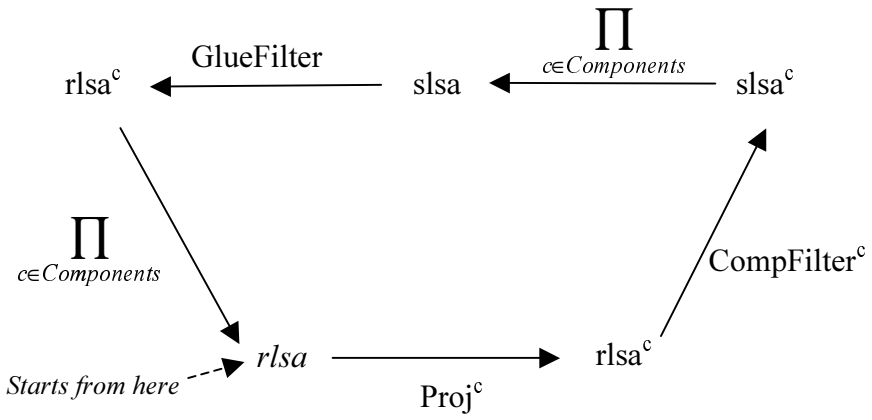

Figure 6: An iteration of Data Flow Analysis realizing the NEXT function.
Iteration starts from *rlsa*.

The NEXT function calls *CompFilter* and *GlueFilter* functions alternatingly (figure 6). The descriptions of the *GlueFilter* and the *CompFilter,* which are parts of the data flow analysis, are given below together with the descriptions of two auxiliary functions, namely, *CSPAnalysis* and *ViolationPrevention*.

CSPAnalysis: Presented in Section 3.3.

ViolationPrevention:

The *ViolationPrevention* function removes all security labels that can cause a BLP-violation from the given security label set. It, then, returns the resulting label set assignment, *SentSet* or *ReceivedSet*.

CompFilter:

It is a function that applies the following operations to perform a data flow and violation prevention analysis within every component instance *c*:

- The function inputs a received label set assignment for the component *c* ($rlsa^c$).
- It invokes *CSPAnalysis* by supplying the computation of *c*, and $rlsa^c$. The *CSPAnalysis* module returns $SentSet^c$.
- It invokes *ViolationPrevention* by supplying $SentSet^c$ and the access control lattice. The module checks the security labels of data potentially output by *c* and authorization levels assigned to its ports (ca_p^c, where $p \in Ports^c$). Then, it returns $slsa^c$ which excludes those security labels that can cause a violation of BLP's 'no-write down' principle. The excluded security labels are put into a list, called *refused sent list.* Note that earlier *refused sent list*s are discarded.
- The function *CompFilter,* then, returns $slsa^c$, the sent label set assignment for the component instance *c*.

In summary, $slsa^c = CompFilter(rlsa^c)$, for each $c \in Components$

GlueFilter:

It is a function that performs a data flow and violation prevention analysis for every component instance $c \in Components$ through the connectors it is incident on:

- *GlueFilter* inputs a sent label set assignment (*slsa*) for *c*.

- It, then, extracts the *data source ports,* DSP_p^c for each port *p* of *c.* Note that a port can play roles in multiple connections. As *GlueFilter* runs on connector basis, the outputs (i.e. DSP_p^c)for the same port playing multiple roles are unioned.
- For each connector instance *a,* the *CSPAnalysis* module is called by supplying the glue of *a*, the ports *p* that play some role in *a*, and $slsa_z^T$ for each $z \in DSP_p^T$, where *T* is a component instance to which *z* belongs. The *CSPAnalysis* function returns $ReceivedSet_z^T$ for each $z \in DSP_p^T$, which are subsequently unioned over *z*.
- The *ViolationPrevention* module is called by supplying $ReceivedSet^c$ and the access control lattice. The module checks the security labels of data potentially input by *c* and authorization levels assigned to its ports (ca_p^c, where $p \in Ports^c$). Then, it returns $rlsa^c$ which excludes the security labels that cause a violation of the BLP 'no-read up' principle. The excluded security labels are put into a list, called *refused received list.* Previous *refused received list*s are discarded.
- The function *GlueFilter,* then, returns received label set assignment for the component instance *c,* $rlsa^c$.

In short, $rlsa^c = GlueFilter(slsa^c)$, for each $c \in Components$

Having described the functions that the NEXT function utilizes, the following paragraph describes the operations NEXT performs on *rlsa* to compute *rlsa'* (figure 6):

- Applies the Projection operator *Proj* to *rlsa* to extract $rlsa^c$ for every $c \in Components$
- Runs *CompFilter* to obtain $slsa^c$ for every $c \in Components$
- Applies the product operator $\prod$ to all $slsa^c$ to form *slsa,* where $c \in Components$. That is, $slsa = \prod_{c \in Components} slsa^c$.
- Runs *GlueFilter* to obtain $rlsa^c$, for every $c \in Components$

- Finally, applies the product operator to obtain *rlsa.*

$$rlsa = \prod_{c \in Components} rlsa^c .$$

To represent and analyze the iteration cycle more formally, we express the function NEXT: GRLSA → GRLSA as a composition of constituent functions:

NEXT $= \prod_{rlsa}$ o GlueFilter o $\prod_{slsa}$ o CompFilter^c o Proj^c, where

Proj^c : rlsa $\mapsto$ rlsa^c, for $c \in$ *Components*

CompFilter^c : rlsa^c $\mapsto$ slsa^c, for $c \in$ *Components*

$\prod_{slsa}$: {slsa^c | $c \in$ *Components*} $\mapsto$ slsa

GlueFilter : slsa $\mapsto$ rlsa′ ^{c} for $c \in$ *Components*.

$\prod_{rlsa}$: {rlsa′ ^{c} | $c \in$ *Components*} $\mapsto$ rlsa′

Having presented a single iteration to compute *NEXT(rlsa),* figure 7 shows the overall data flow analysis phase in terms of *rlsa* transitions. As seen in the figure, each iteration ends up with an *rlsa,* which is the new received label set assignment obtained from the *rlsa* of the previous iteration. The data flow analysis phase terminates when two successive *rlsa*s are the same, i.e., the NEXT function yields no change in the security label sets. Consequently, the data flow analysis returns the fixed point received label set assignment which is, then, subjected to analysis by the portprocessing operations.

3.4.3 Postprocessing

Upon termination of the data flow analysis, the verification procedure moves on to produce reports by checking the resulting label set assignments. The reports can be grouped into three:

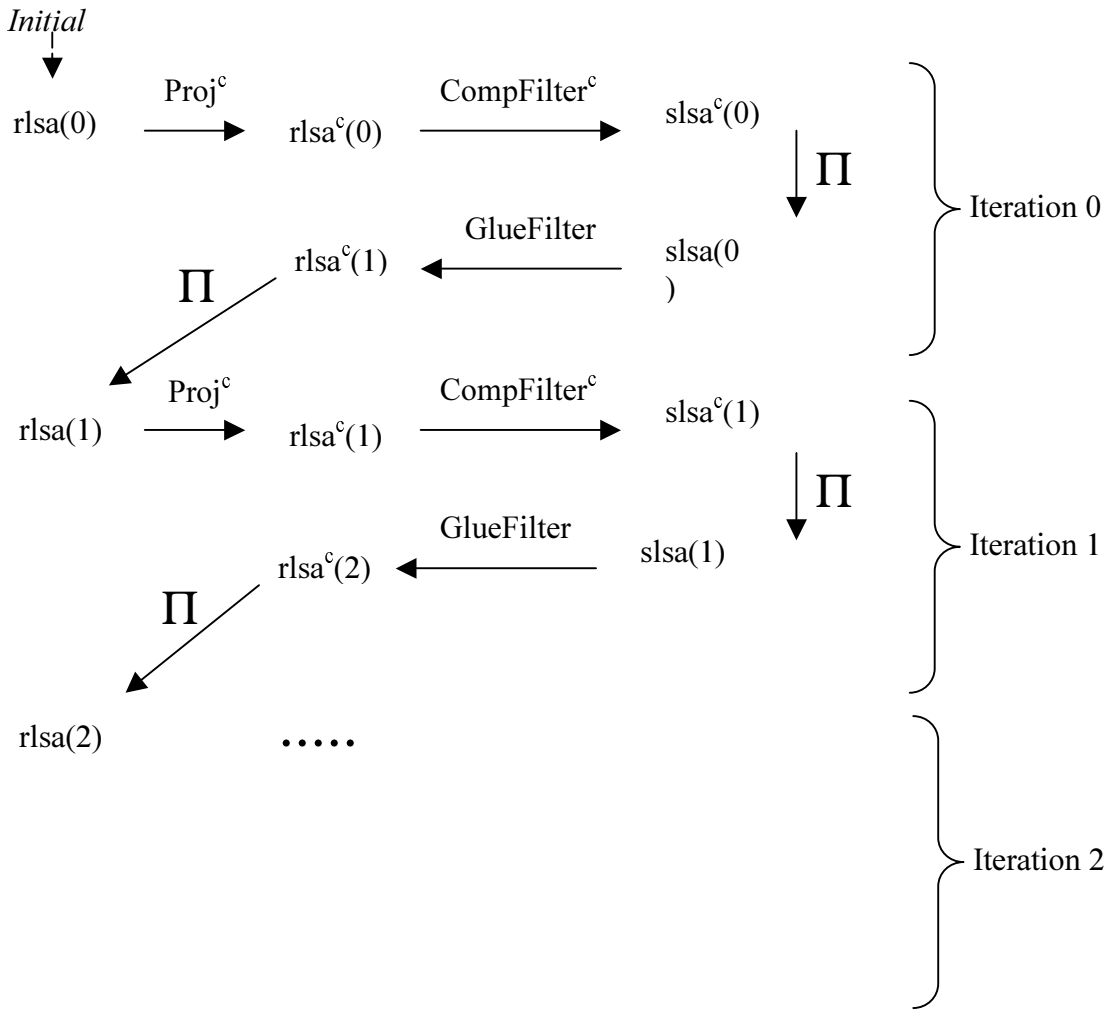

Figure 7: lsa development in the Verification Procedure

i. Flow Anomaly Detection:

It is the function that checks whether an assignment of clearance to the ports of the components in a configuration has any data flow anomalies, granted that the BLP principles are not to be violated. Our view is that in a configuration with a proper clearance assignment, no input port must be sent data that it cannot receive, and no output port must get data that it cannot send without violating the BLP principles.

For detection, the refused lists (*sent refused list, received refused lists)* are checked if there are entries in them. An anomaly is detected in the configuration if,

- *Sent refused list* contains security labels. These labels are in the list because some port *p* can try to send a datum with one of these labels where the *clearance*

assignment (ca_p^c) is not dominated by the label. Though the list entries are discarded in each subsequent iteration, the existence of them at the end indicates a potential violation of BLP '*-property'. The computation of c offers to output port $p \in Ports^c$ with clearance ca_p^c some datum labelled ℓ such that $\ell \geq ca_p^c$ does not hold.

- *Received refused list* contains security labels. These labels are put into the list because they are attempted to be input by a port whose *clearance assignment* (ca_p^c) does not dominate these labels. Though the list entries are discarded in each iteration, the existence of them at the end indicates potential violation of BLP 'simple-security property'. The glue of some connector offers input port $p \in Ports^c$ with ca_p^c some data labelled ℓ such that $\ell \leq ca_p^c$ does not hold.

ii. Excess Privileges

Excess privilege can be defined as the privilege (clearance) that cannot be exercised under a given clearance assignment for the configuration. Hence, if some input port p has clearance K but can not be sent (through some connector to which it is attached with an inputting role) a datum with label ℓ dominated by K, then K is an excess privilege. Thus, there *might* be some lower clearance than K to be assigned to p without restricting the existing data flow within the configuration. Similarly, if some output port p has clearance K but p can not get (from the computation of its own component) a datum with label ℓ which dominates K, then K is an excess privilege. Thus, there might be a higher clearance than K (defined in the access control lattice model) to be assigned to p without restricting the existing data flow within the configuration.

In order to determine whether excess privileges exist, every port p of each component instance c is checked in terms of their clearance assignment (ca_p^c) against its the label set

assignment (lsa_p^c). If there is another clearance assignment, say $ca_p^c{}'$, which is dominated by ca_p^c, and does not cause any decrease in existing data flow then the designer is reported about this situation.

If either, or both, of these cases occur, the verification procedure warns the users to consider the anomalies and revise the clearance assignment, and perhaps, the access control lattice. The report, in general, contains information on component instance basis. For every component instance, the ports are reported as:

- The component instance name it belongs,
- Port name
- Clearance assigned,
- Security labels of data that are potentially input by the port,
- Security labels of data that are potentially output from the port,
- Security labels of data that potentially cause anomalies with respect to BLP's simple security property,
- Security labels of data that potentially cause anomalies with respect to BLP's *-property,
- Excess privileges (if any) by recommending more suitable privilege (if any) in place of its assigned clearance,
- Trustworthiness of each component instance.

iii. Successful Verification

If no flow anomalies and no excess privileges are detected, a message stating the completion of successful verification is announced.

3.5 Programmer's View of the Verification Procedure

The following pseudo-code presents the "programmer's view" of the verification procedure described above.

It can be clearly identified that the steps S0 through S2 are the preprocessing phase, step S3 is the iteration phase, and the remaining steps constitute the postprocessing phase.

3.6. Illustration of the Verification Procedure for Secure Print Server

In this section, an illustration of the verification procedure including iteration steps is presented. The variations of the Secure Print Server [43], to show possible violations is also included in the illustration.

The reader is supposed to refer to the following remarks in order to follow the illustration tables given in this section:

- Each row of the tables depicts state transitions for a port of a component instance,
- A state consists of a *Received Label Set Assignment (rlsa)* and a *Sent Label Set Assignment (slsa). rlsa* of the first state is constructed by applying the *flooding*,
- Each *rlsa* or *slsa* entry is a list of *sublattices* that are represented by their maximum and the minimum elements.
- *P* and *S* stand for PUBLIC and SECRET data labels, respectively. Therefore, an entry ‘P S’ denotes a sublattice with the maximum element *S,* and the minimum element *P.* A single data label is represented by either ‘P P’ (PUBLIC) or ‘S S’ (SECRET). An entry valued as *Empty* refers to an empty list.
- The strikethrough over a label denotes the *refusal* because of the violation prevention with respect to BLP model,
- The bold face emphasizes the updates during the state transitions, and if there appears no change in either *rlsa* or *slsa* lists, the procedure stops,
- At the termination of the procedure, if there is a potential violation to BLP principles, the *rlsa* or *slsa* entry that causes the violation is shaded.

S0. Set *IterationCount* to *0*

‘ Step S1 determines the initial *rlsa* by *flooding.* Initial *rlsa* consists of all data
‘ security labels corresponding the conforming input port events

S1. *input*: none
output: rlsa(IterationCount).

‘ Step S2 extracts the *Data Source Ports* for each port of every component instances

S2. Extract DSP_p^c *for each* $p \in Ports^c$ *and* $c \in Components$

‘ Step S3 performs the *Data Flow Analysis*

S3. *WHILE (IterationCount=0) OR (rlsa(IterationCount)* $\neq$ *rlsa(IterationCount-1)) DO*

S3.1. Apply the Projection operator $Proj^c$ *to rlsa(IterationCount)*
output: $rlsa^c$*(IterationCount) for each* $c \in$ *Components.*

S3.2. CALL *CompFilter for each* $c \in$ *Components*
input: $rlsa^c$*(IterationCount)*
output: $slsa^c$*(IterationCount)*

S3.2.1. CALL *CSPAnalysis*
input: the computation expression of c, and $Ports^c$
output: the sets of security labels ($SendSet^c$*),*
trustworthiness of c (1: trusted, 0: Not trusted).

S3.2.2. CALL *ViolationPrevention*
input: $SentSet^c$
output: $slsa^c$*(IterationCount) together with Refused labels in SentRefusedList.*

S3.3. *Apply the product operator* Π*;*
input: $slsa^c$*(IterationCount), for each* $c \in$ *Components;*
output: slsa(IterationCount).

S3.4. Set $ReceivedSet_p^c$ to *{}, for each* $p \in Ports^c$ *and* $c \in$ *Components*

S3.5. CALL *GlueFilter for each* $n \in$ *Connectors*
input:slsa(IterationCount)
output: $ReceivedSet_p^c$ *for each* $p \in Ports^c$ *and* $c \in$ *Components*

S3.5.1. *FOR each role* $r \in Roles^n$ *DO*

S3.5.1.1. Retrieve *the set of ports* $P_r^c \subseteq Ports^c$ *playing the role r from the attachments, for each* $c \in$ *Components.*

S3.5.1.2. Retrieve *the data source ports* DSP_p^c *and their entries from the slsa(IterationCount), where* $p \in P_r^c$*, for each* $c \in$ *Components.*

END FOR

S3.5.2. Set $TempReceivedSet_p^c$ to $ReceivedSet_p^c$,
for each $c \in$ *Components and* $p \in Ports^c$

S3.5.3. Call *CSPAnalysis*
input: the glue of n, DSP_p^c*, where* $p \in P_r^c$*,* $r \in Roles^n$*,* $c \in$ *Components;*
output: $ReceivedSet_p^c$*, for each* $c \in$ *Components.*

S3.5.4. Set $ReceivedSet_p^c$ to $ReceivedSet_p^c \cup TempReceivedSet_p^c$,
for each $c \in$ *Components and* $p \in Ports^c$

S3.6. *CALL ViolationPrevention;*
input: $ReceivedSet^c$*;*

Figure 8: Programmer’s View of the Verification Procedure

output: $rlsa^c$(IterationCount+1) together with Refused labels in ReceivedRefusedList, for each c∈ Components.

S3.7. *APPLY the product operator Π;*

input: lsa^c(IterationCount+1);

output: rlsa(IterationCount+1), where c∈ Components.

S3.8. *IterationCount = IterationCount + 1*

END WHILE

‘ Step S4 checks for *the simple security property* and ***-property* for each port of
‘ every component instances on the results of Data Flow Analysis

S4. *FOR each port p of every component instance c DO*

S4.1. *IF there exists any security label in the $ReceivedRefusedList_p^c$ THEN*

S4.1.1. *DISPLAY “Potential No-read up violation is detected for port” p “of the component instances” c*

END IF

S4.2. *IF there exists any security label in the $SentRefusedList_p^c$ THEN*

S4.2.1. *DISPLAY “Potential No-write down violation is detected for port” p “of the component instance” c*

END IF

END FOR

‘ Step S5 checks for *the excess privileges*, if any exists, for each port of every
‘ component instances

S5. *FOR each port p of component instance c DO*

S5.1. *CALL ExcessPrivilege;*

input: ca_p^c and lsa_p^c, where c∈ Components and p∈ $Ports^c$

output: a clearance assignment ca_p^c'.

S5.1.1. *IF ca_p^c' <> ca_p^c THEN*

S5.1.1.1. DISPLAY *“Excess Privilege is detected for port” p “of the component instances” c*

S5.1.1.2. *DISPLAY ca_p^c' “can be assigned instead of “ ca_p^c*

S5.1.2. *ELSE*

S5.1.2.1. *DISPLAY “No Excess Privilege was detected”*

END IF

END FOR

‘ Step 6 displays a message stating that the *component must be trusted* if it was determined so
‘ in the last (stable) iteration of CSP Analysis in CompFilter.

S6. *FOR component instance c DO*

S6.1. *IF trustworthiness of c = 1 THEN*

S6.1.1. DISPLAY *“Component “ c “ must be TRUSTED”*

END IF

END FOR

‘ Step 7 displays the successful verification message if no violation is observed

S7. *IF ReceivedRefusedList is empty AND SentRefusedList is empty THEN*

S7.1. *DISPLAY “SUCCESSFUL VERIFICATION”*

ENDIF

Figure 8: Programmer’s View of the Verification Procedure *(continued)*

Table 1 illustrates that the algorithm executes for three iteration steps since the content of the *rlsa* in step 3 does not change. The procedure starts by flooding the *rlsa* regarding the clearance of the ports. Next, the *slsa* is computed for the initial step. Note that PUBLIC labels for *OutputP* port of *PS*, and *Receive* port of *PUBLICPRINTER* are refused by the violation checking algorithm since their clearance are not proper.

Since there is no refused label left after termination of the procedure (i.e. after state is completed), the verification results with a success. That means, statically, there is no data flow that may potentially cause a violation of Bell LaPadula principles.

The description written above agrees with Bell LaPadula principles for preserving confidentiality:

- No read up: the connections are established such that the documents are not directed to an unintended port. Moreover, the server reads secret data only from its authorized port and assumes that it is not public. Therefore, it selects the Secret printer, which is located in a secure room, for the output of documents read from authorized port. Then, unauthorized persons can not access these printouts of secret documents.
- No write down: The server computation is designed so that it does not produce secret printouts using the public printer which everybody can access.

If the description is created disregarding the principles, there may appear some situations that may cause violations for confidentiality. The algorithm that we propose detects potential violations and produces warnings for the developers.

Improper clearance might be assigned to the ports while instantiating the components

1.a. Components are instantiated and their ports are given clearance before attachments are made. In order to fulfill the rules of principles, sufficient clearance must be associated with the port instances. Otherwise, a component instance can try to read a secret document through

an unauthorized port. Moreover, a component instance can try to write secret data through a public port. Both of these cases violate confidentiality principles. For example, in our Secure Print Server, assume that U_A is associated with a clearance value AUTHORIZED. Their ports, by default, inherit this clearance. In such a case, U_A (or its port *PrintP* in particular) will try to write public data which violates the *no write-down* rule of BLP model.

Table 1. Illustration of the iteration steps for the Secure Print Server

Component Instance Name	**Port Name**	**Clearance of the Port**	**STEP 1**		**STEP 2**		**STEP 3**	
			rlsa (***Flooded***)	**slsa**	**rlsa**	**slsa**	**rlsa** *(no change)*	**slsa**
U_1	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	
U_2	PrintS	AUTHORIZED	P S	S S	*Empty*	S S	*Empty*	
U_2	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	
PS	RequestS	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	
PS	RequestP	EVERYONE	P S	*Empty*	P **P**	*Empty*	P P	
PS	OutputP	EVERYONE	P S	P S	*Empty*	P **P**	*Empty*	
PS	OutputS	AUTHORIZED	P S	~~P~~ S S	*Empty*	S S	*Empty*	
SECURE PRINTER	Receive	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	
PUBLIC PRINTER	Receive	EVERYONE	P S	*Empty*	P P ~~S~~	*Empty*	P P	

1.b. Another case for improper clearance might be originated from the behaviour of the component, i.e. from its computation. For example, an authorized component instance can lower the secrecy label of some datum and output through one of its ports. However, if the clearance of the port for such an output is not considered in parallel to this action, a violation to BLP appears. Assume that the computation part of component *PrintServer* is modified as below:

$$\text{Computation} = \text{RequestP.Request?x} \rightarrow (\overline{\text{OutputS.Print!x}} \rightarrow \text{Computation} \sqcap \overline{\text{OutputP.Print!x}} \rightarrow \text{Computation}) \square \text{RequestS.Request?x} \rightarrow \overline{\text{OutputS.Print!x}} \rightarrow \text{Computation}$$

This modified computation, clearly, violates the BLP model. The server reads a public document from its public port and directs it to be printed by a secure printer using *OutputS* port. Since that port has a AUTHORIZED clearance, it violates *no write-down* principle.

2. Improper attachments

Once the components and connectors are instantiated, their ports are attached to suitable roles in the attachment section. Since ports have their own clearance, the attachments need to be established by respecting the principles. If a high clearance port (say an output port) is attached to a role of a connector whose some other roles are attached to ports (say input ports) with low clearance, a potential violation may appear. For example, assume that our configuration had an attachment like :

Attachments

U_B.PrintS as $CONN_1$.ClientP

PS.RequestP as $CONN_1$.ServerP

U_B is an authorized user. It sends SECRET documents through port *PrintS*. However, on the other side of the connection, the server tries to read this secure document through its *RequestP* port whose clearance is EVERYONE. Therefore, the *no read-up* principle is violated.

3. To ignore the simple security property of Bell LaPadula model in Glue part of a connector.

If the description of the glue unexpectedly changes the secrecy level of data flowing through it, this may also cause a violation. For example, suppose that the glue of PrintConnector is rewritten as below:

Glue = ClientP.request?x → $\overline{\text{ServerP.request!x}}^{\text{SECRET}}$ → **Glue**

The modification says that whatever the secrecy label of data received by the connector is, it is carried to a port as a SECRET data. This may potentially cause a *no read-up* principle violation if the receiving port has the EVERYBODY clearance.

Similar to the original description, their verification steps are depicted in Table 2, Table 3, Table 4, and Table 5, respectively.

In Table 2, the procedure terminates in step 3 where there is no modification in the *slsa.* It is shown that *PrintP* port of component instance U_A violates the '*no-write down*' principle since it tries to output a PUBLIC datum while it is given an AUTHORIZED clearance.

Table 2. Illustration of the iteration steps for Secure Print Server (improper clearance, case 1.a)

Component Instance Name	**Port Name**	**Clearance of the Port**	**STEP 1**		**STEP 2**		**STEP 3**	
			rlsa (*Flooded*)	**Slsa**	**rlsa**	**slsa**	**rlsa**	**slsa** (*no change*)
U_A	PrintP	**AUTHORIZED**	P S	~~P~~ ~~P~~	*Empty*	~~P~~ ~~P~~	*Empty*	~~P~~ ~~P~~
U_B	PrintS	AUTHORIZED	P S	S S	*Empty*	S S	*Empty*	S S
U_B	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
PS	RequestS	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PS	RequestP	EVERYONE	P S	*Empty*	P **P**	*Empty*	P P	*Empty*
PS	OutputP	EVERYONE	P S	P S	*Empty*	P **P**	*Empty*	P P
PS	OutputS	AUTHORIZED	P S	~~P~~ S S	*Empty*	S S	*Empty*	S S
SECURE PRINTER	Receive	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PUBLIC PRINTER	Receive	EVERYONE	P S	*Empty*	P P ~~S~~	*Empty*	P **P**	*Empty*

Table 3 illustrates another case for improper clearance assignment (1.b). In this case, *OutputS* port of *PS* component also causes a potential violation '*no-write down'* principle.

Table 4 is a trace of the verification for improper attachments. It shows the occurrence of the flow anomalies violation when an attachment is modified as given above (Case 2).

RequestP port of *PS* component violates '*no-read up*' principle when such an improper attachment is established.

Lastly, Table 5 shows the case where the glue of a connector causes a potential violation as in described above. The *RequestP* port of *PS* component violates '*no-read up*' principle.

Table 3. Illustration of the iteration steps for Secure Print Server (improper clearance, case 1.b)

Component Instance Name	**Port Name**	**Clearance of the Port**	**STEP 1**		**STEP 2**		**STEP 3**	
			rlsa (*Flooded*)	**slsa**	**rlsa**	**slsa**	**rlsa**	**slsa** (*no change*)
U_A	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
U_B	PrintS	AUTHORIZED	P S	S S	*Empty*	S S	*Empty*	S S
U_B	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
PS	RequestS	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PS	RequestP	EVERYONE	P S	*Empty*	P **P**	*Empty*	P P	*Empty*
PS	OutputP	EVERYONE	P S	P S	*Empty*	P **P**	*Empty*	P P
PS	OutputS	AUTHORIZED	P S	~~P~~ S S	*Empty*	~~P~~ S S	*Empty*	~~P~~ S S
SECURE PRINTER	Receive	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PUBLIC PRINTER	Receive	EVERYONE	P S	*Empty*	P P ~~S~~	*Empty*	P **P**	*Empty*

Table 4. Illustration of the iteration steps for Secure Print Server (improper attachment, case 2)

Component Instance Name	**Port Name**	**Clearance of the Port**	**STEP 1**		**STEP 2**		**STEP 3**	
			rlsa (***Flooded***)	**slsa**	**rlsa**	**slsa**	**rlsa**	**slsa** (***no change***)
U_A	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
U_B	PrintS	AUTHORIZED	P S	S S	*Empty*	S S	*Empty*	S S
U_B	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
PS	RequestS	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PS	RequestP	EVERYONE	P S	*Empty*	P P ~~S~~	*Empty*	P P ~~S~~	*Empty*
PS	OutputP	EVERYONE	P S	P S	*Empty*	P **P**	*Empty*	P P
PS	OutputS	AUTHORIZED	P S	~~P~~ S S	*Empty*	S S	*Empty*	S S
SECURE PRINTER	Receive	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PUBLIC PRINTER	Receive	EVERYONE	P S	*Empty*	P P ~~S~~	*Empty*	P **P**	*Empty*

Table 5 Illustration of the iteration steps for Secure Print Server (invalid glue description, case 3)

Component Instance Name	**Port Name**	**Clearance of the Port**	**STEP 1**		**STEP 2**		**STEP 3**	
			rlsa (***Flooded***)	**slsa**	**rlsa**	**slsa**	**rlsa**	**slsa** (***no change***)
U_A	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
U_B	PrintS	AUTHORIZED	P S	S S	*Empty*	S S	*Empty*	S S
U_B	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
PS	RequestS	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PS	RequestP	EVERYONE	P S	*Empty*	~~S~~ ~~S~~	*Empty*	~~S~~ ~~S~~	*Empty*
PS	OutputP	EVERYONE	P S	P S	*Empty*	***Empty***	*Empty*	*Empty*
PS	OutputS	AUTHORIZED	P S	~~P~~ S S	*Empty*	S S	*Empty*	S S
SECURE PRINTER	Receive	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PUBLIC PRINTER	Receive	EVERYONE	P S	*Empty*	~~S~~ ~~S~~	*Empty*	***Empty***	*Empty*

3.7 Trace Model of the Behavior of Wright/c Descriptions

We utilize the CSP trace model for the behaviour of a Wright/c description as defined by Allen [1] to show the correctness of the verification algorithm given in section 3.5. We also rely on the fact that an event structure model of CSP exists, see [22]. Thus we can refer to the dependence preorder on the events of a CSP process.

Let *Behaviour* be the CSP process derived from the Wright/c description of the architectural configuration under consideration as described by Allen. Consider traces of *Behaviour* restricted to the set of port events, such that each event occurring in a trace is a conforming event (in other words, no port event violates the BLP). We call such sequences *port traces*. More formally, t is called a port trace if $t = s\lceil_{CPE}$ for some s in *Traces(Behaviour)*, where *CPE* is the set of conforming port event types. The notion of a port trace allows us to ignore events occurring inside a component (in a computation) or a connector (in a glue).

Definition (*alt*): One can partition a port trace t into substrings each consisting of only input events or only output events. More specifically, assuming that t contains some input events, we can write $t = O_0 \wedge I_0 \wedge O_1 \wedge \ ... \ \wedge O_n \wedge I_n \wedge O_{n+1}$ for some $n \geq 0$ such that each I_j $(0 \leq j \leq n)$ is a nonempty string of input events, each O_k $(1 \leq k \leq n)$ is a nonempty string of output events, and O_0 and O_{n+1} are (possibly empty) strings of output events. We define $alt(t)=n$.

Definition (*Rank):* Let e be a port event occurring in t. Then e must occur in some I_j if e is an input event, or O_k if e is an output event. We define $rank(e,t)=j$ in the former case, and $rank(e,t)=k$ in the latter case.

Definition (*Normal form):* We say that a port trace t is in *normal form* if it has at least one input event and $rank(e,t)$ is minimum for each e occurring in t. More formally, let u be port trace consisting of exactly the same events as t (i.e. u is a permutation of t). Then $rank(e,t) \leq rank(e,u)$ for every e occurring in t (or u).

Observe that for a port trace t in normal form with $t = t' \wedge I_n \wedge O_{n+1}$ for some $n \geq 1$, the port trace t′ is also in the normal form.

Lemma 1: For any port trace t with at least one input event there is a port trace u in the normal form consisting of exactly the same events as t has.

Proof:

Based on the dependency relation of the event structure model of *Behavior*, t can be shuffled so that each event e is now occurring "as early as possible". More precisely, an input event now occurs in a substring I_j where j is as small as possible without violating the dependency relation; similarly for output events. Note that there cannot be cyclic dependence between two events as the dependency relation is a preorder.

Lemma 2: The first n iterations of the data flow analysis algorithm yield a port trace t in the normal form with $alt(t)=n$, for $n \geq 0$.

Proof:

Consider the first n iterations of the algorithm. Then, the following sequence of label set assignments is formed:

$$rlsa(0),\ slsa(0),\ rlsa(1),\ldots,\ slsa(n-1),\ rlsa(n),$$

where $rlsa(0)$ is the initial *rlsa* obtained by flooding, and $slsa(i)=CompFilter(rlsa(i))$, and $rlsa(i+1)=GlueFilter(slsa(i))$ as described in section 3.4.2. Then listing the input events associated with the members of $rlsa(i)$ in some arbitrary order we obtain the string I_i; similarly, listing the output events associated with the members of $slsa(i)$ in some arbitrary order we obtain O_j. Concatenating these substrings yields the port trace $t= I_0 \wedge O_0 \wedge I_1 \wedge \ldots \wedge O_{n-1} \wedge I_n$.

Note that all events in the same substring occur independently (possibly in different parallel processes). Furthermore, the generation of the dependent events are not delayed. That is, all output events depending directly on the events in I_j and not depending on any event in

some $I_{j'}$ with $j' > j$, occur in O_j; similarly all input events depending directly on the events in O_j and not depending on any event in some $O_{j'}$ with $j' > j$, occur in I_{j+1}. Thus t is in normal form.

Lemma 3: Let t be a port trace in the normal form with $alt(t)=n$. Then, for any input event e (on a port p of component c) occurring in t, $\lambda(e)$ is in $rlsa_p^c(n)$, where $rank(e,t)=n$.

*Proof (*by induction on *alt(t)):*

For the basis, assume $alt(t)=0$. Then, $t=O_0 \wedge I_0$. Suppose e appears in I_0. Then $\lambda(e)$, which must be conforming, is in $rlsa_p^c(0)$ by initialization (flooding offers to all input ports all conforming labels). Suppose now $alt(t)=n+1$ and e occurs in I_{n+1}. As t is in normal form, all the output events on which e directly depends are in O_{n+1}, and all the input events on which events in O_{n+1} directly depend are in I_n. By inductive hypothesis, the labels of all events in I_n are in $rlsa(n)$. Then $rlsa(n+1)=NEXT(rlsa(n))$ contains all input events that follow the input events in $rlsa(n)$, in particular e.

Definition (*Infinite traces):* An infinite trace is a member of Σ^{ω}, where Σ is an action alphabet, the sequences of the form $<a_i, i \in N>$, which represents a complete (i.e., throughout all time) communication history of a process that neither pauses indefinitely without communicating nor terminates [30].

The CSP modeling of Wright/c configuration behaviour, as described above, can be analyzed using its *infinite traces.* Our approach to the verification adopts this semantic model. We also consider finite prefixes of infinite traces, that is, traces in the ordinary sense.

3.7.1. The Correctness of the Verification Procedure

In this section, the correctness of the verification procedure will be established. The correctness of the result of the iteration phase will be analyzed in terms of soundness and completeness. Soundness helps to ensure “no false alarms”, and completeness helps to ensure “no missed alarms” when the flow anomalies are reported.

Definition (*Complete Lattice):* A lattice $\mathcal{L}$ is a *complete lattice* is every subset S of $\mathcal{L}$ has both the least upper bound ($\sqcup S$) and the greatest lower bound ($\sqcap S$). [30].

We also use *dual lattices*. Given a lattice $(\mathcal{L},\leq)$, the dual lattice of $\mathcal{L}$ is defined as $(\mathcal{L},\geq)$, that is, one gets the dual lattice of $\mathcal{L}$ by inverting $\mathcal{L}$'s order. The corresponding definition is also quite simple: one only has to exchange the operations $\sqcap$ and $\sqcup$, that is given such a lattice $(\mathcal{L}, \sqcap, \sqcup)$, its dual lattice is $(\mathcal{L}, \sqcup, \sqcap)$. It is clear that dual of a complete lattice is also a complete lattice.

Definition: A binary relation on received label set assignments is defined as follows. Let A and B be any two received label set assignments, that is, $A, B \in GRLSA$. Then,

$(A \sqsubseteq B)$ if and only if $(A_p^c \subseteq B_p^c)$ holds for all $c \in Components$ and for every $p \in Ports^c$.

Lemma 4: (GRLSA, $\sqsubseteq$) is a complete lattice with respect to the order $\sqsubseteq$ as defined above.

Proof:

In order *(GRLSA,* $\sqsubseteq$*)* to be a complete lattice:

i. *(GRLSA,* $\sqsubseteq$*)* must be a poset,

ii. Each subset S $\in$ GRLSA must have a least upper bound and a greatest lower bound in GRLSA.

To see (i) we note that the $\sqsubseteq$ relation on *rlsa*s inherits partial order properties from the subset relation on the components of *rlsa*s.

To see (ii) we note that each subset S of GRLSA has a least upper bound (written $\sqcup$S) and a greatest lower bound (written $\sqcap$S) given as follows:

$$\sqcup S = \prod_{c \in Components} \prod_{p \in Ports^c} \bigcup_{A \in S} A_p^c$$

$$\sqcap S = \prod_{c \in p} \prod_{p \in} {}_{c} \bigcap_{A \in S} A_p^c$$

As (GRLSA, $\sqsubseteq$) is a complete lattice (GRLSA, $\sqsupseteq$) is also a complete lattice. In the correctness analysis of the verification procedure, we will make use of (GRLSA, $\sqsupseteq$) due to the direction of *rlsa* development during the iteration phase of the verification procedure. Initial *rlsa* consists of all data security labels corresponding the conforming input port events (*flooding)*. It is the maximum element in the original lattice, thus, the minimum of the dual lattice.

3.7.2 Fixed Point Induction

To show the correctness of the algorithm, we apply a fixed point induction principle [25,30]:

$$\frac{\phi \vdash [\bot / x]A,\ \phi \cup \{[c/x]A\} \vdash [F(c)/x]A}{\phi \vdash [fix\ F/x]A} \quad \text{constant } c \text{ not occuring in } \phi.$$

If we think of A as a way of saying that the variable x has some property, then $[\bot/x]A$ says that the property A holds for $\bot$ (initial value for x). The second hypothesis, $\phi \cup \{[c/x]A\} \vdash [F(c)/x]A$, is a way of saying that if A holds for some arbitrary fixed value c, then it holds for $F(c)$. We conclude that the property holds for every element of the set $\{F^n(\bot) \mid n \geq 0\}$, by induction on n. This implies that the property holds for the fixed point of F.

Definition *(Complete partial order)* [30]: A *complete* partial order (cpo) is a partial order in which every directed set has a least upper bound, and has a bottom ($\bot$).

Clearly every complete lattice is a cpo. So, (GRLSA, $\sqsupseteq$) is also a cpo.

Theorem (Tarski's theorem for complete partial orders) [30]: Suppose P is a *complete*

partial order and *f: P → P* is *continuous*. Then *f* has a *least fixed point* given by $\sqcup_{n\geq 0} f^n(\perp)$.

At this point, we show that fixed point induction is applicable to the verification algorithm using Tarski's theorem. According to the theorem, if the set of received label set assignments (*rlsa*) is a complete partial order, then the NEXT function implemented in the algorithm must have a least fixed point. We need to establish that the algorithm's NEXT operator that is applied to an *rlsa* (which corresponds to the function *F* in the fixed point induction principle) is *monotonic*. Then, the stable *lsa* that is reached after applying the NEXT for a finite number of iterations, will be the *greatest* fixed point of NEXT as we find the least fixed point *rlsa* in the dual lattice. (The fixed point we find is the greatest fixed point in the original lattice, and the least fixed point in the dual lattice.)

Lemma 5: *CSPAnalysis* function $f: rlsa^c \rightarrow SentSet^c$ ($f: slsa^c \rightarrow ReceivedSet^c$) is *monotonic,* where $c \in Components$.

Proof:

Let *P* be a CSP expression and *po* be an output channel. Call the input channels, the data input from which affects the data sent to *po* as $pi_1, pi_2, \ldots, pi_k$. Let $S(pi_i)$ be the set of data security labels input from pi_i and *S(po)* be the set of the data security labels output to *po*. Generally, *S(po)* depends on *P*, and $S(pi_i)$:

$S(po) = f(S(pi_1), S(pi_2), \ldots, S(pi_k))$; the specific *f* depending on *P*.

Then, we need to show that *f* is monotonic on subset order in every parameter.

First, observe that the run-time behaviour of *P* does not depend on *S(pi)* as there is no branching that depends on the security label of some data. Second, the outputs to a channel are done only by output events. There are two types of output events:

OUTPUT(port, FIXED(security_label, value_list))

OUTPUT(port, DEFAULT(value_list))

In the first case, the security label of the data output is fixed (in the Wright/c description) and does not depend on the security labels in value_list.

Let us consider the second case. The DEFAULT construct models some unspecified function application that takes value_list as parameter. Let V_i be the i^{th} value, $S(V_i)$ be the security label of V_i, and $S(o)$ be the security label output to channel.

Clearly, $S(o)$ should only depend on $\{S(V_i)\}$.

CSPAnalyser defines $S(o)$ as:

$S(o) = LUB\{S(V_i)\}$

Before run-time we may not actually know $S(V_i)$'s, instead we will know the set $SET\_S(V_i)$ of security labels that $S(V_i)$ is an element of. Let $AllP$ be the set of all minimal sets that has an element from all $S(V_i)$:

$$AllP = \{Pos \mid (\forall i\ \exists x:\ x \in SET\_S(V_i) \text{ and } x \in Pos) \text{ and } (\forall x: x \in Pos \Rightarrow \exists i: x \in SET\_S(V_i))\}$$

Then, the set $SET\_S(o)$ of the security labels output to the channel will be:

$$SET\_S(o) = \{LUB(Pos) \mid Pos \in AllP\}$$

That is, $SET\_S(o)$ is the minimal set such that for each possible choice Pos of $S(V_i)$ from $SET\_S(V_i)$ for all i, $SET\_S(o)$ has LUB(Pos) as an element.

Now, it is clear that $SET\_S(o)$ is monotonic on every $SET\_S(V_i)$. Without loss of generality suppose:

$SET\_S(V_1) \subset SET\_S(V_i)'$, $SET\_S(V_i) = SET\_S(V_i)'$ for all other i.

Let $$AllP = \{Pos \mid (\forall i\ \exists x:\ x \in SET\_S(V_i) \text{ and } x \in Pos) \text{ and } (\forall x: x \in Pos \Rightarrow \exists i: x \in SET\_S(V_i))\},$$

$$AllP' = \{Pos \mid (\forall i\ \exists x:\ x \in SET\_S(V_i)' \text{ and } x \in Pos) \text{ and } (\forall x: x \in Pos \Rightarrow \exists i: x \in SET\_S(V_i)')\},$$

Then, $SET\_S(o) = \{LUB(Pos) \mid Pos \in AllP\}$, and

SET_S(o)' ={LUB (Pos) | Pose∈ AllP'}

But AllP ⊆AllP'. So, SET_S(o) ⊆SET_S(o)', which completes the proof.

Lemma 6: The NEXT: GRLSA → GRLSA function, defined in the verification procedure, is *monotonic*.

Proof:

Let *A* and *B* be any two elements of GRLSA, where $A \sqsubseteq B$. Let, also, $A' = NEXT(A)$ and $B' = NEXT(B)$. In order NEXT to be a monotonic function, $A' \sqsubseteq B'$ must be satisfied.

Recall that NEXT function can be decomposed as:

$NEXT = \Pi_{rlsa}$ o GlueFilter o Π_{slsa} o $CompFilter^c$ o $Proj^c$

For the NEXT function to be monotonic, it is sufficient that the constituent functions be monotonic. By their definitions, *Proj* and Π have no effects in achieving the monotonicity since they project the output of the function result on component basis and compose them, respectively. Therefore, monotonicity of NEXT function requires:

$CompFilter(A^c) \sqsubseteq CompFilter(B^c)$, and

$CompGlue(A^c) \sqsubseteq CompGlue(B^c)$, where $c \in Components$

These two functions, *CompFilter* and *GlueFilter*, both consists of two phases: the *CSPAnalysis* and the *ViolationPrevention*. The former is monotonic as shown by Lemma 5.

The *ViolationPrevention* refuses (removes) any label from its input label set assignment if it can cause a violation with respect to BLP principles. If some label $\ell \in rlsa$ is filtered out (refused) in A', and if ℓ still appeared in B′, then it must also be filtered out from B', by the definition of the *ViolationPrevention*, which preserves the monotonicity of CSPAnalysis.

Therefore, NEXT function is monotonic.

Having a complete partial order GRLSA, which is finite, and the monotonicity of the NEXT function implies that NEXT is continuous. Then, the least fixed point exists according to the Tarski's theorem.

We now aim to formulate the property which will be the predicate of induction.

Definition: (*Infinitely often):* Let *t* be an infinite trace of a configuration. If at all time instants, there is a future instant that an event of type *e* occurs, then the event type *e* is said to occur in *t infinitely often* (i.o.)[1].

Events occurring i.o. allow us to relate the architecture of the system, which is a static notion, to the behavior of the system, which is a dynamic notion. The behavioral counterpart of a port event e (input/output action e in CSP terminology) in the description is, in our view, an event type e occurring i.o. in some infinite trace of the configuration. To put another way, port events persist. Event types that can occur only finitely many times (events belonging to the transient behavior of the system, such as initializations) are discarded from our analysis. Although transient behavior could be important in analyzing dynamic architectures, we do not attribute any structural significance to them in a static architecture. From an architectural standpoint, an event that is in violation of BLP principles, presumably, will not be allowed to occur, e.g. thanks to the action of a reference monitor. (Note that enforcing of the architecture on the actual software is beyond the scope of this work.) However, if such an event is offered i.o. we recognize this as an anomalous phenomenon; this must be the result of a self-defeating assignment of privileges.

The predicate we develop is based on the infinite traces of the configuration as it refers to i.o. event types. If some event *e* appears in a trace of the configuration, all the events that *e*

[1] In temporal logic, this can be expressed by the formula "G F occurs (e)", which is interpreted on the infinite traces of the CSP process, which is the behavior of the configuration.

depends on (directly or indirectly) must also be in the trace. If e occurs i.o. in some infinite trace then all the events that e depends on must also occur i.o. in the trace.

Definition *(D(s,e))*: Define a predicate *D(s,e)* for an *rlsa s* and event type *e* as follows:

D(s,e): there is an infinite trace *t* (of the configuration) such that all input events in *t* are in (some member of) *s* and *e* occurs infinitely often in *t*. (Intuitively, all events that *e* depend on are in *s*.)

We define the property *A* of the fixed point induction as follows:

A(s) : *rlsa s* contains all conforming input event types *e* that satisfies *D(s,e).*

We see that the property *A* is *admissible* since if it is satisfied in a set *S* of *rlsas,* it must be satisfied by ⊔S of the set as ⊔S is the componentwise union of S.

Lemma 7: *A*(*u*) is satisfied, where *u* is the greatest fixed point *rlsa* of the NEXT function, denoted *u=Fix(NEXT)*.

Proof:

We use the fixed point induction to prove that *A(u)* is satisfied.

Basis (for ⊥): (Note that ⊥ denotes the top element of the dual lattice.) All conforming input events are included in initial *rlsa* denoted by ⊥ as a result of flooding. Therefore, *A(⊥)* is satisfied.

Induction hypothesis: Assume that A(s) is satisfied.

Induction step: Show that A(s′), where s′=NEXT(s), must be satisfied.

Assuming A(s) is equivalent to assuming that *s* includes (the labels of) all event types *e* that satisfy *D(s,e).* Then, either some event that does not satisfy *D(s′,e)* is filtered out from s′ or s=s′ in which case the algorithm terminates thereby obtaining *s′* = *u*. In the latter case we are done. So assume the former case. Let *e* be an event in *s*-*s′*. That is, *e* is refused in some iteration *n*, it does not occur infinitely often (i.o.). To see this, suppose that *e* occurs i.o. Then, *e* occurs in some normal form trace *t* with *rank*(e,t) = k > n. By Lemma 5.3, *ℓ(e)* is in rlsa(k).

Since NEXT function is monotonic and $\ell(e)$ is not in *rlsa*(n+1), then, it can not be in rlsa(k) -- *contradiction.* Therefore, *e* is not infinitely often, which subsequently leads to *D(s-s',e)* is not satisfied.

Therefore, by principle of induction we have *A(u)* satisfied, where *u* is a the least fixed point *rlsa* (in the dual lattice).

At this point we only know by Lemma 7 that greatest fixed point *rlsa u* contains all the port events *e* satisfying *D(u,e).* Next, we must come up with an argument to show that *u* satisfies *D(u,e)* for any *e* contained in *u*.

Lemma 8: Let u be the greatest fixed point of NEXT. Then *D(u,e)* holds for any input event *e* contained in *u*.

Proof:

Let *d* be in *u*. Show that *D(u,d)*. We have *NEXT(u)=u*. Therefore, by an easy induction, *d* is in $NEXT^n(u)$ for any *n*. So *d* is i.o. and *D(u,d)* is satisfied by lemma 7.

Thus we have the following result characterizing the outcome of data flow analysis:

Lemma 9: Let *rlsa u = Fix (NEXT)*, computed by the data flow algorithm. Then *u* exactly contains those input event types *e* occurring i.o. in some infinite port trace *t* whose input events (with the possible exception of *e*) are all in *u*. Similarly, *u* exactly contains those output events *e* occurring i.o. in some *t* whose output events (with the possible exception of *e*) are all in *u.*

Here is our main result:

Theorem: The verification algorithm produces a flow anomaly notification if and only if a flow anomaly exists in the configuration.

Proof: *("only if" part –soundness, or "no false alarms")*

Suppose that the verification algorithm produces a type (i) flow anomaly notification. This is when the detection procedure finds an output port *p* (of some component, say *c*) with

$slsa_p^c(u)$ containing some label ℓ, and a connected input port q (of some component, say d) such that $rlsa_q^d(u)$ does not contain ℓ, as $ca_q^d \not\ni \ell$. Let e be the output event corresponding to label ℓ at port p. Then by lemma 9, e occurs i.o. in some infinite trace t consisting of events in u. As the ports p and q are connected (through some connector) some datum with label ℓ is bound to be offered to port q, by the fairness assumption, but it must be rejected so as not violate the simple security property. This constitutes a data flow anomaly of type (i). Argument for type (ii) anomaly is similar. Hence, the "only if" part.

("if" part –completeness, or "no missed alarms")

Let p be an output port (of some connector *c)* and q be an input port (of some component, say *d).* Let also e be the output event at port *p.* As the ports p and q are connected (through some connector) some datum with label ℓ, corresponding to *e,* is bound to be offered to port *q.* Suppose that a data flow anomaly of type (i) exists. This requires that e does not occur i.o. in some infinite trace t consisting of events in u. Thus, by lemma 9, it must be rejected so as not violate the simple security property. This results in that $slsa(u)$ contains the label ℓ whereas $rlsa_q^d(u)$ does not contain ℓ, as $ca_q^d \not\ni \ell$. This concludes that the verification algorithm produces a type (i) flow anomaly notification. Argument for type(ii) anomaly is similar. Hence, the "if" part.

Thus, the greatest fixed point *rlsa u* will contain *all* and *only* the events e that occur infinitely often in some infinite trace whose input events are all in u. The *all* part establishes that the *lsa* at the greatest fixed point is complete --this ensures "no missed alarms" when the flow anomalies are reported. The *only* part establishes soundness --this ensures "no false alarms".

3.8 Computational Complexity Analysis

In this section, we analyze the worst case running time and space complexity of the verification algorithm, whose programmer's view is presented in section 3.5. The analysis will be discussed in three parts corresponding the phases of the algorithm: the preprocessing, the iteration process, and the post-processing. For the analysis we use the following abbreviations:

c: Number of component instances in the configuration

p: Maximum number of ports in a component instance

C: Number of component type definitions

A: Number of connector type definitions

a: Number of connector instances

r: Maximum number of roles in a connector

sl: Number of security labels in the access control lattice model.

cl: Number of clearance declared in the access control lattice model.

The number of attachments is, then, $(p \cdot c \cdot k_p + a \cdot r \cdot k_r)/2$, where k_p is the number of roles that a port plays, and k_r is the number of ports that a specific role is assigned to be played.

3.8.1 Running Time Complexity

The worst case running time (RT) of the verification algorithm, say *RT(A),* is the sum of the running times of its constituent subprocesses:

RT(preprocessing) + RT(iteration) + RT(postprocessing)

The following parts present the RT of each of these subprocesses.

Preprocessing:

The steps S0, S1 and S2 belong to preprocessing. The step S0 is a single assignment statement and can be ignored. In step S1 (*flooding)*, each (conforming) label in the access control lattice model is assigned to every port $p \in Ports^c$, where c∈ *Components*. Note that the

lattice is represented as a list of edges. Therefore, S1 requires a sequential scan on the ports of each component and assignments of the labels for each of them. So, it has a RT of $p \cdot c \cdot sl$.

The step S2 is used to extract the data source ports (DSP_p^c) for all $p \in Ports^c$ and $c \in Components$. It requires a complete sequential scan on the attachments for each indexed collection of ports of components. So, its RT is $\frac{1}{2} \cdot (p \cdot c \cdot k_p + a \cdot r \cdot k_r) \cdot (p \cdot c)$.

Thus, $RT(preprocessing) = p \cdot c \cdot sl + \frac{1}{2} \cdot (p \cdot c \cdot k_p + a \cdot r \cdot k_r) \cdot (p \cdot c)$

$$= p \cdot c \cdot sl + \frac{1}{2} \cdot p^2 \cdot c^2 \cdot k_p + \frac{1}{2} \cdot a \cdot r \cdot k_r \cdot p \cdot c.$$

Iteration:

The fixed point iteration is the step S3. Since the iteration continues as long as there is a change (removal of at least one label) in two consecutive received label set assignments, the number of iterations can be at most $p \cdot c \cdot sl$. During each iteration, the substeps result the following RTs:

The step S3.1 is the projection operation that simply has a RT of c.

The step S3.2 consists of a call to *CSPAnalysis* function and a call to *ViolationPrevention* function for each component instance. The former has an RT of $k_{io}^2 \cdot p^2 \cdot sl^2$, where k_{io} is a positive constant of proportionality. The *ViolationPrevention* is a sequential traversal on the label set assignments of every indexed collection of the ports, and a comparison of it versus its clearance assignment. So, its RT is $p \cdot sl$, which results that the RT of step S3.2 is $c \cdot (k_{io}^2 \cdot p^2 \cdot sl^2 + p \cdot sl)$.

The step S3.3 is a product operations on each sent label set assignment, which requires a sequential scan on the sent label set assignments. So, its RT is c.

The step S3.4 is just an initialization of *ReceivedSet*s of each port of every component instance. So, its RT is $p \cdot c$.

The function *GlueFilter* is called for each connector instance $cn \in Connectors$ in the configuration in step S3.5. So, it is executed a times. Thus, the step S3.5 yields the RT:

$$a\cdot(r\cdot[\frac{1}{2}\cdot(p\cdot c\cdot k_p + a\cdot r\cdot k_r) + k_r\cdot(\frac{1}{2}\cdot p\cdot c)] + p\cdot c\cdot sl + k_{io}^2\cdot r^2\cdot sl^2 + p\cdot c\cdot sl)$$

$$= \frac{1}{2}\cdot a\cdot r\cdot p\cdot c\cdot(k_p + k_r) + \frac{1}{2}\cdot a^2\cdot r^2\cdot k_r + 2\cdot a\cdot p\cdot c\cdot sl + a\cdot k_{io}^2\cdot r^2\cdot sl^2$$

The step S3.6 is a call to *ViolationPrevention* function for each port of every component instance. So its RT is $p\cdot c\cdot sl$.

The last steps of the iteration phase are S3.7 and S3.8. The former has an RT of c (the product operator), and the latter is a constant value that is ignorable since it is just a simple integer assignment.

Therefore, the running time of the iteration phase, RT(iteration) is:

$$p\cdot c\cdot sl\cdot(3\cdot c + k_{io}^2\cdot c\cdot p^2\cdot sl^2 + 2\cdot p\cdot c\cdot sl + p\cdot c + \frac{1}{2}\cdot a\cdot r\cdot p\cdot c\cdot(k_p + k_r) + \frac{1}{2}\cdot a^2\cdot r^2\cdot k_r + 2\cdot a\cdot p\cdot c\cdot sl + a\cdot k_{io}^2\cdot r^2\cdot sl^2)$$

Postprocessing:

The steps S4, S5, S6, and S7 constitute this part. S4 is a repetition statement executed for each port of every component instance. The two refused label lists (*ReceivedRefusedList* and *SentRefusedList*) are sequentially traversed in each iteration. So, RTs of the step S4 and S5 are $p\cdot c\cdot 2\cdot sl$, $p\cdot c\cdot cl\cdot 2\cdot sl$, respectively. The step S6 is also a sequential traversal on the number of component instant, which is c. Lastly, the step S7 is just a check on two refused lists, so it has a constant RT.

Therefore, the running time of the portprocessing phase is:

$$RT(postprocessing) = p\cdot c\cdot 2\cdot sl + p\cdot c\cdot cl\cdot 2\cdot sl + c = 2\cdot p\cdot c\cdot sl\cdot(1+cl) + c$$

Consequently, the running time of the algorithm is:

$$RT(A) = p\cdot c\cdot sl + \frac{1}{2}\cdot p^2\cdot c^2\cdot k_p + \frac{1}{2}\cdot a\cdot r\cdot k_r\cdot p\cdot c$$

$$+p\cdot c\cdot sl\cdot(3\cdot c+k_{io}^{2}\cdot c\cdot p^{2}\cdot sl^{2}+2\cdot p\cdot c\cdot sl+p\cdot c+\frac{1}{2}\cdot a\cdot r\cdot p\cdot c\cdot(k_p+k_r)+\frac{1}{2}\cdot a^{2}\cdot r^{2}\cdot k_r$$

$$+2\cdot a\cdot p\cdot c\cdot sl+a\cdot k_{io}^{2}\cdot r^{2}\cdot sl^{2})\ +2\cdot p\cdot c\cdot sl\cdot(1+cl)+\ c$$

$$=3\cdot p\cdot c^{2}\cdot sl+\ k_{io}^{2}\cdot c^{2}\cdot p^{3}\cdot sl^{3}+\ p^{2}\cdot c^{2}\cdot(\frac{1}{2}\cdot k_p+2\cdot sl^{2}+sl)\ +\frac{1}{2}\cdot a\cdot r\cdot p^{2}\cdot c^{2}\cdot sl\cdot(k_p+\ k_r)$$

$$+\frac{1}{2}\cdot p\cdot c\cdot sl\cdot a^{2}\cdot r^{2}\cdot k_r+2\cdot a\cdot p^{2}\cdot c^{2}\cdot sl^{2}+\ a\cdot p\cdot c\cdot r^{2}\cdot k_{io}^{2}\cdot sl^{3}+\frac{1}{2}\cdot a\cdot r\cdot k_r\cdot p\cdot c+\ p\cdot c\cdot sl\cdot(3+2\cdot cl)\ +\ c$$

Practically, the number of security labels *sl* and the number of clearance assignments *cl* are very small values compared to the total number of ports and the total number of roles. Moreover, *sl* and *cl*, in principle, are independent of the software architecture. So, they can be considered as constants. Applying these considerations to the result (after simplification):

$RT\ (A)=k_1\cdot a\cdot r\cdot p^{2}\cdot c^{2}+\ k_2\cdot a^{2}\cdot r^{2}\cdot p\cdot c\ +k_3\cdot a\cdot p^{2}\cdot c^{2}\ +\ k_4\cdot p^{2}\cdot c^{2}+\ k_5\cdot a\cdot p\cdot c\cdot r^{2}\ +\ k_6\cdot p^{3}\cdot c^{2}$

$+\ k_7\cdot p\cdot c^{2}+k_7\cdot a\cdot r\cdot p\cdot c+\ k_9\cdot p\cdot c+\ k_{10}\cdot c\ +\ k_{11},$ where k_i*'s, i:1..11,* are the coefficients

Therefore, the running time complexity of the algorithm is polynomial with respect to the number of ports in a component and the number of roles in connector together with the number of component instances and the number of connector instances. The dominating terms are either $k_1\cdot a\cdot r\cdot p^{2}\cdot c^{2}$ *or* $k_2\cdot a^{2}\cdot r^{2}\cdot p\cdot c$ depending on the topology of the configuration. If the number of component instances and the number of ports are larger than those of connector instances and the roles, then the former will dominate, otherwise the latter will. However, letting $n=max(p,c,a,r)$ for the worst case, then the running time complexity of the algorithm becomes $\theta(n^{6}),$ regardless of the topology of the configuration.

3.8.2 Space Complexity

The verification algorithm inputs the Wright/c description and the access control lattice model in an abstract form, which are produced by the parser, using the *list* and *record* data types of ML. The verification algorithm mainly applies basic list and record operations on

these inputs. Therefore, the worst case space complexity (*SC*) of the algorithm depends on the maximum sizes of these data structures used in the algorithm.

To check the utilization of these data structures, we list them below with the maximum sizes that can possibly be allocated using the abbreviations given at the beginning of this section:

- *SentSet:* This list consists of two constituent lists, namely the sent label set assignment (*slsa)* and SentRefusedLabelSet. Both of the lists have an entry for each port of every component (*p·c)*. In each entry, a set of data security labels is stored. There can be a maximum of *sl* labels that can be stored since we assume the worst case. Therefore, the space needed by *SentSet* is *2·p·c·sl*. Note also that, there is no further dynamic allocation for the elements of the list, once they are initialized by the parser.
- *ReceivedSet:* This list is has the same utilization as of *SentList.* It consists of lists of the received label set assignment (*rlsa*) and the ReceivedRefusedLabelSet whose entries are as above. So, the space needed by *ReceivedSet* is also *2·p·c·sl*.
- *Attachments*: For each attachment in the configuration, there is an entry for this list including a component name, a port name, a connector name and a role name. So, the maximum size of the list is $\frac{1}{2}\cdot(p\cdot c\cdot k_p + a\cdot r\cdot k_r)$. Since each attachment entry has 4 elements, the space needed by the list is $2\cdot(p\cdot c\cdot k_p + a\cdot r\cdot k_r)$.
- *Instances*: There are *c*+*a* instances in a configuration. So, the size of this list is *c*+*a.*
- *Components*: We store the definitions of a component types in this list. So we have *C* entries in the list. Each entry stores *p* number of ports and a CSP expression to store the computation in CSP notation. Therefore, the space needed by this list is *C·(p+op),* where *op* represents the number of operations in the CSP expression.

- *Connectors:* Similar to the list of components, this list needs $A\cdot(r+op)$ size of space.
- *Port Adjacency:* This list keeps data source ports for each port of every component instance in the configuration. Totally, there are $p\cdot c$ number of entries in the list. For the worst case, we can take all of the ports of the configuration to be a data source for each port. Moreover, each data source port keeps a set of data security label, which has a maximum size of *sl.* Therefore, the list needs a data space of $(p\cdot c)^2\cdot sl$.

Having determined the spaces needed by each of the lists of Wright/c configurations and styles, the total space is calculated as:

$$SC(Wright)=4\cdot p\cdot c\cdot sl+2\cdot(p\cdot c\cdot k_p+a\cdot r\cdot k_r)+c+a+C\cdot(p+op)+A\cdot(r+op)+(p\cdot c)^2\cdot sl$$

$$=4\cdot p\cdot c\cdot sl+2\cdot p\cdot c\cdot k_p+2\cdot a\cdot r\cdot k_r+c+a+C\cdot p+C\cdot op+A\cdot r+A\cdot op+(p\cdot c)^2\cdot sl$$

Using the same remarks as in the calculation of the running time complexity, the number of security labels ℓ, the number of component type and connector type definitions, and the number of operations in the CSP expressions are practically very small values and they can be considered as constants. Therefore, rewriting the result, after the simplification:

$SC(Wright)=k_1\cdot(p\cdot c)^2+k_2\cdot p\cdot c+k_3\cdot a\cdot r+k_4\cdot p+k_5\cdot r+k_6\cdot c+k_7\cdot a+k_8$, where k_i*'s, i:1..9* are the coefficients. It is clear that the dominating term is $k_1\cdot(p\cdot c)^2$, which is polynomial in terms of the number of ports in a component instance and the number of component instances. Letting $n=max(p,c,a,r)$, and we get the space complexity of a Wright/c description, used by the algorithm, as $\theta(n^4)$.

The verification algorithm also inputs the access control lattice model, which is also stored in some list data types. There are 3 lists allocated to represent the lattice contents:

- *SecurityLabels:* The data security labels declared in the lattice are held in this list. So, it has *sl* entries.

- *ClearanceList:* The authorization level types declared in the lattice with the security labels they dominate are held in this list. The entries are a clearance and a security label it dominates. If there is a distinct clearance declared for each of the security labels, the size of the list becomes the maximum. So, the list has a data space of $2 \cdot sl$.
- *Ordering:* The edges of the lattice are stored in this list pairwise. Since transitive closures are not stored, we have a maximum of $\frac{sl}{2} log_2\ sl$ pairs. So, the size of the list is : $2 \cdot \frac{sl}{2} log_2\ sl = sl\ log_2\ sl$.

Thus, the space complexity of the access control lattice model is:

$SC(Lattice) = sl + 2 \cdot sl + sl\ log_2\ sl$. So, $SC(Lattice) = \theta(sl\ log_2\ sl)$.

4. CONCLUSIONS AND DISCUSSION

In this report, a static verification of a software architecture in terms of data-flow based confidentiality is proposed by relating the studies on software architectures and confidentiality.

For architectural study, Wright architecture description language (ADL) is selected due to its amenability to static analysis. The extended Wright, Wright/c, enables static analysis of data flow over the architecture subject to confidentiality requirements as per ‘no-read up’ and ‘no-write down’ principles of Bell-LaPadula.

The lattice model and the Wright/c desciption can be formulated in XML notation. XML notation facilitates the tasks performed by the parser which produces necessary information in abstract syntax for the verification.

Taking the ADL description in a suitable abstract syntax and the access control lattice model as inputs, a verification procedure that includes a data flow analysis and an anomaly detection process is developed. The verification procedure checks if there is a potential violation with respect to Bell LaPadula principles. For data flow analysis, input and output

data flows through the ports of component instances in the software configuration are analyzed by an algorithm. A violation prevention, which is a part of the data flow analysis, is performed by checking the clearance of ports of component instances against the security labels of data input or output through these ports. The algorithm benefits from a CSP analysis study developed separately.

The verification procedure also reports on excess authorizations. It extracts the minimum required clearance of the constructs (ports) without disturbing the existing data flow over the architecture. Therefore, if it is given a clearance more than needed, the surplus authorization (excessive clearance) can be reclaimed.

A software tool, which features an XML-based front-end to the algorithm is constructed. An implementation of the verification and the CSP Analysis (without parallel || operator) are implemented in ML. Taking concrete syntax of the system configuration and access control lattice model in XML notation as inputs, a front-end processing provides their abstract syntax that is offered to the verification procedure and the CSP Analysis. A potential work could be the integration of these processes in order to provide an easy-to-use interface to the designers.

The worst case computational complexity of the verification algorithm is discussed in terms of running time and space. The running time complexity is polynomial, $\theta(n^6)$, where n is the maximum value out of the number of ports in a component, the number of port instances, the number of roles in a connector, and the number of connector instances. The space complexity, on the other hand, is analyzed for both Wright/c descriptions and the access control lattice model. The former is found as $\theta(n^4)$, where n is taken as the maximum value out of the number of ports in a component and the number of component instances. The latter is calculated as $\theta(nlogn)$, where n is the number of data security labels in the lattice.

This work can be effectively applied to a software system during the design phase of its development. Having an access control model based on the security policy of the institution,

the verification of the Wright/c description of the system helps to see the possible data flow confidentiality violations in advance.

The verification algorithm deals with the confidentiality of a software system. The integrity model, proposed by Biba, can be incorporated using the same way. Similar to the lattice model for confidentialy, an integrity lattice can be constructed and the principles of Biba model can be applied with respect to the integrity lattice. To do that data are grouped in terms of integrity classes, and each class is denoted by an integrity label. Authorization levels (clearance) of the ports are then associated with the integrity lattice to obtain the dominance relation.

REFERENCES

[1] Allen, R.J. A "Formal Approach to Software Architecture". *Ph.D. Thesis Report, School of Computer Science, Carnegie Mellon University,* May *1997.*

[2] Back Johan R. "Wright J.V. Refinement Calculus: A Systematic Introduction". *Springer-Verlag New York Inc. 1998.*

[3] Bai Y. and Varadharajan V. "A High Level Language for Conventional Access Control Models". *Proceedings of the Australasian Conference on Information Security and Privacy. Also in Springer-Verlag's LNCS 1998, pp273-283, Vol.1438, 1998.*

[4] Bai Y. and Varadharajan V. "Access Control: Its Representation and Evaluation". *Proceedings of IFIP/SEC2000 International Information Security Conference, pp 232-235, 2000.*

[5] Bass L., Clements P., Kazman R. "Software Architecture in Practice". *Addison Wesley, 1998.*

[6] Bell D.E. and LaPadula L.J. "Secure Computer Systems: Mathematical Foundations and Model". *M74-244, Mitre Corporation, Bedford, Massachusettes, 1975.*

[7] Biba K.J. Integrity Considerations for Secure Computer Systems". *Mitre TR-3153, Mitre Corparation, Bedford, Massachusetts, 1977.*

[8] Bishop J. Faria R. "Connectors in Configuration Programming Languages: are they necassary?". *Proceedings of the Third IEEE International Conference on Configurable Distributed Systems, Annapolis, pp.11-18, May 1996.*

[9] Bodei C., Degano P., Nielson F., Nielson H.R. "Static Analysis of Processes for No-Read-Up and No-Write-Down". *Proc. FoSSaCS'99, Vol.1578 of LNC, pages 120-134, Springer-Verlag, 1999.*

[10] Denning D.E. "A Lattice Model of Secure Information Flow". *Communications of ACM 19(5):236-243, 1976.*

[11] Denning D.E., Denning P.J. "Certification of Programs for Secure Information Flow". *Communications of the ACM, July 1977 Vol.20, No: 7, pages 504-513.*

[12] Garlan D. "Software Architecture: a Roadmap". *The Future of Software Engineering, A. Finkelstein ed. ACM Press, 2000.*

[13] Garlan D. "Software Architectures". *In Encyclopedia of Software Engineering, John Wiley & Sons, Inc. 2001.*

[14] Garlan D., Shaw M. "An Introduction to Software Architecture". *Advances in Software Engineering and Knowledge Engineering, Vol.I, World Scientific Publishing Company, 1993.*

[15] Hinchey M.G., Jarvis S.A. “Concurrent Systems: Formal Development in CSP”. *McGraw-Hill International Series in Software Engineering, 1995.*

[16] Hoare C.A.R. “Communicating Sequential Processes”. *Prentice-Hall series in computer sciences, 1985.*

[17] Jaeger T., Tidswell J.E. “Practical Safety in Flexible Access Control Models”. *ACM Transactions on Information and System Security, Vol.4 No.2, May 2001, pages 158-190.*

[18] Jensen J.C. “Secure Software Architectures”. *Proceedings of the Eighth Nardic Workshop on Programming Environment Research, pp 239-246, Ronneby, August 1998.*

[19] Landwehr C.E. “Computer Security”, *Springer-Verlag 2001.*

[20] Landwehr C.E. “Formal Models for Computer Security”. *Computing Surveys, Vol. 13, No.3, 247-278, September 1981.*

[21] Lin T.Y. “Bell and LaPadula Axioms: A ‘New’ Paradigm for an ‘Old’ Model”. *Proc. Of the 1992 1993 ACM SIGSAC on New Security Paradigms Workshop. Little Compton: ACM Press, pp 82-93, 1993.*

[22] Loogen, R. and Goltz, U. “A non-interleaving semantic model for nondeterministic concurrent processes”, *RWTH Aachen Fachgruppe Informatik, Aachener Informatik-Berichte, Nr. 87-15, Aachen 1987.*

[23] Luckham D., Vera J. “Three Concepts of System Architecture”. In *Technical Report CSL-TR-95-674, Stanford University, July 1995.*

[24] Luckham D.C., Kenney J.J., et.al. “Specification and Analysis of System Architecture Using Rapide”. In *IEEE Transactions on Software Engineering, 21(4):336-355, April 1995*.

[25] Martin K. “A Principle of Induction”. *CSL’01, LNCS, Vol.2142, p.458 2001.*

[26] McLean J. “A Comment on the ‘Basic Security Theorem’ of Bell and LaPadula”. *Information Processing Letter, vol.20, No.2: 67-70, February 1985.*

[27] Meldal S., Luckham D. C. “Defining a Security Reference Architecture”. *Technical report CSL-97-728, Program Analysis and verification Group, June 1997.*

[28] Moriconi M., Qian X., Riemenschneider A., Gong L. “Secure Software Architectures”. *Proceedings of the 1997 IEEE Symposium on Security and Privacy, pages 84-93, May 1997.*

[29] Olson I.M., Abrams M.D. “Computer Access Control Policy Choices”. *Computers & Security, Vol.9, No. 8, 1990, pp. 699-714.*

[30] Roscoe A.W. “The Theory and Practice of Concurrency”. *Prentice-Hall, 1998*.

[31] Rosen K.H. “Discrete Mathematics and Its Applications Fourth Edition”. *McGraw-Hill 1999.*

[32] Sabelfeld A., Myers A.C. “Language-based Information Flow Security”. *IEEE Journal on Selected Areas in Communications, 21(1), 2003.*

[33] Samarati P. and Vimercati C.S. “Access Control: Policies, Models and Mechanisms”, *LNCS 2171, Springer-Verlag 2001.*

[34] Sandhu R., Munawer Q. “How to do Discretionary Access Control Using Roles”. *Proceedings of 3rd ACM Workshop on Role-Based Access Control, Fairfax, Virginia, October 22-23,1998.*

[35] Sandhu R..S., Samarati P. “Authentication, Access Control and Audit”. *ACM Computing Survey, Vol.28, No.1, March 1996.*

[36] Sandhu R.S and Samarati P. Authentication, Access Control, and Intrusion Detection, *in IEEE Communications, vol.32 no. 9, pp.40-48.*

[37] Sandhu R.S. “Lattice-Based Access Control Models”. *IEEE Computer Vol. 26, No: 11, Nov. 1993 9-19.*

[38] Sandhu R.S. “The Typed Access Matrix Model”. *Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, May 4-6, 1992, pages 122-136.*

[39] Sandhu R.S., Samarati P. “Access Control: Principles and Practice”. *IEEE Communications 32,9, 40-48, 1994*.

[40] Shaw M, Deline R. “Abstractions and Implementations for Architectural Connections”. *Proceedings of the Third International Conference on Configurable Distributed Systems, 1996.*

[41] Shaw M., Deline R., Klein D.V., et.al. “Abstractions for Software Architecture and Tools to Support Them”. *IEEE Transactions on Software Engineering, Vol.21, No.4, pp:314-335, 1995.*

[42] Spitznagel B., Garlan D. “A Compositional Approach for Constructing Connectors”. *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001).*

[43] Ulu C., Oğuztüzün H. “Specification of Confidentiality Authorizations at Architectural Level”, *In the Proceedings of the 3rd Asia Pacific International Symposium on Information Technology (APIS’03) Jan. 13-14, 2004, İstanbul.*

[44] Yolaçan B., Ulu C., Oğuztüzün H. “An XML Schema for Wright with Confidentiality Extensions (in Turkish)”. *Proceedings of the National Symposium on Software Engineering (UYMS’03). İzmir, Turkey, October 2003.*