Carnegie Mellon University
Software Engineering Institute

Modeling and Validating Security and Confidentiality in System Architectures

Aaron Greenhouse
Jörgen Hansson
Lutz Wrage

March 2021

TECHNICAL REPORT
CMU/SEI-2021-TR-004
DOI: 10.1184/R1/13659911

Software Solutions Division

http://www.sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100

DM21-0145

Table of Contents

List of Figures

Listings

Abstract

The importance of security in computer and information systems is increasing as network-connected computer systems become more ubiquitous. The objective of security is to verify that the computing platform is secured and that data and information are properly accessed and handled by users and applications, ensuring data confidentiality and integrity. To develop a framework for modeling and verifying security as a data quality attribute, designers need to identify parameters and variables with the expressive power to capture and represent security models and determine the type of analysis to enable. This report presents an approach for modeling and validating confidentiality based on the Bell–LaPadula security model using the Architecture Analysis and Design Language (AADL). The report describes the Bell–LaPadula security model and elaborates how security and Bell–LaPadula attributes are mapped to concepts and represented in AADL. It then describes modeling and validating security in AADL models, considering conditions that need to be enforced for a system to ensure conformance to the Bell–LaPadula security policy. It also presents the analysis capabilities provided by AADL and examples modeled in AADL.

1 Introduction

Security in computer and information systems has received a lot of attention over the past few decades, and its importance will only increase as the use of network-connected computer systems becomes more ubiquitous. Ensuring computer security is a multifaceted technical challenge, and it focuses on developing a secure computing platform by means of protective measures that include prevention, detection, and reaction. *Prevention* measures the focus on protecting resources from being used, accessed, or damaged. *Detection* concerns measures for detecting the improper use of a resource. Finally, *reaction* concerns measures taken to enable recovery or remedy a security breach.

Security can be viewed as the concurrent existence of (1) availability for authorized users only, (2) confidentiality, and (3) integrity [1]. Confidentiality addresses concerns that sensitive data be disclosed to or accessed by only authorized users (i.e., enforcing prevention of unauthorized disclosure of information). Data integrity is closely related, as it concerns the prevention of unauthorized modifications of data. The specific objective of security is to verify that the computing platform is secured and that data and information are properly accessed and handled by users and applications, ensuring confidentiality and data integrity.

In this report, we focus on data confidentiality and data integrity, describing how to model and verify them in a system. Common techniques conducive to confidentiality include the following:

- enforcing access control to stored data objects
- communicating data in encrypted forms when transmitted over networks
- partitioning systems to group subsystems and applications based on their authorized access privileges and the security level of the resources they access

The sensitivity of the data, based on security-level classifications, helps in choosing an appropriate means for enforcing confidentiality. Supporting data integrity implies that modifications or alterations of data are controlled (e.g., by using an access matrix specifying permissible operations by users on data objects).

When developing a framework for modeling and verifying security as a data quality attribute, we need to (1) identify parameters and variables with the expressive power to capture and represent previously developed security models and (2) determine the type of analysis that should be enabled. Some significant contributions to security models have been made (for example, see the paper "Security Models" [20] for good introductions to the field), and important frameworks for enforcing confidentiality have been proposed, specifically the Bell–LaPadula [5], Chinese wall [7, 17], role-based access control [13], Biba [6], and information-flow models [19].

In this report, we present an approach for modeling and validating confidentiality based on the Bell–LaPadula security model and discuss integrity in this context. Specifically, we cover the following topics:

- representation of confidentiality requirements of resources (i.e., objects)
- representation and generation of security clearance/privileges of subjects operating on the objects
- the role of the access matrix, specifying allowed access operations of subjects on objects to support integrity

- analysis of an Architecture Analysis & Design Language (AADL) model of a system with respect to the basic principles of confidentiality, need to know, least privilege, and controlled sanitization

Section 2 provides a brief background and a description of the Bell–LaPadula security model in more detail. In Section 3, we elaborate how security and Bell–LaPadula attributes are mapped to concepts and represented in AADL. Section 4 describes modeling and validating security in AADL models, discussing conditions that need to be enforced for a system. Section 5 walks through examples modeled in AADL. We conclude this report with observations on how model-based engineering can support early modeling and validation of security in Section 6.

2 The Bell–LaPadula Security Model

The Bell–LaPadula security model is a mathematical framework and model for designing secure computer system architectures. In particular, it regulates the dynamic behavior of a system as "subjects" with differing privileges to read from and write to "objects" with different access restrictions. A system in conformance with the model never allows a subject to access an object that it is not permitted to and never allows the subject to manipulate an object in non-permitted ways. An early use of the model was to design a security kernel for the Multics operating system. The model was initially described and evolved in a series of technical reports [3, 2, 4]; it is collected, and its application to Multics described, in the more definitive report [5]. In this section, we summarize the portions of the Bell–LaPadula model needed to understand our mapping into AADL. The interested reader may refer to the report *Secure Computer Systems: Unified Exposition and MULTICs Interpretation* [5] for more details.

2.1 Subjects and Objects

The Bell–LaPadula model regulates the manner in which active **subjects** access passive **objects**. A subject, denoted s_i, is drawn from the set of subjects $S = \{s_1, \ldots, s_n\}$. Examples of subjects in a system include processes, threads, and other software components at a software level and processors and memory at a hardware level. Similarly, an object, denoted o_j, is drawn from the set of objects $O = \{o_1, \ldots, o_m\}$. Examples of objects in a system include executable code and data objects at the software level and memory at the hardware level.[1]

The model considers four **access modes** to objects by subjects based on the four combinations of *observing* and *altering* the object:

- **execute** (no *observation*, no *alteration*)
- **read** (*observation*, no *alteration*)
- **append** (no *observation*, *alteration*)
- **write** (*observation*, *alteration*)

The set $A = \{\mathbf{e}, \mathbf{r}, \mathbf{a}, \mathbf{w}\}$, respectively, denotes the access modes. As with subjects and objects, the exact interpretation of each access mode depends on the context of the model application.

The system keeps track of the **current access set** B. A current access $(s, o, a) \in B$ means that subject s currently has access a to object o. Allowed accesses are governed by a conceptual access matrix M. Matrix component $M_{i,j} \subseteq A$ records the modes in which subject s_i is permitted to access object o_j. An operational system should not allow a current access (s_i, o_j, a) such that $a \notin M_{i,j}$.

2.2 Security Labels

A **security label** $(level, categories) \in B$ is a pair constructed from a classification $level \in L$ and set of categories $categories \subseteq P$. The set L has a linear order $\leq$, and thus the elements of L correspond to "classification" or "clearance" levels, such as unclassified or confidential. The

[1] The model places no restrictions on entities that may be both subjects and objects. In this case, memory can both be a subject and an object. Modeling a software application, we want to ensure that a data element (object) is allowed to be stored only in dedicated memory (subject) that has support for hardware encryption. The security level of the memory should dominate the security level of the data element. Modeling the system at a hardware component level, we want to ensure that a processor (subject) is only allowed to access memory (object) for which it has clearance.

categories in P embody the concept of "need to know": it is not enough to possess the correct clearance. This requirement is captured by the **dominates** relation $\gg$ that imposes a partial order over $B = L \times P$:

$$(\text{level}_u, \text{categories}_u) \gg (\text{level}_v, \text{categories}_v) \Leftrightarrow \text{level}_v \leq \text{level}_u \wedge \text{categories}_v \subseteq \text{categories}_u$$

Every object o_j has a security label $f_O(o_j)$. Every subject s_i has a **maximum security label** $f_S(s_i)$ and a **current security label** $f_C(s_i)$. The maximum security label of a subject does not change and denotes the upper bound of the subject's permission. A subject is allowed to operate at a lower security label—indeed, this is necessary in some situations (see Section 2.3)—and this is captured as the current security label. It is required that $f_S(s_i) \gg f_C(s_i)$.

2.3 Security Properties

For each current *observe* access, when the maximum security label of the subject dominates the security label of the object, the **simple security property** is satisfied. More specifically, the property requires $\forall(s, o, a) \in B : \ a \in \{\mathbf{r}, \mathbf{w}\} \Rightarrow f_S(s) \gg f_O(o)$.

The simple security property protects only information containers, not the information itself. In particular, it does not prevent a subject from reading information from a high-level object and writing that information to a low-level object. Enforcement of the ***-property** or **star property** is necessary to prevent this from occurring. A subject that has simultaneous *observe* access to one object and *alter* access to another object satisfies the star property if the security label of the altered object dominates the security label of the observed object:

$$\forall(s_i, o_i, a_i), (s_j, o_j, a_j) \in B : \ a_i \in \{\mathbf{r}, \mathbf{w}\} \wedge a_j \in \{\mathbf{a}, \mathbf{w}\} \Rightarrow f_O(o_j) \gg f_O(o_i)$$

In plain English, the security label of information can only increase. A consequence of satisfying the star property is that the access modes constrain the current security label of a subject. When the star property is satisfied, we have

$$\forall(s, o, a) \in B : \begin{cases} a = \mathbf{a} & \Rightarrow \quad f_O(o) \gg f_C(s) \\ a = \mathbf{w} & \Rightarrow \quad f_O(o) = f_C(s) \\ a = \mathbf{r} & \Rightarrow \quad f_C(s) \gg f_O(o) \end{cases}$$

Specifically, a subject may need *to lower* its current security label before it can write information to an object. Additionally, we emphasize that it is implicit in the model that a subject s does not hold on to or remember information read from an object o when it no longer has access to the object, that is, when $(s, o, a) \notin B$. Being able to do so would allow a subject to circumvent the star property by observing a high-level object, remembering the high-level information, disconnecting from the high-level object, and then writing the remembered information to a low-level object.[2]

Finally, the **discretionary security property** has already been mentioned. All current accesses must respect the access matrix: $\forall(s_i, o_j, a) \in B : \ a \in M_{i,j}$.

[2] Alternatively, this behavior can be prevented by considering all memory addresses to be objects requiring a security label before data can be remembered in or read from them. A physical example would be the requirement that writing notes into a notebook while reading a secret document causes the notebook to also be considered secret.

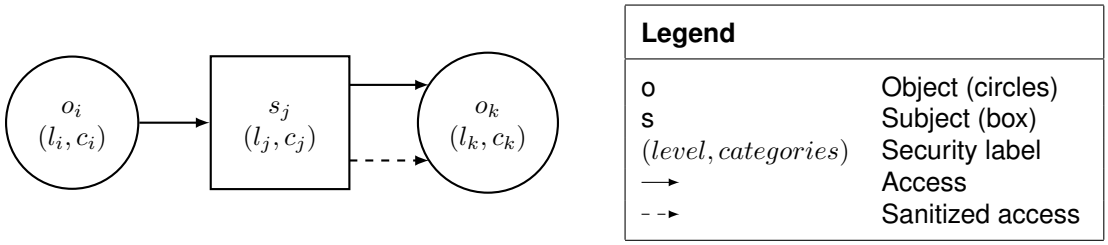

Figure 2.1: Graphical Notation of $o_i \rightarrow s_j \rightarrow o_k$

2.4 Trusted Subject

By design, a consequence of the star property is that a subject cannot derive low-level data from high-level data or have simultaneous *observe* and *alter* access to unrelated objects with incompatible security labels. But some subjects, such as the scheduler in an operating system or an encryption component, can be trusted to not perform a security-breaching information transfer even if it is possible. So-called **trusted subjects** are exempt from the star property. The design and implementation of trusted subjects are expected to receive extra scrutiny. We say that the low-level output of a trusted subject is **sanitized**.

2.5 Examples and Discussion

We now introduce an example to better elaborate the details of the Bell–LaPadula model. Figure 2.1 introduces the graphical notation we use for subjects and objects in this and following sections. The figure shows a subject s_j accessing the objects o_i and o_k. Subject s_j has security label $f_S(s_j) = (l_j, c_j)$, and the objects o_i and o_k have the security labels $f_O(o_i) = (l_i, c_i)$ and $f_O(o_k) = (l_k, c_k)$, respectively. The relation $o_i \rightarrow s_j$ implies that s_j *observes* o_i. The relation $s_j \rightarrow o_k$ shows that s_j *alters* o_k. Thus, s_j has **r** access to o_i and **a** has access to o_k. A feedback loop between a subject and an object, as with o_1 and s_1 in Figure 2.2, signifies **w** access.[3] We say that s_j uses o_i as input and produces o_k as output. We represent sanitized output by a dashed arrow in the figure and denote it as $s_j \rightsquigarrow o_k$; sanitization is allowed to be performed only on altered objects.

[3] This notation cannot express **e** access.

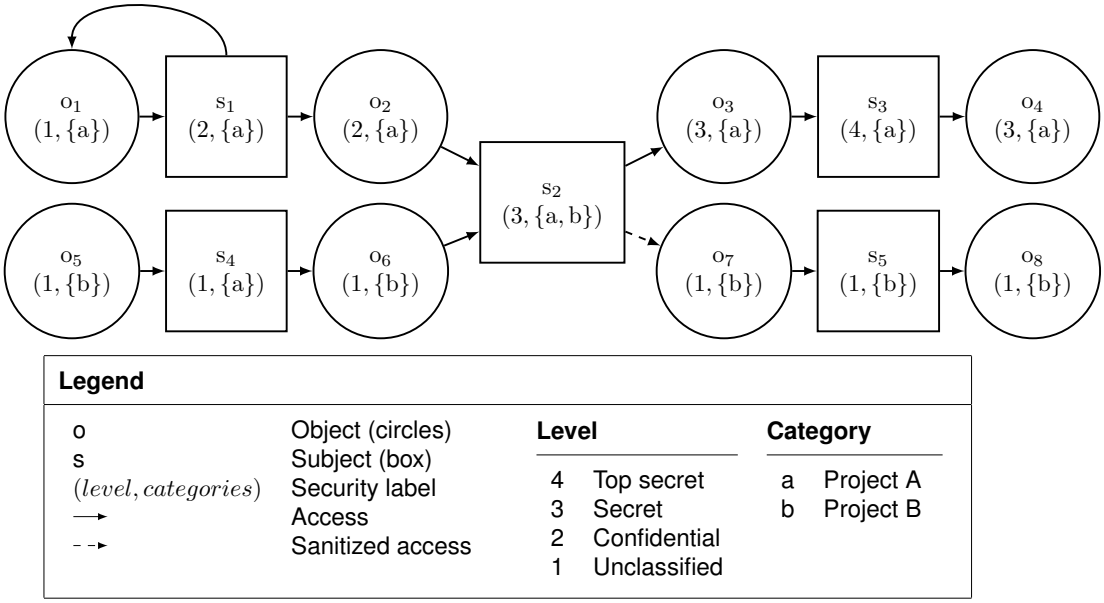

Figure 2.2: An Example with Object–Subject Dependencies

Figure 2.2 depicts an example with $S = \{\mathrm{s}_1, \ldots, \mathrm{s}_5\}$ and $O = \{\mathrm{o}_1, \ldots, \mathrm{o}_8\}$ consisting of a number of scenarios, potential faults, and errors that we refer to below. The example contains several object–subject dependencies:

- a feedback loop $\mathrm{o}_1 \to \mathrm{s}_1 \to \mathrm{o}_1$ indicating that s_1 has **w** access to o_1
- subjects, such as s_1 and s_2, that read from and write to multiple objects
- a trusted subject with sanitization: $\mathrm{s}_2 \rightsquigarrow \mathrm{o}_7$

2.5.1 Access Matrix

The access matrix M for the example is

$$M = \begin{array}{c} \\ \mathrm{s}_1 \\ \mathrm{s}_2 \\ \mathrm{s}_3 \\ \mathrm{s}_4 \\ \mathrm{s}_5 \end{array} \begin{array}{c} \begin{array}{cccccccc} \mathrm{o}_1 & \mathrm{o}_2 & \mathrm{o}_3 & \mathrm{o}_4 & \mathrm{o}_5 & \mathrm{o}_6 & \mathrm{o}_7 & \mathrm{o}_8 \end{array} \\ \begin{pmatrix} \{\mathbf{w}\} & \{\mathbf{a}\} & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \{\mathbf{r}\} & \{\mathbf{a}\} & \emptyset & \emptyset & \{\mathbf{r}\} & \{\mathbf{a}\} & \emptyset \\ \emptyset & \emptyset & \{\mathbf{r}\} & \{\mathbf{a}\} & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \{\mathbf{r}\} & \{\mathbf{a}\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{\mathbf{r}\} & \{\mathbf{a}\} \end{pmatrix} \end{array}$$

As stated previously, in an operational system, the access matrix is consulted during execution to dynamically determine whether a subject should have access to an object. Furthermore, the matrix itself is dynamic: whether and how a subject is allowed to access a particular object can be updated during execution (consider the case of altering file permissions, for example). This implies that the access matrix itself is an object and requires protection; see the discussion of Multics access control lists in the report *Secure Computer Systems: Unified Exposition and MULTICs Interpretation* [5].

When modeling a system, however, the role of the access matrix is more flexible:

- On one hand, the matrix can continue to be interpreted as a constraint on the system. In this view, the accesses described in the model need to be checked against the accesses allowed by the matrix, and they need to be rechecked whenever the model changes.
- On the other hand, the access matrix can be derived from a model and serve as a summary of the extant access patterns. However, the level of detail in the model determines the completeness of the access matrix, so an access matrix should be generated only from a model where subject–object dependencies are specified in full.

2.5.2 Security Violations

We begin by noting that the relationship $o_5 \rightarrow s_4$ violates the simple security property because $f_S(s_4) = (1, \{a\}) \not\gg f_O(o_1) = (1, \{b\})$. That is, even though s_4 *has high enough clearance* to access o_1, it does not have a *need to know* based on the sets of categories.

In addition, the security level of s_3 is larger than necessary: $f_S(s_3) = (4, \{a\}) \gg (3, \{a\})$. While this is consistent with the simple security property, it is problematic from the point of view of *data integrity* [18]. Such considerations were outside the scope of the original problem addressed by Bell–LaPadula, but they are easily incorporated: the security label of a subject should be the least-restrictive security label capable of satisfying the simple security and star properties.

2.5.3 Hierarchical Models

Absent from the original Bell–LaPadula model is any consideration of the hierarchical composition of subjects, that is, subjects constructed of other subjects. Decomposing subjects in this manner supports *incremental modeling* and directly relates to the structure of AADL models. As an example, Figure 2.3 reimagines subjects s_1 and s_4 as subcomponents of a new subject s_0:

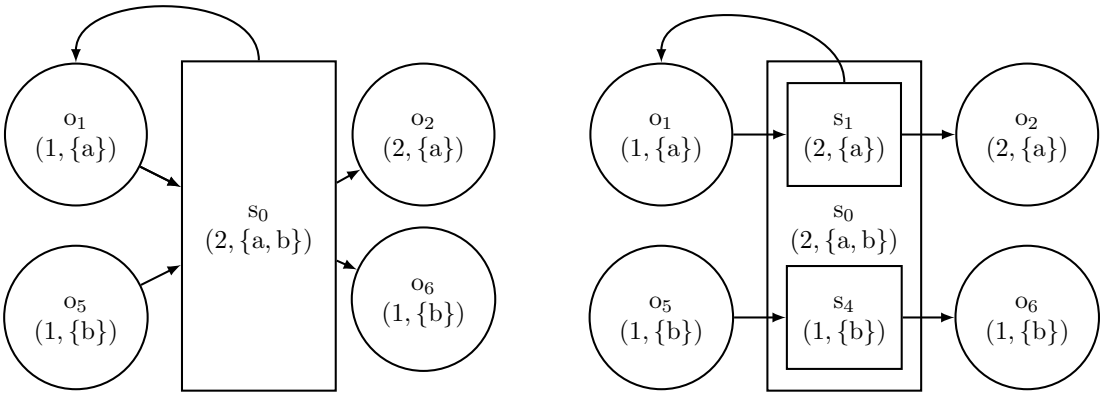

Figure 2.3: Hierarchical Modeling of s_1 *and* s_4

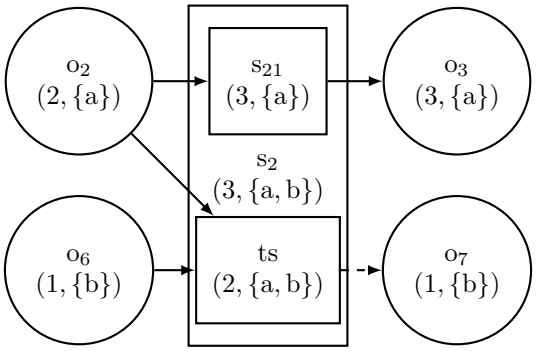

Figure 2.4: Decomposition of s_2 *Showing a Trusted Subject* ts

- The left side shows s_0 substituted for s_1 and s_4.
- The right side shows the decomposition of s_0:
  - Subjects s_1 and s_4 are nested inside s_0 to show the hierarchical structure.
  - The erroneous security level of s_4 mentioned above is corrected here to $(1, \{b\})$.
- The security level of s_0 needs to be permitted to access all the objects accessed by its subcomponents. More specifically, the security level must dominate the security levels of its subcomponents: $f_S(s_0) \gg f_S(s_1) \wedge f_S(s_0) \gg f_S(s_4)$. In this case, we have $f_S(s_0) = (2, \{a, b\})$.

Although not shown here, it is reasonable for s_0 to have accesses of its own that do not become delegated to subcomponents. These accesses would be subject to the simple security property and star property as usual, and the security level of s_0 would need to reflect this. Additionally, a single access by an outer component could decompose to multiple accesses across multiple nested components or even the outer component itself. Finally, it is natural to continue to decompose subjects s_1 and s_4 as necessary.

2.5.4 Trusted Subject

Access $s_2 \rightsquigarrow o_7$ is sanitizing, in this case combining information from o_2 and o_6. Such an action should yield an object with the security label $(2, \{a, b\})$, but in this case, the resulting security label is instead $(1, \{b\})$. In contrast, access $s_2 \rightarrow o_3$ is not sanitizing. This requires that no information from o_6 may be written to o_3 because o_3 misses category b. Subject s_2 is decomposed in Figure 2.4:

- Subject s_2 contains a trusted subject ts that performs the required sanitization: ts $\rightsquigarrow o_7$.
- Subject s_2 contains a subject s_{21} with *observe* access to only o_2 and no access to o_3.
- Access $o_2 \rightarrow s_2$ decomposes to the accesses $o_2 \rightarrow s_{21}$ and $o_2 \rightarrow$ ts.

Introducing a trusted subject in this case serves to reduce the complexity of the trusted subject. It accesses only the minimum number of objects necessary to fulfill its function, thus simplifying verification.

3 Representing Bell–LaPadula in AADL

In this section, we describe our approach to mapping the concepts from the Bell–LaPadula security model to architectural models written in AADL.[1] This mapping is described in three steps:

1. We describe our use of the AADL property mechanism to define security labels.
2. We describe our mapping of subjects and objects to AADL language features.
3. We describe how this mapping affects Bell–LaPadula access modes.

3.1 Security Labels in AADL

As described in Section 2.2, the security label of a subject/object consists of a security level value and a set of categories. The level value is drawn from a totally ordered set and denotes how securely an object must be handled and how privileged a subject is. Categories further refine the security level by labeling objects and giving subjects permission to access appropriately labeled objects. This style of security label captures the typical governmental model, wherein security levels are **unclassified**, **confidential**, **secret**, or **top secret** and categories are used to further restrict access. Represented this way, a security label $(\text{level}_1, \text{categories}_1)$ is said to *dominate* another security label $(\text{level}_2, \text{categories}_2)$, if and only if $\text{level}_1 \geq \text{level}_2$ and $\text{categories}_1 \supseteq \text{categories}_2$.

We use AADL property associations to add security labels to AADL components and features that represent subjects and objects. We use a pair of properties to associate both a level and a list of categories with each subject/object.[2] The declarations of the properties `Level` and `Level_Caveats` that declare an item's security level and set of categories, respectively, are shown in Listing 3.1.

```
1  -- Security Levels
2
3  Level: inherit Security_Type_Specifications::Level_Type =>
4      Security_Type_Specifications::Minimum_Level
5    applies to (system, processor, virtual processor, thread,
6      thread group, subprogram, subprogram group, data, port,
7      feature group, process, device, memory, abstract, flow,
8      parameter, access);
9
10 -- Security Categories
11
12 Level_Caveats: inherit list of
13     Security_Type_Specifications::Caveat_Type => ()
14   applies to (system, processor, virtual processor, thread,
15     thread group, subprogram, subprogram group, data, port,
16     feature group, process, device, memory, abstract, flow,
17     parameter, access);
```

Listing 3.1: The `Level` and `Level_Caveats` Property Declarations

[1] This work is based on earlier work published in Hansson et al. [14].

[2] For this work, we use the `Security` property set from the proposed AADL security annex. The scope of the property set is more general than the work presented here, so the properties are more widely applicable than our needs require. For example, the `Level` property may be applied to `subprogram` and `subprogram group` classifiers even though, as we shall see, they are neither subjects nor objects in our mapping.

3.1.1 The `Security` and `Security_Type_Specifications` Property Sets

Both the `Level` and `Level_Caveats` properties are marked as `inherit`. This means that if a component or feature is not explicitly associated with a value for the property, it will receive the property value of its containing component. The definitions reference two property types, `Level_Type` and `Caveat_Type`, and a property constant, `Minimum_Level`, that are defined in a secondary property set `Security_Type_Specifications`. This facilitates customization of the space of security levels and is analogous to the AADL standard's use of the `AADL_Project` property set to allow customization of the property definitions in the otherwise fixed `AADL_Project` property set.

More specifically, the `Level_Type` property type of the `Level` property is expected to be an enumeration. We consider the enumeration literals as being ordered based on the order they are declared in the enumeration type; the `Level_Type` type thus provides a totally ordered set of classifications for the space of security levels. We interpret them as going from the highest to the lowest level. The property constant `Minimum_Level` provides the default value for the `Level` property. The value of this constant is expected to be the last (that is, lowest) literal of the `Level_Type` enumeration type.[3]

The `Caveat_Type` property type is also expected to be an enumeration. In this case, the enumeration is used to define the set of categories applicable to the problem space of the model. Acceptable values for the `Level_Caveats` property are actually *lists* of values of type `Caveat_Type`, thus providing the second component of the security label: the set of categories.[4]

```
1 -- Security Classification Properties type declarations
2
3 Level_Type:  type  enumeration (
4   TopSecret, Secret, Confidential, Unclassified);
5
6 Minimum_Level: constant Security_Type_Specifications::Level_Type
7   => Unclassified;
8
9 Caveat_Cype: type  enumeration (A, B, C, D, E);
```

Listing 3.2: The `Level_Type` and `Caveat_Type` Property Type Declarations

The property types `Level_Type` and `Caveat_Type` and the property constant `Minimum_Level` are declared in the `Security_Type_Specifications` property set, shown in Listing 3.2. The default security levels are the standard military classification levels with the ordering `TopSecret` > `Secret` > `Confidential` > `Unclassified`. For the categories, we use A, B, C, D, and E as placeholders for more meaningful project-specific values. As stated previously, the intent is that the modeler customizes the enumerations based on the domain of the system being modeled.

As discussed in the following sections, our approach imposes the modeling constraint that all components and features must have property associations for the `Security::Level` and `Security::Level_Caveats` properties. By defining default values for these two properties, we satisfy this constraint. Furthermore, by declaring the property values as `inherit`, we make it much more likely that the simple security property (see Section 4.2) will be satisfied because—unless otherwise specified—a subcomponent or feature will have the same security

[3] This constant is necessary because there is no type-independent way to identify the last element of an enumeration type, and we would like the default value of the `Level` property to be the least restrictive classification level. It is trivial for an analysis to verify that this constant does indeed refer to the last enumeration literal.

[4] AADL does not support the specification of sets, only of lists (i.e., there is no language mechanism that prevents the repetition of an item in the list). It is easy enough for an analysis to be insensitive to repetition of enumeration literals in the property value.

attributes as its container. Using inheritance and default property values also lightens the annotation burden for the modeler. Specifically, a model with no explicit security property associations will always be correct because all subjects and objects will have the same security label: the lowest security level and the empty set of categories.

3.2 Subjects and Objects in AADL

In the Bell–LaPadula model, active subjects act on passive objects. In AADL, components communicate through ports and other categories of features. For the most part, data is not explicitly represented in the model, although `data` subcomponents can be shared with other components via `data access` features. Instead, `data port` and other features belonging to components are connected to describe the transfer of information throughout the modeled system. A feature is thus *a proxy for the data that pass through it.* These observations motivate us to consider, in general, *AADL components as subjects* and *AADL features as objects.* The specifics of, and exceptions to, these considerations are discussed in the sections below, based on AADL language constructs.

3.2.1 AADL Components

We treat all AADL components—with the exception of `data`, `subprogram`, and `subprogram group` components—as *subjects.* Components are sites of activity that coordinate the movement and generation of data throughout the system. Each component (subject) is expected to have a security label, expressed using property associations, to describe its clearance to utilize objects. This is the *maximum* security level that the component requires to operate (see Section 4.1). Listing 3.3 shows an example of a component type with a security level.

```
1 thread producer
2   -- Features, etc., are elided
3   properties
4     Security::Level => confidential;
5     Security::Level_Caveats => (A);
6 end producer;
```

Listing 3.3: A Thread Component with Security Level (confidential, {A})

3.2.1.1 Data Components

AADL `data` components are *objects.* Although they can contain `subprogram` features, `data` components do not possess an active nature; external threads of control must invoke a `data` component's subprograms. A `data` component is operated on through `data access` features that enable direct access to its contents. Thus, a `data` component is expected to have a security label expressed using property associations to describe the classification of its contents.

3.2.1.2 Subprogram and Subprogram Group Components

AADL `subprogram` components are neither subjects nor objects. A subprogram is not data that is manipulated by the system, so it is not an object.[5] Furthermore, a subprogram itself does not manipulate data; instead, a particular invocation, or *call* in AADL parlance, does. And that call actually happens within a particular thread. Thus *the calling `thread` component is the subject*, not the subprogram (see Section 4.2.1).

[5] A system that manipulates code sequences as data is certainly possible, but in such a case the subprograms being manipulated would be modeled as `data` components, not as an AADL `subprogram`.

An AADL `subprogram group` is a passive collection of `subprogram` subcomponents meant to represent a library. Obviously, if a `subprogram` is neither a subject nor an object, then the same is true of a `subprogram group`.

3.2.2 AADL Features

AADL defines a feature as "a part of a component type definition that specifies how that component interfaces with other components in the system" [15, §8 ¶1]. In most cases, a fully specified feature has a classifier associated with it; usually it is a `data` classifier. In general, AADL uses the `data` classifier to describe the data that passes through the port. It is thus natural to use the security-level property associations *of the classifier associated with the feature* as the security level of the data represented by the feature.[6] While this is our preference, it is not always possible:

- A fully specified `event port` feature does not specify a classifier.
- A model is not required to be fully specified; therefore, a feature may be missing a classifier.

The security-label property associations in these cases must be made on the feature declaration. We would prefer, however, to avoid the situation where the security label is specified on *both* the feature and the classifier; we revisit this problem in Section 4.4.1.

The security label of a feature represents the *exact* security label of the data that passes through the feature. In particular, unlike with components, it *does not* represent the maximum security level of the data. Such a choice would cause less precise modeling and analysis.

3.2.2.1 Data Port Features

The `data port` feature is the most obvious example of a feature that transmits data objects. A fully specified `data port` feature has a `data` classifier. Listing 3.4 shows the declaration of the `data port` feature `output` in the thread type `producer`. The security label of the port is (confidential, {A}) because that is the security label of the data classifier `A`.

```
1  data A
2      properties
3          Security::Level => confidential;
4          Security::Level_Caveats => (A);
5  end A;
6
7  thread producer
8      features
9          output: data port A;
10         interrupt: event port {
11             Security::Level => confidential;
12             Security::Level_Caveats => (B);
13         };
14     properties
15         Security::Level => confidential;
16         Security::Level_Caveats => (A, B);
```

[6] The rationale for introducing types that differ only by their security properties came through realizing that modeling intends to make particular semantic properties of the system more apparent and easier to reason about. It is thus acceptable for the model to contain abstractions whose distinctions may be blurred in the actual implementation. In this context, differentiating data type classifiers based on their security properties makes sense; it reduces the modeling overhead and makes clear in the model that different features treat data differently. In fact, it helps prevent connections from being made between features with different security levels because AADL semantics require that the data classifiers on either end of a connection be the same. The data type variants, however, can be mapped to the same implementation type using the standard AADL property `Programming_Properties::Type_Source_Name`.

```
17 end producer;
```

Listing 3.4: Declaring the Security Levels of Features

An `event data port` feature differs from a `data port` feature only by its delivery semantics. It is also straightforward to consider them to be Bell–LaPadula objects.

Likewise, a `parameter` feature of a `subprogram` classifier represents the transfer of data, and it is also a Bell–LaPadula object.

Again, a fully specified `event data port` or `parameter` feature has a `data` classifier, but if the classifier is absent, the property association must be made on the feature itself.

3.2.2.2 Event Port Features

An `event port` does not pass an explicit data object between components. The raising of an event, however, can be interpreted as the transfer of an "event happened" data object, one that need not be explicitly represented because there is only one value. One can easily imagine the need to constrain the observation of particular events to those components with an appropriate clearance. For example, an event that communicates that an intruder was detected should not be publicly available because we might not want the intruder to be able to learn of the detection by querying a public access point. As we see, an `event port` is also an object in the Bell–LaPadula model.

Because it *never* has an associated classifier, an `event port` feature's security-label property values are always retrieved from the feature itself. Listing 3.4 also shows the declaration of an `event port` feature `interrupt` in the thread type `producer`. Its security label (confidential, {B}) must be explicitly declared using property associations on the feature.

3.2.2.3 Feature Group Features

An AADL `feature group` aggregates features. From an architectural point of view, it is a container for features. A fully specified `feature group` feature includes a reference to a `feature group` type classifier. The `feature group` type declares the features of the `feature group`. As with port features, the security label of the `feature group` feature is obtained from the `feature group` type classifier if it is present and from the feature itself if it is not present. As a basic principle of containment, we require that the security label of the feature group dominate the security labels of the features in the feature group. This requirement can be viewed as an application of the **simple security property** (see Section 4.2). The feature declarations in the `feature group` type determine the security labels of the features in the `feature group`.

3.2.2.4 Data Access Features

A `data access` feature differs from a `data port` in that it represents direct access to a data object—one that is ultimately represented by a `data component` instance elsewhere in model. We must still, however, treat the feature as a proxy for the data because the exact `data` component being accessed is unknown outside the component that actually connects the `data` component to the feature.

As with `port` features, a completely specified `data access` feature includes a `data` classifier that describes the data object being shared. We retrieve the security label from the classifier when it is present and from the feature itself if the classifier is unavailable.

3.2.2.5 Subprogram and Subprogram Group Access Features

AADL `subprogram access` and `subprogram group access` features simply make `subprogram` and `subprogram group` components accessible to other components. As pre-

viously discussed, these components are neither subjects nor objects, and thus these features *do not* represent objects.

3.2.2.6 Abstract Features

An abstract feature is a generic or placeholder feature that can later be `refined to` a more specific kind of feature, such as a `data port`, an `event port`, or `data access`. That is, the specification of the semantics of the feature is deferred to a subtype of the containing classifier. However, abstract features may have a classifier that describes the data object being shared. We retrieve the security label from the classifier when it is present and from the feature itself if the classifier is unavailable.

3.2.2.7 Bus Access and Virtual Bus Access Features

In this section, it is enough to remark that `bus access` and `virtual bus access` features *do not* model data in an AADL model and thus *are not objects* for the Bell–LaPadula model. We revisit these kinds of features in Section 4.2.2, where we discuss the impact of software–hardware bindings, and in Section 4.4.2.

3.3 Access Modes in AADL

The Bell–LaPadula model defines four access modes to describe a subject's effects on an object:

1. **Execute** access does not permit the subject to observe or alter the contents of the object.
2. **Read** access permits a subject to observe but not alter the contents of the object.
3. **Append** access permits a subject to alter but not observe the contents of the object.
4. **Write** access permits a subject both to alter and observe the contents of the object.

Given our mapping of objects to AADL features and data components, we must derive access rights based on the AADL semantics for those features. Considering that AADL is not executable, this exercise may seem to be purely theoretical. However, we need to understand the access modes so we can properly enforce the **simple security property** and **star property** of the Bell–LaPadula model.

3.3.1 AADL Port Features

We begin by considering the `data port`. A `data port` transmits and receives data objects by marshalling and un-marshalling, respectively, complete objects between threads through buffers that represent the port in the component. That is, objects obtained from an `in data port` are read, and objects sent through an `out data port` are *newly created*. Thus, an `in data port` corresponds to **read** access, and an `out data port` corresponds to **append** access. An `in out data port` is bidirectional, but not on the same data object; it is more like a port that can be used for both sending and receiving objects, but not simultaneously. When used to receive, an `in out data port` thus corresponds to **read** access; when used to transmit, it corresponds to **append** access.[7]

An `event port` communicates events between threads. Following the notion above that the port transfers an "event happened" object, an `in event port` corresponds to a **read** access, and an `out event port` corresponds to an **append** access. Again, an `in out`

[7] In practice, an `in out data port` is checked as if it were an `out data port` because the `out` nature of an `in out data port` is more constraining than its `in` nature.

`event port` is never simultaneously observing and announcing, so its access mode is determined according to its current usage.

An `event data port` combines the semantics of an `event port` with those of a `data port`. For our purposes, an `event data port` is like a `data port`; in particular, an `event data port` transmits complete data objects between threads. Unlike an `in data port`, an `in event data port` queues received objects. You might suspect that an `in event data port` would have **write** access because it must read from and modify the queue. However, the queue is not interesting from the point of view of a security analysis because we are interested in the data being transmitted *through* the queue and not the underlying implementation of the AADL semantics.

The semantics of subprogram `parameter` features are similar to those of `data port` features. Data is copied into the subprogram at the time it is called and out of the subprogram when it returns. So as with a `data port`, an `in parameter` corresponds to **read** access, an `out parameter` corresponds to **append** access, and an `in out parameter` is **read** access during the call phase and **append** access during the return phase.[8]

3.3.2 AADL Feature Group Features

Fundamentally, a feature group is simply a bundle of features, including those features of any nested feature groups. Conceptually, components interact via the features contained in a feature group, not via the feature group itself; the feature group is simply an abstraction that bundles together related features. Thus, a feature group itself has no direction, but the features it contains do have direction. Therefore, in general, we cannot speak of the access mode of the feature group feature, only of the access modes of the individual features contained in the feature group. These access modes are determined as described above.

There are, however, two aspects of AADL that complicate the above reasoning:

1. A feature group might be empty. This aspect supports incremental modeling by allowing empty feature groups to be declared to abstractly represent the communication that occurs between components, even when the exact nature of the communication has not yet been decided.
2. Two feature groups may be directly connected via feature group connections. In fact, AADL allows access to the contained features of a feature group only when connecting a subcomponent and its containing component; sibling subcomponents must connect feature groups in their entirety using feature group connections.

There are thus cases where we must consider the feature group as a single entity from the point of view of data access. In these cases, clearly the security level of the accessed object is the security level of the feature group. Just as the security level of a feature group must be a maximization of the security levels of its constituent features, so must the access attributed to the feature group be a maximization of the accesses attributed to its features. Conservatively, this implies that we treat a feature group as an **append** access—the only choice when dealing with an empty feature group (because we do not know the directions of the features yet to be added).[9]

[8] It might seem that an `in out parameter` should capture the fact that the data might be modified internal to the subprogram as the subprogram executes. But the semantics of AADL dictate that the internal actions of the subprogram are not relevant here: "An `in out` parameter declaration represents a parameter whose value is passed in and returned *by value*. Parameters passed *by reference* are modeled using `requires data access`" [emphasis added] [15, §8.5 ¶3].

[9] We can be less conservative by inspecting the directions of the feature group's features: If all the features in a feature group (including any nested feature groups) are **in** features, then the feature group can be treated as a **read** access.

3.3.3 AADL Access Features

AADL data access features represent direct access to shared data. Although AADL distinguishes between `provides` and `requires` data access features, this distinction captures where the shared object is located and does not describe whether the object is written to or read from. This access is determined instead by the standard property `Access_Right` on the feature.[10] Acceptable property values are `read_only`, `write_only`, `read_write`, and `by_method`. The first three values indicate direct manipulation of the shared data and map naturally to the access modes **read**, **append**, and **write**, respectively. The `read_write` access right, however, much like the `in out port`, is not used for both actions simultaneously. Rather, we determine the **read** or **append** mode based on context as determined by AADL flow specifications (see Section 4.3.1).

The `by_method` access right indicates that the shared data object may be manipulated only by using the subprogram features declared within its data classifier. Because we do not know what the subprograms do to the data, we map `by_method` to the **write** access mode.

As stated in Section 3.2.2.7, `bus access` and `virtual bus access` features are not objects, so they do not have access modes.

[10] The direction of the connection can also be used to determine the direction of data flow. The property value has a default value and thus is always defined for access features, so it is the most reliable approach to determining access. Also, it is the only way to specify `by_method` access rights.

4 Analysis and Validation in AADL

The Bell–LaPadula model requires the security labels of subjects and objects in a system to be consistent with the simple security property and the star property. In addition, the semantics of AADL require that several additional properties be enforced for the security modeling to make sense. This section describes the analysis rules that must be enforced over an AADL instance model to ensure the consistency of a security model. We begin by revisiting what exactly the security label of a subject means in our implementation of a Bell–LaPadula model.

4.1 Maximum vs. Current Security Label

The Bell–LaPadula model associates two security labels with a subject:

1. the maximum security label
2. the current security label

The maximum security label, as the name implies, is the maximum security label (with respect to the **dominates** ordering) at which the subject may operate. In particular, this label represents the full capability of the subject to access objects in general. Note, however, that it does not say anything about how a subject may manipulate a specific object; an access matrix fills this role. In AADL, the security label associated with a component via the `Level` and `Level_Caveats` properties specifies *the component's maximum security label*.

The current security label represents the security label at which the subject is currently operating (i.e., the subject does not need to operate with its full capabilities). The intent is that a subject modifies its current security label as it acquires and releases access to objects during the course of its execution. Thus, the current security label captures a dynamic behavior of the subject (e.g., as a user modifies its current state to access files in a shared file system). Unfortunately, AADL is not well suited to capturing dynamic system behavior at this level of detail.[1] We choose, therefore, not to represent the current security level of a subject explicitly but to obtain this information implicitly from additional features of the AADL model. We return to this issue in Section 4.3.

4.2 Checking the Simple Security Property

The simple security property enforces the expectation that a subject should access only objects that it is allowed to use. Specifically, it requires that the maximum security label of a subject dominates the security label of each object that it accesses. In AADL, a component declares all its data accesses as explicit features. Thus, to check the simple security property, we need to enforce the following rule:

Rule 1 (Simple Security Property for Components) *The security label of each component must dominate the security label of each feature in the component.*

Feature groups do not require special treatment with respect to this rule because of the requirement from Section 3.2.2.3 that the security label of a feature group must dominate the security labels of its features. If the security label of a component dominates the security label of a feature group, it must also dominate the security labels of the feature group's features;

[1] AADL modes allow a modeler to describe alternate aspects of a component's operation. Mode transitions capture the dynamic aspects of system behavior. Modes are better suited to describe alternate operating states of a component in which it has differing interactions with other components in the system (e.g., encrypting versus clear-text or normal operation versus recovering from failure) than to describe the dynamics of a component during its normal course of operation.

otherwise, there will be errors reported on the feature group's features. We now recast this previous architectural rule as an additional analysis rule:

Rule 2 (Simple Security Property for Feature Groups) *The security label of each feature group must dominate the security labels of each feature in the feature group.*

We also require a rule for enforcing the simple security property on subcomponents:

Rule 3 (Simple Security Property for Subcomponents) *The security label of each component must dominate the security label of each subcomponent in the component.*

This rule has no direct analog in the Bell–LaPadula security model because the model does not consider a hierarchical relationship among subjects. The rationale for this check in the case of non-`data` components is that a subcomponent performs work for—as part of—its containing component. Its container, therefore, must be authorized to access the data that the subcomponent may access. A `data` subcomponent exists in one of two scenarios:

1. **As a subcomponent of a non-`data` subcomponent.** In this case, the subcomponent is data, an *object* accessed by the containing *subject*, and the above rule is a direct application of the simple security property.
2. **As a subcomponent of a `data` subcomponent.** The subcomponent represents a portion of the overall data component. Therefore, the containing `data` component must have a security level at least as great as that of the subcomponent.

4.2.1 Checking Subprogram Calls

As mentioned in Section 3.2.1.2, `subprogram` subcomponents are not subjects and are instantiated as part of an AADL call sequence. As the name implies, a call sequence describes a sequence of `subprogram` invocations. Call sequences may appear only in `subprogram` and `thread` components; the AADL standard specifies *which component* executes any given subprogram call within a sequence [15, §5.2 ¶19]. In short, depending on how the `subprogram` is (1) declared in the model and (2) referenced in the call sequence, the `subprogram` may be executed by one of the following components:

- the calling `thread`
- the called `thread` (i.e., a remote procedure call)
- the providing `device`
- the providing `processor`

This yields the following rule:

Rule 4 (Simple Security Property for Subprogram Calls) *The security label of a feature in a subprogram call must be dominated by the security label of the component that executes the subprogram.*

This rule is presently of no practical importance because the current AADL tool sets do not actually instantiate call sequences.

4.2.2 Software–Hardware Bindings

AADL components can be divided into software components, such as `thread`, `data`, and `process`, and hardware components, such as `bus`, `memory`, and `processor`. In a *completely instantiated and bound model* [15, §13.1 ¶3], the software components must be *bound* to hardware components. The binding is specified via property associations on the software components. The full details are described in the work of Feiler and Gluch [11, §11.2]. The basic intent is that threads are bound to processors, data is bound to memory, and connections are bound to buses, although more complicated bindings are possible. Of particular note

is that `data port` and `event data port` features should be bound to a memory component that indicates where the actual data that flows through that port is stored.

The software–hardware binding is very much analogous to the subcomponent hierarchy discussed above:

- Because a `processor` executes a `thread`, it must be authorized to access the data (that is, objects) that the `thread` needs to access.
- Memory must be authorized to store the actual data that is bound to it.

This leads to the following rule:

Rule 5 (Simple Security Property for Binding) *If a component c or feature f is bound to a component t, the security label of t must dominate the security label of c or f.*

If multiple components or features are bound to the same hardware component, its security label must dominate all their security labels.

Connections are interesting in this context because—excepting `bus access` and `virtual bus access` connections—they act like objects.[2] The data passing through the connection must be accessible to all the components the connection is bound to. As described in Section 4.4.2, the security label of the data comes from the connection end points.

This yelds the following rule:

Rule 6 (Simple Security Property for Bound Connection) *If a connection c is bound to a component t, the security label of t must dominate the security label of c's data.*

4.3 Checking the Star Property

The star property enforces the expectation that a subject should not be able to leak classified information by reading data from a high-level object and then writing that information into a low-level object. Fundamental to the star property is that if a subject has simultaneous "observe" access to one object and "alter" access to another object, the security label of the alterable object must dominate the security label of the observable object. This principle can be restated in terms of the relationship between the current security label of a subject s and the security label of a single object o, as follows:

- If s has **append** access to o, the security label of o must dominate the current security label of s.
- If s has **write** access to o, the security label of o must be equal to the current security label of s.
- If s has **read** access to o, the security label of o must be dominated by the current security label of s.

4.3.1 Representing the Current Security Label

To check the star property, we must resolve the issue of representing the current security label. The simplest approach is to leave the concept out of our mapping into AADL (i.e., the maximum security label is always the current security label). This choice would result in the requirement that all `out` ports have the same security label as their containing components:

- Per Section 3.3.1, `out` ports have **append** access, and the simple security property requires that *the component's label dominate the port's label.*

[2] While not addressed in the standard [15], it makes no sense for a `bus access` or a `virtual bus access` connection to be bound because it creates the hardware structure to which other connections are bound.

- The star property, above, requires that because the port has **append** access, *the port's label dominates the component's label.*

This requirement would be overly restrictive when a component has multiple output ports because all the ports would have to have identical security labels. Such a requirement may not be satisfiable without artificially inflating the security labels of ports throughout the system.

An alternative approach to modeling the current security label presents itself when we recall the intent behind the star property: Data should not flow from a high-level object to a low-level object. While AADL is not well suited to describing dynamic component behavior, it is designed to describe data flow through a component. AADL `flow path` specifications explicitly declare that data arriving at the source feature flows to (or otherwise influences) the data written to the destination feature. These factors suggest that we can check the star property by using flows as a surrogate for explicit current security label information. In particular, the natural way to check the star property via flows would be to check that the security label of the feature that is the flow source is dominated by the security label of the feature that is the flow sink.

Under our mapping of the Bell–LaPadula model into AADL, this is exactly what *must* be the case:

- AADL semantics require the source of a `flow path` to be an `in` feature or an `access` with `Read_Only` or `Read_Write` access rights. As previously discussed, these map to **read** mode ("observe") access.
- Similarly, AADL semantics require the destination of a `flow path` to be an `out` feature or an `access` with `Write_Only` or `Read_Write` access rights. As previously discussed, these map to **append** mode ("alter") access.
- Therefore, the star property dictates that the security label of the flow source must be dominated by the security label of the flow destination.

This improvement allows a component to have `out` ports with differing security labels because it no longer relates the security label of a feature to the security label of its containing component.

Although the Bell–LaPadula model is traditionally considered to be an *access control model* [16] of security, this approach to checking the star property actually makes our security analysis implementation an *information flow model* [12, 9, 10] that operates on the semantics of AADL rather than of a lower level programming language.

4.3.2 Checking the Star Property in AADL

The following rule summarizes the above discussion:

Rule 7 (Star Property for Flows) *For each flow path in a component, the security level of the destination feature/component must dominate the security level of the source feature/component.*

There is no need to check `flow sink` and `flow source` specifications; they do not denote a path from one feature to another.

In light of the discussion in Section 3.3.2, it is worth noting that a `feature group` feature that appears as an end point of a `flow path` does not require special treatment. As features, they must have security labels.

4.4 Checking Architectural Consistency

We must perform several additional checks on a model to make sure that its security attributes are consistent with AADL-specific structures.

4.4.1 Features

In our modeling approach, we prefer that the security label of a feature is specified by the property associations of the feature's data classifier. This is not always possible, so sometimes the property associations on the feature itself must be used (see Section 3.2.2). Because AADL semantics dictate that the property associations on the feature take precedence [15, §11.3 ¶12], our approach becomes confusing when security-label property associations are present on both the feature and the feature's classifier. Such a situation conflicts with our desire that the feature's classifier be the sole descriptor of the data. The following rule clarifies this situation:

Rule 8 (Feature–Classifier Label Equality) *If both a* feature *and the* feature's classifier *have security-label property associations, the two security labels must be equal.*

4.4.2 Connections

We must check that the security label of the source of a connection is equal to the security label of the destination of a connection. This check is necessary because the AADL feature (or data component) that is the connection destination represents the *same* data object as the AADL feature (or data subcomponent) that is the connection source. If the end points were allowed to have different security labels, we would be saying that a single data object could simultaneously have two different security labels.

As mentioned above, ideally, ports would have a `data` classifier specified, and these would be necessarily identical at the source and destination of a connection. Were this universally true, this check would be unnecessary. We have, however, already discussed several exceptions to this expectation and now further point out that AADL allows the source and destination to have *different* classifiers subject to reasonable type-matching rules [15, §9.2 ¶13].

Access connections involving `bus` and `virtual bus` components and access features create hardware paths that other connections can be bound to. As mentioned previously, `bus access` and `virtual bus access` features are not considered objects but rather proxies for *subjects*, so the above discussion of connections would not seem to apply to them. However, the situation is analogous, and we require that the security label of the bus shared by an access feature is adequately described by the security label of the access feature. This description is ideally obtained from the feature's `bus` or `virtual bus` classifier but is otherwise obtained from the feature itself.

In an AADL instance model, "a connection instance is created from the ultimate source to the ultimate destination component by following a sequence of connection declarations" [15, §9 ¶2]. That is, the destination of an individual connection declaration becomes the source of the next individual connection declaration, so all the features must have identical security labels.

This discussion is summarized as the following rule:

Rule 9 (Connection Label Equality) *The security label of the source feature of a connection must be identical to the security label of the destination feature of a connection. The security labels of the source and destination features of all the individual connection declarations of a connection instance must be identical.*

4.4.3 Flows

Because we rely on `flow path` specifications to check the star property, it is essential that the flow specifications for a component be complete. If there are undocumented data flows, high-level data could be written into low-level objects without being detected by analysis. In particular, we must check that the flow specifications in each component type account for all possible data flows in implementations of that type. For each component implementation, we must determine its actual flow sinks, flow sources, and flow paths and compare them against those declared in the implementation's type.

We discover an implementation's actual flows by using its connection declarations and the flow specifications of its subcomponents. This analysis approach is *modular*. We can rely on the flow specifications of a subcomponent because the analysis of the subcomponent's classifier determines whether that subcomponent has missing flow specifications. We search for flow paths and flow sinks starting from each `in` port feature. Essentially, we follow each alternating sequence of connections and subcomponent flows until the sequence connects to an `out` port of the implementation, in which case we have found a flow path, or we encounter a subcomponent flow sink, in which case we have found a flow sink in the implementation. If the component's type does not declare a flow path between the two ports or a flow sink at the specific `in` port, an error should be reported. We search backward from `out` ports in a similar manner to discover flow sources.

4.4.4 Least Privilege

A component should not have a higher security label than it needs to access the data that it manipulates. There is no advantage to a properly functioning component to have a higher security label than necessary. By minimizing the security label of a component, we minimize the risk that the component will manipulate data irrelevant to its task. We thus recommend that each component be checked to ensure that it has the least privilege it needs to do its job. This guideline relates to the principle of "need to know."

More formally, the security label of a component should be the least upper bound of the labels of its constituents. The least upper bound of two security labels $(\text{level}_1, \text{categories}_1)$ and $(\text{level}_2, \text{categories}_2)$ can be defined as $(\max(\text{level}_1, \text{level}_2), \text{categories}_1 \cup \text{categories}_2)$.[3]

Specifically, we recommend the following rules be enforced:

Rule 10 (Least Privilege) *The security label of a feature group type should be the least upper bound of the security labels of its features. The security label of a component type should be the least upper bound of the security labels of its features. The security label of a component implementation should the least upper bound of the security labels of its features, subcomponents, and subprogram calls.*

4.5 Sanitizing Information

The purpose of enforcing a security model on a system is to prevent information that should be kept secret from being released to those who cannot be trusted to see it. In the Bell–LaPadula model, the star property is central to achieving this goal: It prevents secret information from being written into nonsecret containers. In other words, *data can only become more secure*. Of course, in the real world, this approach is too limiting. Systems need to be allowed to perform the following practical behaviors:

- derive less secret data from secret data and pass that derived data to other less-than-fully-trusted systems
- pass secrets over public communications channels

[3] This definition exploits the fact that our level ordering is linear.

The former might be possible by removing or obfuscating personally identifiable information from a database record; the latter could be achieved by encrypting the information before it is sent over the wire. In the literature, these kinds of actions are often referred to as *sanitizing* (the term we use), *write downs*, or *downgrades*. From the modeling point of view, however, these legitimate actions are valid exceptions to, but violations of, the star property. In this section, we describe an approach to incorporating these sorts of exceptions into the model.

4.5.1 Sanitized Flows

We accommodate sanitization in the model through the concept of the *sanitized flow*: a `flow path` whose output feature derives a portion of its output from an input feature whose security label is not dominated by the security label of the output feature. In other words, a sanitized flow *is not* checked against the star property. The implication is that the data is manipulated in some way that allows the lowering of its security label (e.g., the data is encrypted, degraded, obfuscated, aggregated, or inherently low-level data that is extracted from surrounding high-level data). Note that by modeling sanitization, we can determine that it is performed within the capabilities of a trusted subject. Because sanitization represents an exception to the security rules, it should be used cautiously and infrequently. The purpose of a sanitized flow in an AADL model is to mark those places in the system architecture where sanitization may occur so that the sanitization may receive the extra scrutiny that it deserves. Our analysis does not attempt to determine whether the sanitization is performed correctly.[4]

A `flow path` is marked as sanitized by using the property `Security::Downgrading`. Listing 4.1 shows the definition of the property. In our usage of the property, it only makes sense when used on a `flow path` specification. By default, the property value is `false`, indicating that a flow is not a sanitized flow. The intent is that specific flows are declared to be sanitized by explicitly associating the property with `true`, as shown in Listing 4.2. We assume that the security label of data type `Data.High` dominates that of data type `Data.Low`.

```
1 -- A downgrading flow path is one where the security level of some or
2 -- all of the  data passing through the input port of the flow path
3 -- is of a higher security level than that of the data passing
4 -- through the output port of the flow path.
5
6 Downgrading: aadlboolean => false applies to (flow);
```

Listing 4.1: The `Downgrading` Property Declaration

```
1 system Downgrader
2   features
3     input: in data Data.High;
4     output: out data Data.Low;
5   flows
6     downgraded_flow: input -> output {
7       Security::Downgrading => true;
8     };
9 end Downgrader;
```

Listing 4.2: An Example of Sanitization

[4] Chong and Myers describe a type system that prevents sanitization from occurring at the wrong time [8]. Such a type system would be useful for checking the source text implementation of a subprogram component that has a sanitized `out` parameter.

4.5.1.1 Analysis of Sanitized Flows

The primary purpose of marking a flow as sanitized is to exempt its destination port from the star property.

Rule 11 (Sanitized Flow) *When a flow path is sanitized, analysis does not check whether the security label of its destination dominates the security label of its source.*

The destination of a sanitized flow must still respect the simple security property: The sanitizing activity must still be within the capabilities of the component.

4.5.2 Sanitization Metrics

Because marking a flow as sanitized sidesteps the security model, the use of sanitization should be kept to a minimum. A system with many sanitized flows is suspicious. Of course, it is hard to say how many is too many. To this end, we recommend that analysis tools report metrics on the number of sanitized flows in a system so that sanitization is always visible to the modelers.

There is no need to declare a sanitized flow when the security label of the flow destination dominates the security label of the flow source. It is useful to identify such sanitized flows to prevent them from being accidentally (or, even worse, deliberately) misused. Analysis tools should issue at least a warning, if not an error, when a flow is unnecessarily marked as sanitized.

4.5.3 Caveats

It is beyond the scope of the current modeling approach to capture the exact nature of the sanitization, that is, whether it is achieved by encryption, obfuscation, or some other process. Accordingly, the present approach is not capable of assuring the correct use of the sanitized data once it leaves the component performing sanitization. For example, if the sanitization is performed by encrypting **top secret** data, there is no way to guarantee that the data is later decrypted to **top secret** data as we would expect. Without some kind of additional labeling of the data, there is nothing to prevent the system design from decrypting to **secret**, which would be a violation of the security rules.

5 Examples and Case Studies

This section contains complete examples of AADL models to illustrate two technical problems and show how they are represented using AADL.

5.1 Example 1

Our first example is a simple system shown in AADL graphical notation [15, Appendix D] in Figure 5.1. It has two producer systems, a computation system, and a consumer system. The diagram shows the four subsystems, their ports, the connections between them, and the flows through them. The data ports are labeled with their associated data classifiers to enhance the traceability of the security labels that are declared in the textual representation of the data types in Listing 5.1. The system includes flow specifications so that the star property can be enforced.

This example is intentionally straightforward to demonstrate the basic ideas of modeling security in AADL. In particular, the example demonstrates the following:

- Data ports receive their security attributes from their data classifier.
- Event ports receive their security attributes from their local property associations.
- Implementations inherit security attributes from their type.
- Security attributes are applied hierarchically, at each layer of the system.

This example, as well as the next, uses the security categories A, B, and C. Thus, we assume that the property type `Security_Type_Specifications::Caveat_Type` has been declared as

```
Caveat_Type: type enumeration (A, B, C);
```

The declarations for data types `X`, `Y`, and `Z` are shown in Listing 5.1 and contain the security property associations necessary to mark them as having the security labels (confidential, {A}), (confidential, {B}), and (confidential, {A, B}), respectively.

```
data X
    properties
        Security::Level => confidential;
        Security::Level_Caveats => (A);
end X;

data Y
    properties
        Security::Level => confidential;
        Security::Level_Caveats => (B);
end Y;

data Z
    properties
        Security::Level => confidential;
        Security::Level_Caveats => (A, B);
end Z;
```

Listing 5.1: Security-Annotated Data Classifier Declarations for the System in Figure 5.1

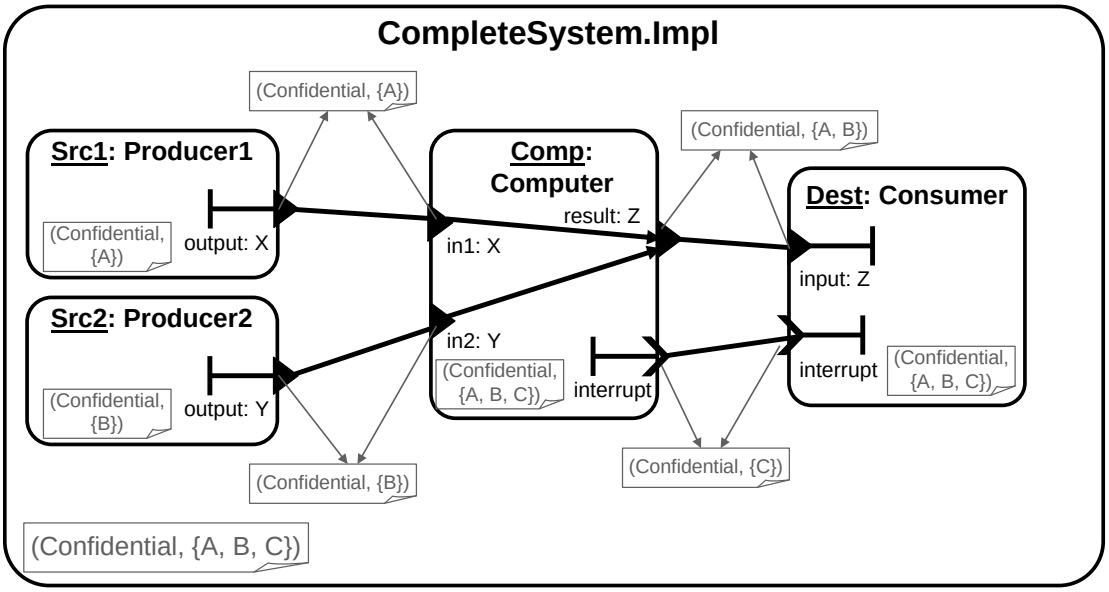

Figure 5.1: An Example System Annotated with Security Labels

The two producer system declarations are shown in Listing 5.2. Each system type contains the property associations necessary to declare the appropriate security label. No property associations are required for the data port `output` of each type because the security label is derived from the data classifier (`X` and `Y`, respectively) of the feature.

```
system Producer1
    features
        output: out data port X;
    flows
        src: flow source output;
    properties
        Security::Level => confidential;
        Security::Level_Caveats => (A);
end Producer1;

system Producer2
    features
        output: out data port Y;
    flows
        src: flow source output;
    properties
        Security::Level => confidential;
        Security::Level_Caveats => (B);
end Producer2;
```

Listing 5.2: Security-Annotated Producer Systems for the System in Figure 5.1

The declarations for the final two subsystems, `Computer` and `Consumer`, are shown in Listing 5.3. The security label of the event port `interrupt` in both systems must be declared using property associations on the feature declaration because event ports do not have an associated data classifier.

```
system Computer
    features
        in1: in data port X;
        in2: in data port Y;
        result: out data port Z;
        interrupt: out event port {
            Security::Level => confidential;
            Security::Level_Caveats => (C);
        };
    flows
        through1: flow path in1 -> result;
        through2: flow path in2 -> result;
        src: flow source interrupt;
    properties
        Security::Level => confidential;
        Security::Level_Caveats => (A, B, C);
end Computer;

system Consumer
    features
        input: in data port Z;
        interrupt: in event port {
            Security::Level => confidential;
            Security::Level_Caveats => (C);
        };
    flows
        snk1: flow sink input;
        snk2: flow sink interrupt;
    properties
        Security::Level => confidential;
        Security::Level_Caveats => (A, B, C);
end Consumer;
```

Listing 5.3: Security-Annotated `Computer` and `Consumer` Systems for the System in Figure 5.1

The declarations for the outer system, `CompleteSystem`, and its implementation, `CompleteSystem.impl`, are shown in Listing 5.4. The security label of the system is declared in the type and inherited by the implementation.

```
system CompleteSystem
    properties
        Security::Level => confidential;
        Security::Level_Caveats => (A, B, C);
end CompleteSystem;

system implementation CompleteSystem.Impl
    subcomponents
        src1: system Producer1;
        src2: system Producer2;
        comp: system Computer;
        dest: system Consumer;
    connections
        c1: data port src1.output -> comp.in1;
        c2: data port src2.output -> comp.in2;
        c3: data port comp.result -> dest.input;
        c4: event port comp.interrupt -> dest.interrupt;
end CompleteSystem.Impl;
```

Listing 5.4: Outer System Declaration for the System in Figure 5.1

5.2 Example 2

Here we present a second example of modeling security attributes in AADL using the example from Section 2.5. Figure 5.2 redisplays the example from Figure 2.2 but with the security label of s_4 corrected. This example illustrates the following:

- It emphasizes that AADL features represent data objects.
- It emphasizes that AADL components represent subjects.
- It emphasizes that AADL connections pass objects between subjects.
- It demonstrates a sanitized flow.
- It suggests an approach to mapping a complex subject–object diagram into an AADL model.

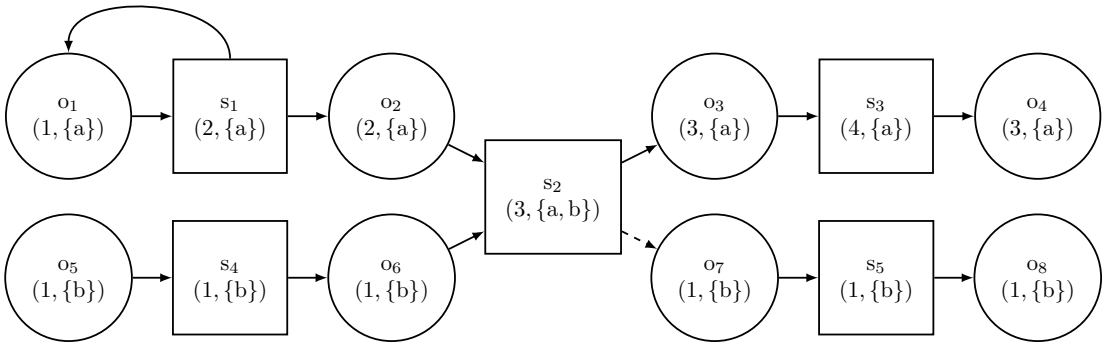

Figure 5.2: Example from Figure 2.2 with the Security Label of s_4 Corrected

In Figure 5.2, objects are shown explicitly, although in AADL the objects are implicitly represented as data port features. Thus, we create a data type for each of the objects in the figure to be used as the data classifiers for data ports. These types are shown in Listing 5.5. Specifically, there are four data component types:

1. The data type `Data_Unclassified` has the security label (unclassified, {A}) and is used for accessing object o_1.
2. The data type `Data_Confidential` has the security label (confidential, {A}) and is used for accessing object o_2.
3. The data type `Data_Secret` has the security label (secret, {A}) and is used for accessing objects o_3 and o_4.
4. The data type `Other_Data` has the security label (unclassified, {B}) and is used for accessing objects o_5, o_6, o_7, and o_8.

```
data Data_Unclassified
    properties
        Security::Level => unclassified;
        Security::Level_Caveats => (A);
end Data_Unclassified;

data Data_Confidential
    properties
        Security::Level => confidential;
        Security::Level_Caveats => (A);
end Data_Confidential;

data Data_Secret
    properties
        Security::Level => secret;
        Security::Level_Caveats => (A);
end Data_Secret;

data Other_Data
    properties
        Security::Level => unclassified;
        Security::Level_Caveats => (B);
end Other_Data;
```

Listing 5.5: Data Types for the Objects in Figure 5.2

For each subject shown in Figure 5.2, we declare a system type of the same name. Systems `S1`, `S2`, `S3`, `S4`, and `S5` are shown in Listing 5.6. To facilitate traceability to the original example, each port feature is named based on the object that it accesses (e.g., `o1_in` and `o4_out`), and flows are named based on the objects at the end points of the flow (e.g., `o1_to_o2`). The flow from o_2 to o_7 in system `S2`, `o2_to_o7`, is marked as sanitized by a `Downgrading` property association on the flow specification declaration. Note that there can be no flow from o_6 to o_3 in `S2` because the access $s_2 \rightsquigarrow o_3$ is not sanitizing.

```
system S1
    features
        o1_in: in data port Data_Unclassified;
        o1_out: out data port Data_Unclassified;
        o2_out: out data port Data_Confidential;
    flows
        o1_to_o1: flow path o1_in -> o1_out;
        o1_to_o2: flow path o1_in -> o2_out;
        o1_src: flow source o1_out;
    properties
        Security::Level => confidential;
```

```
        Security::Level_Caveats => (A);
end S1;

system S2
    features
        o2_in: in data port Data_Confidential;
        o6_in: in data port Other_Data;
        o3_out: out data port Data_Secret;
        o7_out: out data port Other_Data;
    flows
        o2_to_o3: flow path o2_in -> o3_out;
        o2_to_o7: flow path o2_in -> o7_out {
            Security::Sanitized => true;
        };
        o6_to_o7: flow path o6_in -> o7_out;
    properties
        Security::Level => secret;
        Security::Level_Caveats => (A, B);
end S2;

system S3
    features
        o3_in: in data port Data_Secret;
        o4_out: out data port Data_Secret;
    flows
        o3_to_o4: flow path o3_in -> o4_out;
    properties
        Security::Level => top_secret;
        Security::Level_Caveats => (A);
end S3;

system S4
    features
        o5_in: in data port Other_Data;
        o6_out: out data port Other_Data;
    flows
        o5_to_o6: flow path o5_in -> o6_out;
    properties
        Security::Level => unclassified;
        Security::Level_Caveats => (B);
end S4;

system S5
    features
        o7_in: in data port Other_Data;
        o8_out: out data port Other_Data;
    flows
        o7_to_o8: flow path o7_in -> o8_out;
    properties
        Security::Level => unclassified;
        Security::Level_Caveats => (B);
end S5;
```

Listing 5.6: System Types for the Subjects in Figure 5.2

The overall example is assembled as the implementation of a larger system (see Listing 5.7), specified by the system type `Example` and implemented by system implementation `Example.Impl`:

- The input object o_5 and the output objects o_4 and o_8 are the features `o5_in`, `o4_out`, and `o8_out`, respectively. Each connection is named based on the object that passes through it.
- Object o_1 is purely internal. It exists only as the feedback loop represented by the delayed data connection `o1_feedback`.
- Three flows are declared and implemented. In particular, there are two flow sources in system type `Example` starting from o_1, but whose paths diverge within s_2:
  1. `o1_to_o8`, which feeds back through s_1 and passes through s_2 as object o_2
  2. `o1_to_o4`, which feeds back through s_1 and passes through s_2 as the sanitized object o_7

```
system Example
    features
        o4_out: out data port Data_Secret;
        o5_in: in data port Other_Data;
        o8_out: out data port Other_Data;
    flows
        o5_to_o8: flow path o5_in -> o8_out;
        o1_to_o8: flow source o8_out;
        o1_to_o4: flow source o4_out;
    properties
        Security::Level => secret;
        Security::Level_Caveats => (A, B);
end Example;

system implementation Example.Impl
    subcomponents
        s1: system S1;
        s2: system S2;
        s3: system S3;
        s4: system S4;
        s5: system S5;
    connections
        o1_feedback: data port s1.o1_out -> s1.o1_in {
            Timing => delayed;
        };
        o2: data port s1.o2_out -> s2.o2_in;
        o3: data port s2.o3_out -> s3.o3_in;
        o4: data port s3.o4_out -> o4_out;
        o5: data port o5_in -> s4.o5_in;
        o6: data port s4.o6_out -> s2.o6_in;
        o7: data port s2.o7_out -> s5.o7_in;
        o8: data port s5.o8_out -> o8_out;
    flows
        o5_to_o8: flow path o5_in -> o5 ->
            s4.o5_to_o6 -> o6 ->
            s2.o6_to_o7 -> o7 ->
            s5.o7_to_o8 -> o8 -> o8_out;
        o1_to_o8: flow source s1.o1_src -> o1_feedback ->
```

```
39              s1.o1_to_o2 -> o2 ->
40              s2.o2_to_o7 -> o7 ->
41              s6.o7_to_o8 -> o8 -> o8_out;
42          o1_to_o4: flow source s1.o1_src -> o1_feedback ->
43              s1.o1_to_o2 -> o2 ->
44              s2.o2_to_o3 -> o3 ->
45              s3.o3_to_o4 -> o4 -> o4_out;
46  end Example.Impl;
```

Listing 5.7: Top-Level System Specification for the Example in Figure 5.2

6 Conclusion

The concept of subjects and objects, where subjects operate on objects by permissible access operations (read, execute, append, write), enables us to model and validate security at both the software and execution platform levels. At the software level, we can view processes, threads, and software components as subjects and data objects as objects.

Determining the viability of a system, given confidentiality requirements of data objects and security clearance by users, one can see validation as a two-step process: (1) validation of the software architecture followed by (2) validation of the system architecture, where the software architecture is mapped to execution platform components. Validating the software requires us to do all of the following:

- Identify the data elements that we want to protect (objects).
- Determine their security requirements.
- Identify the components (software components, processes, threads) that should be allowed to access the objects.
- Confirm that the access is as specified by access operations.

Thus, we can ensure that data elements are accessed only by authorized users and that confidentiality (as given by security labels) and integrity (as given by access operations) are enforced.

Mapping the entities of a software architecture (e.g., processes, threads, and partitions) to an execution platform architecture consisting of processors, communications channels, memory, and the like enables us further to ensure that the platform architecture supports the required security labels. Consider the scenario of two communicating processes, both requiring a high level of security as the data objects require secret clearance. Furthermore, the system platform in this scenario consists of a set of CPUs with hardware support for various algorithms that encrypt messages before network transmission. By modeling the system, we can represent and validate that processes and threads (now considered to be objects) will be executed (access mode) on CPUs (subjects) with adequate encryption support. We can also validate that CPUs (objects) communicate data (access modes of writing and reading) over appropriately secured communications channels. In a similar fashion, we can enforce design philosophies in which only processes of the same security label are allowed to coexist within the same CPU or partition or in which they can write to a secured memory.

In this report, we have demonstrated how model-based engineering can support early modeling and validation of security. Specifically, using the AADL, we have specified common and well-defined security attributes and represented them in the AADL models. The adopted notion is primarily based on the Bell–LaPadula model. Using the AADL and the Bell–LaPadula and extended sibling models, one can model and validate security according to flow-based approaches, Bell–LaPadula, Chinese wall, and role-based access. To support security analysis, we have taken established criteria from the Bell-LaPadula model and defined additional criteria that allow us to evaluate how viable a system is to enforce security, given confidentiality requirements of data objects and security clearance by users. For example, we can ensure that processes and threads are mapped to appropriate hardware, communicate over secured channels, and reside/store data in protected memory.

The overall objective of a secure system implies that security clearances are given conservatively (as opposed to generously). To this end, we can analyze models to derive the minimum security clearance on components in the model. Or to put it differently, we can use the notion

of subjects and objects to determine the minimum security clearance for a subject based on the requirements of the objects being accessed by the specific subject. By also pointing out differences between actual security clearances and the minimum security clearance required, a system designer can evaluate how effective and tight security is. By providing mechanisms to ensure that sanitization is conducted within allowed boundaries, the designer can analyze and trace these relatively more threatening security risks, as sanitizing actions are permitted exemptions of security criteria and rules and, as such, should be minimized in the system.

References/Bibliography

URLs are valid as of the publication date of this document.

[1] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. In *Proceedings of the Third IEEE Information Survivability Workshop*, pages 7–12, 2000.

[2] D. E. Bell and L. J. LaPadula. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol. II, The MITRE Corporation, 1973. ESD-TR-73-278-II.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. I, The MITRE Corporation, 1973. ESD-TR-73-278-I.

[4] D. E. Bell and L. J. LaPadula. Secure computer systems: A refinement of the mathematical model. Technical Report MTR-2547, Vol. III, The MITRE Corporation, 1974. ESD-TR-73-278-III.

[5] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and MULTICs interpretation. Technical Report MTR-2997 Rev. 1, The MITRE Corporation, 1976.

[6] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation, 1975.

[7] David F. C. Brewer and Michael J. Nash. The Chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[8] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 198–209, 2004.

[9] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[10] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[11] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL*. Addison-Wesley, 2013.

[12] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(1):143–147, January 1974.

[13] David F. Ferraiolo and D. Richard Kuhn. Role-based access controls. In *15th National Computer Security Conference*, pages 554–563, 1992.

[14] Jörgen Hansson, Peter Feiler, and Aaron Greenhouse. Enforcement of quality attributes for net-centric systems through modeling and validation with architecture description languages. In *Fourth European Congress Embedded Real Time Software and Systems*, 2008.

[15] SAE International. Architecture Analysis & Design Language (AADL). Aerospace Standard AS5506C, SAE International, 2017.

[16] Carl E. Landwehr. A survey of formal models for computer security. Technical Report 8489, Naval Research Laboratory, 1981.

[17] T. Y. Lin. Chinese wall security policy—an aggressive model. In *Fifth Annual Computer Security Applications Conference*, pages 282–289, 1989.

[18] Terry Mayfield, J. Eric Roskos, Stephen R. Welke, John M. Boone, and Catherine W. McDonald. Integrity in automated information systems. IDA Paper P-2316, Institute for Defense Analyses, 1991.

[19] John McLean. Security models and information flow. Technical report, Center for High Assurance Computing Systems, Naval Research Laboratory, Washington, DC, 1990.

[20] John McLean. Security models. In *Encyclopedia of Software Engineering*, volume 2, pages 1136–1145. John Wiley & Sons, Inc., 1994.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	March 2021	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Modeling and Validating Security and Confidentiality in System Architectures	FA8702-15-D-0002

6. AUTHOR(S)

Aaron Greenhouse, Jörgen Hansson, and Lutz Wrage

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2021-TR-004

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

The importance of security in computer and information systems is increasing as network-connected computer systems become more ubiquitous. The objective of security is to verify that the computing platform is secured and that data and information are properly accessed and handled by users and applications, ensuring data confidentiality and integrity. To develop a framework for modeling and verifying security as a data quality attribute, designers need to identify parameters and variables with the expressive power to capture and represent security models and determine the type of analysis to enable. This report presents an approach for modeling and validating confidentiality based on the Bell–LaPadula security model using the Architecture Analysis and Design Language (AADL). The report describes the Bell–LaPadula security model and elaborates how security and Bell–LaPadula attributes are mapped to concepts and represented in AADL. It then describes modeling and validating security in AADL models, considering conditions that need to be enforced for a system to ensure conformance to the Bell–LaPadula security policy. It also presents the analysis capabilities provided by AADL and examples modeled in AADL.

14. SUBJECT TERMS	15. NUMBER OF PAGES
architecture modeling, architecture analysis, confidentiality, security, Bell–LaPadula model	44

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102