

# *Handbook for the Computer Security Certification of Trusted Systems*

---

Chapter 1: Overview

Chapter 2: Development Plan

Chapter 3: Security Policy Model

Chapter 4: Descriptive Top-Level Specification

Chapter 5: Design

Chapter 6: Assurance Mappings

Chapter 7: Implementation

Chapter 8: Covert Channel Analysis

Chapter 9: Security Features Testing

Chapter 10: Penetration Testing



*NRL Technical Memorandum 5540:081A, 24 Jan 1995*

For additional copies of this report, please send e-mail to [landwehr@itd.nrl.navy.mil](mailto:landwehr@itd.nrl.navy.mil), or retrieve PostScript via <http://www.itd.nrl.navy.mil/ITD/5540/publications/handbook> (e.g., using Mosaic).

**Assurance Mappings:**  
**A Chapter of the**  
*Handbook for the Computer Security Certification of*  
*Trusted Systems*

John M<sup>C</sup>Hugh, Principal Investigator

Department of Computer Science  
Portland State University  
Portland, Oregon

Charles Payne  
Naval Research Laboratory  
Washington, DC

and

Charles R. Martin  
The University of North Carolina  
Chapel Hill, North Carolina

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Organization of This Chapter . . . . .	3
<b>2</b>	<b>Overview of the Assurance Mappings</b>	<b>4</b>
2.1	What is an assurance mapping? . . . . .	4
2.1.1	DTLS to Policy Mapping . . . . .	5
2.1.2	Code to DTLS Mapping . . . . .	5
2.2	Where does the assurance mapping fit in the life cycle? . . . . .	6
2.3	What risks are dealt with through an assurance mapping? . . . . .	7
2.4	What resources are required? . . . . .	8
<b>3</b>	<b>Mapping the DTLS to the Security Policy</b>	<b>10</b>
3.1	Identifying the target security policy . . . . .	10
3.2	Mapping the SPM to the COMPUSEC Policy . . . . .	11
3.3	Mapping the DTLS to the SPM . . . . .	13
<b>4</b>	<b>Mapping the Implementation to the DTLS</b>	<b>15</b>
4.1	Overview . . . . .	15
4.1.1	Baby steps vs. Giant Steps . . . . .	15
4.1.2	Maintenance . . . . .	16
4.1.3	The role of standards like 2167A and SDD . . . . .	16
4.2	Decomposing the TCB . . . . .	16
4.2.1	Analysis of the Code . . . . .	17
4.2.2	Graph based methods . . . . .	17
4.2.3	Dealing with data structures . . . . .	18
4.2.4	Code and Data that do not map up . . . . .	18
4.3	DTLS Decomposition . . . . .	19
4.3.1	Identifying mappable specifications in the DTLS . . . . .	19
4.3.2	Analysis of a DTLS . . . . .	19
4.4	Identifying TCB to DTLS Correspondence . . . . .	20
4.4.1	Hooks . . . . .	20
4.4.2	Constructing hooks in the mapped code base . . . . .	21
4.4.3	Hooks in the DTLS . . . . .	21
4.4.4	Tabular methods . . . . .	21
4.4.5	Databases . . . . .	22
4.5	Making the Mapping Argument . . . . .	22

Eventually, there are two “machines”. On the one hand there is the physical machine that . . . can go wrong . . . . On the other hand there is the abstract machine . . . , the “thinkable” machine for which the programmer programs and with respect to which the question of program correctness is settled.

Originally I viewed it as the function of the abstract machine to provide a truthful picture of the physical reality. Later, however, I learned to consider the abstract machine as the “true” one, because that is the only one we can “think”; it is the physical machine’s purpose to supply “a working model”, a (hopefully!) sufficiently accurate physical simulation of the true, abstract machine.

Edsger W. Dijkstra  
**A Discipline of Programming**  
1976

## 1 Introduction

A system must satisfy strict assurance requirements for successful evaluation at the B3 class of the *Trusted Computer System Evaluation Criteria (TCSEC)* [9]. Many of these requirements are essentially requirements on process or on documentation. Unlike many other areas, TCSEC requirements are a job of selling to an uncertain buyer: TCSEC evaluation is a matter of convincing a responsible team of evaluators that the system can be trusted to manage sensitive data in an appropriate fashion.

The definition of an “appropriate fashion” comes from the system’s security policy, and from security requirements which are detailed in the descriptive top-level specification (DTLS). The detailed information necessary to convince the evaluators is provided in the form of an *assurance mapping* between the trusted computing base (TCB), which implements the security-related functions, and the security policy, which defines the secure behavior of those functions. The job of the developers is to sell the trustworthiness of the system to the evaluators using these mappings by making a convincing argument that the TCB correctly enforces the security policy. In other words, the developer must convince the evaluators that the physical machine embodied in the TCB is an accurate simulation of the true, abstract machine prescribed by the security policy. The abstract machine is first described in the Security Policy Model.

This chapter identifies the certification goals for the assurance mappings. Unlike other certification deliverables, the assurance mappings are generated throughout the development life-cycle. They are a collection of documents rather than a single document, and a member of the collection may be part of another certification deliverable, such as the Security Policy Model. The developer must present the mappings as a coherent whole.

This chapter also describes methods that can be used to hand-examine the code for minimality, because the TCSEC also requires the TCB to be “minimized” in the sense that it contains nothing not required to correctly implement the security-related functions. *Proving* that this requirement has been met is very difficult to do.

## **1.1 Organization of This Chapter**

While the assurance mappings are generated throughout almost the entire development effort—from the security modeling exercise to the implementation in source code—and while they address all security-relevant requirements from the most abstract to the very low-level and detailed, the process of constructing these mappings changes little throughout the development process. The same technique is applied repeatedly. Section 2, “Overview of the Assurance Mappings,” discusses the technique and answers some commonly asked questions. For convenience, we divide the assurance mappings into two major parts. Section 3 addresses the DTLS to security policy mapping, and Section 4 addresses the TCB code to DTLS mapping. There is a dearth of literature in this area. Those references that have come to our attention are listed in the bibliography.

## 2 Overview of the Assurance Mappings

### 2.1 What is an assurance mapping?

Given two representations  $x$  and  $y$  of a structure, if we can express their behavior as assertions  $A_x$  and  $A_y$ , respectively, then we say that  $y$  *corresponds to*  $x$  if

$$A_y \Rightarrow A_x.$$

In other words, if  $x$  behaves in a particular way, then  $y$  must also, but  $x$  is allowed to exhibit some behavior that  $y$  does not. Viewed another way, the traces in which  $y$  may engage are a subset of the traces in which  $x$  may engage. In our case,  $A_x$  and  $A_y$  are assertions on security-critical behavior.

In reality, assurance mappings are much less mathematical than suggested above, but we must understand the semantics of the exercise rigorously in order to determine that it is being performed correctly. A simpler approach is described below and illustrated in Figure 1.

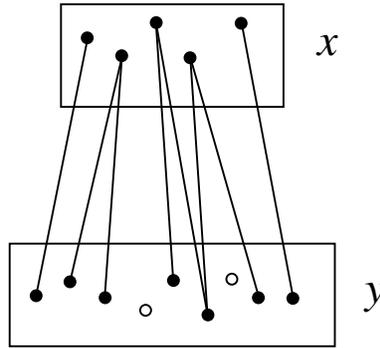


Figure 1: Abstract Mappings

1. Identify the critical elements of  $y$  (illustrated by the closed circles in Figure 1).
2. Derive a correspondence between the critical elements of  $y$  and all of  $x$ . Show that non-critical elements of  $y$  (the open circles in Figure 1) do not need to be mapped to  $x$ . Ensure that all of  $x$  is mapped to  $y$ , as illustrated.
3. Attempt to express the critical properties that are satisfied by  $x$ , i.e.,  $A_x$ , in terms of  $y$ , i.e.,  $A_y$ , using the element correspondence.
4. Examine  $y$  carefully, and demonstrate informally that it satisfies the conditions of  $A_y$ .

Therefore, an assurance mapping consists of four components:

- An exhaustive identification of mappable elements of the primary structure.
- An exhaustive identification of the mappable elements in a secondary structure.
- A mapping: an identification of the elements of the primary structure with subsets of the elements of the secondary structure.

- A rigorous and convincing argument that the properties of the primary structure are reflected in the secondary structure.

All of the mappable elements of the primary structure, e.g.,  $x$ , are considered critical. Most of the mappable elements of the secondary structure, e.g.,  $y$ , are considered critical. While non-critical elements are not mapped, their presence in  $y$  must be justified.

### 2.1.1 DTLS to Policy Mapping

For the DTLS to policy mapping, we proceed in two steps. First we demonstrate that the Security Policy Model (SPM) is an accurate restatement of the policy. It may not be true that the SPM is a *complete* restatement of the security policy, because for reasons of simplicity, the SPM may not restate all of the requirements in the policy. For example, the SPM may not include a trusted audit requirement. The expense of formalization and proof may preclude the developer from modeling any requirements that do not address specifically the *prevention* of security violations. We noted in the Handbook chapter on the SPM that completeness of the SPM may be negotiated between the developer and the evaluator. Consequently, a slight interpretation of our mapping procedure is necessary for the SPM to security policy mapping. In the steps and figure above, let  $x$  be the SPM and  $y$  be the security policy. Steps 1 and 2 remain the same. For steps 3 and 4, reverse the occurrences of  $x$  and  $y$ .

Second, we demonstrate that the kernel calls and system structures of the DTLS are consistent with the instructions and the computational framework of the SPM. The kernel calls and system structures of the DTLS represent requirements on the underlying TCB. The SPM's instructions are a computational representation of the formal assertions. The DTLS kernel calls should restate the instructions at a lower level of abstraction. The mapping between the DTLS and the SPM will resemble Figure 1 when  $x$  is the SPM and  $y$  is the DTLS. In other words, all instructions in the SPM should be mapped to the calls of the DTLS, but some calls in the DTLS may not map to SPM instructions. The unmapped DTLS calls must be justified in the mapping.

### 2.1.2 Code to DTLS Mapping

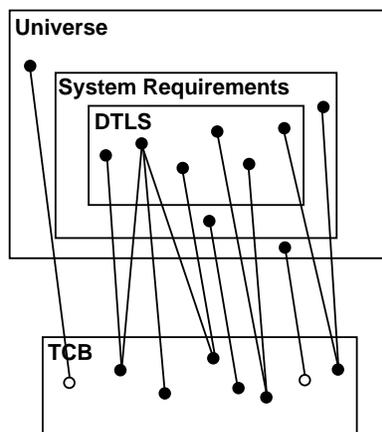


Figure 2: A Code to DTLS Mapping

Applying the abstract mapping process of Figure 1 to the code to DTLS mapping, results in the situation illustrated in Figure 2. The DTLS can be viewed as though it is contained within a more general set of System Requirements. The System Requirements are, in turn, contained within an arbitrary Universe of requirements. All of the critical features of the TCB (the closed circles) are either responding to some requirement of the DTLS or to some System requirement that is not part of the DTLS but must be contained in the TCB for some reason. We should not find any TCB functionality that responds solely to requirements from the external universe (as represented by the open circles in the figure).

Conceptually, the system overall could be described in terms of an assurance mapping between all the system specifications and all the code implementing those specifications. We are concerned with constructing an assurance mapping between the components of the TCB and the mappable elements, i.e., the requirements, in the DTLS. The mappable elements for the DTLS were identified earlier during the DTLS to SPM mapping. The TCB source code must now be decomposed into a collection of separately identified, discrete components.

Each component of the TCB must be identified as contributing to the satisfaction of *some* requirement. Many of these requirements are contained within the DTLS; some of these requirements may not be. *All* requirements in the DTLS must be identified with components of the TCB that implement those requirements. This means that the TCB will actually contain two parts: a collection of components that are mapped to components of the DTLS, and those that are not.

Developing confidence in the TCB breaks similarly into two parts: we must first argue that those TCB components mapped to the DTLS requirements satisfy those requirements. It is equally important to identify those TCB components that are *not* mapped to requirements in the DTLS, and then answer two questions about each: (1) what other system-wide requirements do these components satisfy; and (2) why must these components be in the TCB?

## 2.2 Where does the assurance mapping fit in the life cycle?

Assurance mappings tie together views of the system at different levels of detail and abstraction. Because the validity of a mapping can be affected by changes to either of the items for which correspondence is being shown, the actual mapping process and the production of the final mapping documents should take place late in the development portion of the life cycle process. The project manager should ensure that great effort is not spent producing mappings that will require extensive revisions due to changes in the system representations being mapped. Experience shows that substantial code changes can and do occur during system integration and testing, no matter how undesirable that may be. Thus, the code to DTLS mapping should be deferred until it is clear that the code will not undergo substantial change.

On the other hand, the system security policy is usually defined early in the development process. If a rational development approach is followed, the DTLS will also be an early deliverable. The mapping between the DTLS and the security policy can be performed whenever both have been defined and it is clear that neither will undergo substantial revision. A cautionary note is in order, however. Although it is undesirable, it is all too common to separate the system development responsibilities from the security engineering responsibilities. As a result, the TCSEC deliverables, i.e., the security policy, DTLS, etc., are developed in parallel with the system implementation—creating two views of the system that diverge due to poor or

non-existent communications between the two development teams. If this occurs, the DTLS and the security policy may have to change after the implementation is finished in order to align them with the actual system. If this kind of approach is followed, performing the mapping between the policy and the DTLS in the absence of the implementation may incur substantial risks of rework.

This is not to say that all mapping work should be ignored until the items that are to be mapped are finished. As the security policy, DTLS, and implementation are being created, the developers must constantly be aware that the mapping will be required at some point. As work products such as the security policy, the SPM, the DTLS, and the design for the implementation are being developed, each should be evaluated with respect to the mapping requirement. Design tradeoff analyses should include ease of mapping to the DTLS as a factor in considering alternative designs. The implementation should be structured to facilitate the mapping. The DTLS should be organized so as to support both the code and security policy mappings.

### **2.3 What risks are dealt with through an assurance mapping?**

There are two classes of risks that are (at least partially) mitigated by the assurance mapping process. The first is the risk that security flaws have been deliberately introduced into the system maliciously, to create a subvertable system. The second is the risk that errors in the design or implementation of the system could result in its compromise. Following the taxonomy of Landwehr[6], we see a wide variety of flaws that could be detected in the mapping process. These flaws are divided into two primary classes: intentional and inadvertent.

In particular, the mapping process is one place in which a search for intentional, malicious flaws can be made. These would typically include Trojan Horse and Trapdoor mechanisms and could include Logic/Time Bomb mechanisms. In the intentional, nonmalicious category, the mapping process is not intended to address covert channels, though it might identify mechanisms that could be used to construct covert channels. It could identify flaws in the miscellaneous category, intentional, nonmalicious, other, but many of the examples given in this category are based on very subtle flaws that would be more likely to be discovered during penetration or security testing. In general, the analysis of the implementation that is required during the code to DTLS mapping should be sufficiently thorough to minimize the possibility that intentional malicious code has been introduced into the implementation. It should also aid in identifying intentional, nonmalicious flaws.

The mapping process should help identify many inadvertent flaws that may arise during implementation. Because each part of the implementation must be shown to respond to either a DTLS requirement or to some other system functional requirement that must be a part of the TCB, the inspection process involved in the mapping will focus on a variety of potential problems. These are variously categorized as:

- validation errors that occur when a program fails to check assumptions made about parameters,
- domain errors in which assumptions about protection domains are violated,
- serialization or aliasing errors that permit unexpected changes in validated parameters either because they are accessible in multiple threads of control or by different names in the same thread of control (time of check to time of use errors),

- identification/authorization errors in which the identity or authority of an invoking agent is inadequately checked,
- boundary condition errors in which constraints on resource allocation are not adequately checked, and
- other exploitable logic errors, which is a catch-all for errors that do not fall into one of the above categories.

If carried out conscientiously, the code to DTLS mapping will have a good chance of identifying implementation errors in all of the categories listed. Errors in the DTLS or in the security policy can also be identified during the mapping process. Whenever the mapping effort uncovers an inconsistency between the entities being mapped, the inconsistency must be resolved. It is not necessarily the case that the more concrete representation is at fault and there is always the possibility that the implementor's intuition led to a correct solution in spite of a faulty specification. It is also possible for the mapping process to identify errors even though a faulty specification has been faithfully implemented. For this to happen, we must rely on the experience and skill of the mapping team and their ability to identify aspects of the system that "just don't feel right."

## 2.4 What resources are required?

The mapping process requires both time and manpower to perform. The code to DTLS mapping will consume far more resources than the DTLS to security policy mapping. The amount of time that should be allocated is difficult to estimate as it depends on a variety of factors including the extent to which the code and DTLS are similarly structured, the amount of detail in the security policy and in the DTLS, etc. The effort is substantial and we do not have sufficient historical data to accurately bound the level of effort involved. As a rule of thumb, the effort allocated for a correspondence mapping on a pair of entities (code and DTLS or DTLS and security policy) should be at least five times that allocated for doing a detailed walkthrough or inspection on the more concrete of the entities involved in the mapping. In cases where the entities being mapped involve greatly different levels of abstraction or where the entities have substantially different structures, much more effort will be required.

The team that performs the mapping should be familiar with both the system requirements and with the implementation. At the same time, the need to consider the possibility of intentional, malicious flaws dictates that developers (or specifiers) should not be responsible for mapping their own work. In a suitably large system, it should be possible to have developers responsible for one component or subsystem perform the mapping for another. An alternative is to have the mapping performed by an external group such as the developer's quality assurance organization or an independent verification and validation contractor. In any event, the mapping team must be capable of understanding and explaining to the evaluators the structure and functioning of each of the entities being mapped. Having the mapping performed by an independent group can also serve to examine the quality of the implementation and the usability of its documentation. As is noted in the Implementation Evaluation Guideline chapter of this handbook [3], these are important factors that are not directly addressed by the evaluation criteria.

The evaluator has certain obligations in evaluating a mapping that is presented by the developers. In particular, the evaluator must be prepared to interpret the TCSEC's mapping requirements for the trusted system and the mission that it supports. The evaluator must be familiar with the specification techniques used at each stage of the life cycle, and he or she should be aware of alternative mapping techniques. Finally, the evaluator must understand the purpose and scope of the assurance argument.

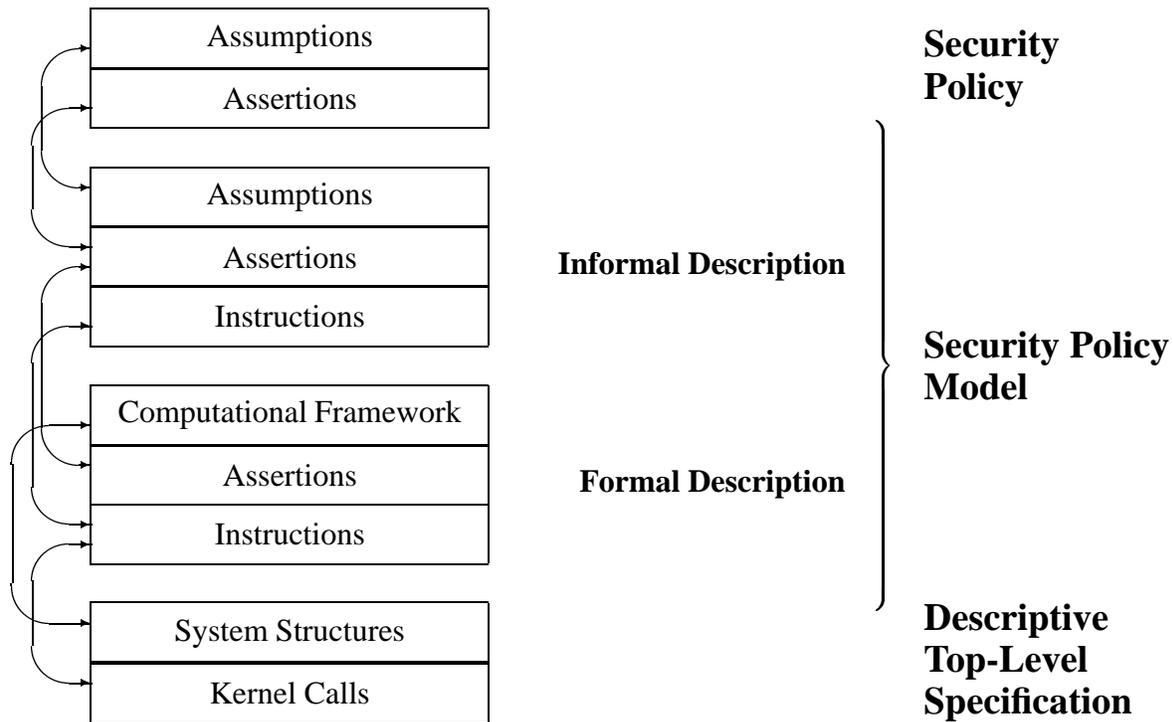


Figure 3: Mapping the DTLS to the Security Policy

### 3 Mapping the DTLS to the Security Policy

The TCSEC imposes the following requirements on the DTLS, SPM and security policy for a B3 TCB.

- The formal model shall be “proven that it is sufficient to enforce the security policy”.
- “The specific TCB protection mechanisms shall be identified and an explanation given to show that they satisfy the model.”

This section describes the assurance mappings, highlighted in Figure 3, necessary to satisfy these requirements. The arrows along the left illustrate the mappings that we will consider. We proceed from top to bottom to form the argument that the DTLS is sufficient to satisfy the security policy. First, however, we must understand *which* security policy the DTLS must satisfy.

#### 3.1 Identifying the target security policy

As Sterne noted, there are many definitions of the term “security policy” [14]. Sterne distinguishes three major types of policy:

- *security policy objectives* — constraints imposed from outside an organization, such as by a government entity,

- *organizational security policy* — constraints that an organization imposes on its own practices, and
- *automated security policy* — constraints that an organization places on the computer that supports the organization.

Unfortunately, two critical elements are missing from his policy framework: the mission in which the organization engages (and which the computer supports), and the use of information security (INFOSEC) countermeasures (instead of just computer security countermeasures) to support that mission. We modified Sterne’s framework to include these elements [11]. Figure 4 illustrates our modified policy framework and the derivation of a computer security (COMPUSEC) policy from the security policy objectives.

Central to our policy framework is the INFOSEC policy that defines requirements on the INFOSEC countermeasures. These countermeasures can be drawn from any of the INFOSEC disciplines, such as TEMPEST, administrative security, COMPUSEC, communications security, physical security, personnel security and others. Together they address the threats targeted by the INFOSEC policy, which in turn addresses threats to the organization’s mission.

The COMPUSEC policy defines requirements on the COMPUSEC countermeasure, i.e., the trusted system. It identifies the assertions that must be enforced by the trusted system, and it identifies any assumptions about enforcement provided by the trusted system’s environment (i.e., the other INFOSEC countermeasures). The COMPUSEC policy is the target of our assurance mapping.

### 3.2 Mapping the SPM to the COMPUSEC Policy

Effective communication of the SPM includes an informal description of the model, a formal description, and a validity argument. The informal description includes a user’s view of the system being modeled, a set of assertions that the system must enforce, the set of assumptions that underlie those assertions, and the behavior-generating instructions for the model. The formal description restates the assertions and the instructions in a mathematical framework.

The validity argument embodies the assurance mapping between the SPM and the COMPUSEC policy. The assertions and assumptions of the informal description are shown to be derived from the COMPUSEC policy, the formal assertions are shown to correspond to the informal assertions, and the behavior-generating, formal instructions are shown to satisfy the constraints of the formal assertions. Each of these elements of the validity argument is addressed below.

#### 1. *Map the informal description of the SPM to the COMPUSEC policy.*

To begin, the COMPUSEC policy should be examined for explicit and implicit assertions and assumptions. They will be the targets of the mapping. Then a correspondence argument, e.g., a table, should be constructed that maps the assertions and assumptions of the informal description of the SPM to the assertions and assumptions of the COMPUSEC policy. The mapping should include any justification for requirements in the COMPUSEC policy that are not restated in the SPM.

#### 2. *Map the formal description to the informal description.*

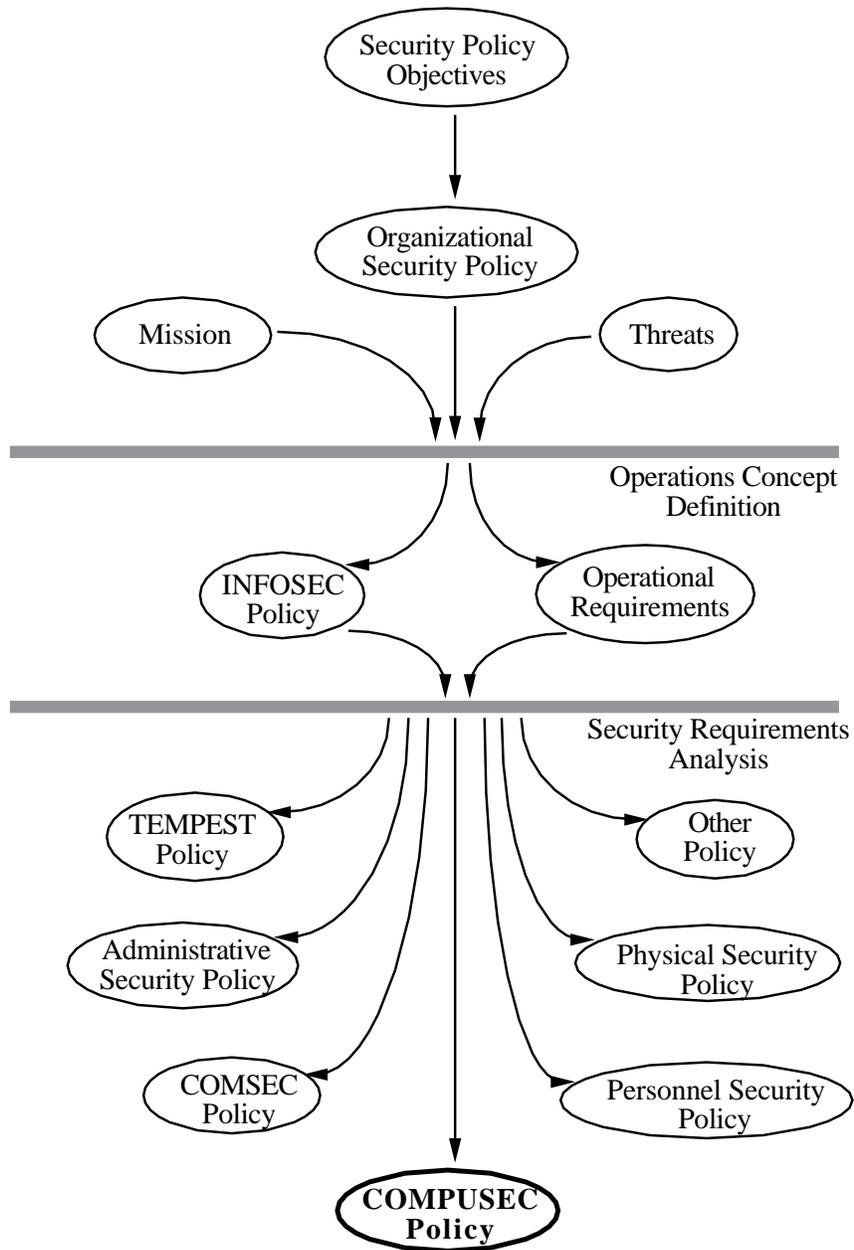


Figure 4: Derivation of the COMPUSEC Policy

In other words, they should be two descriptions of the same entity! This will not be a rigorous mapping; instead, as the formal description is constructed, it should be obvious how it relates to the informal description.

### 3. *Demonstrate the sufficiency of the SPM.*

We demonstrate that the SPM is sufficient to enforce the COMPUSEC policy by proving that the behavior generated by the SPM’s formal instructions satisfies the constraints of the formal assertions (which are derived from the COMPUSEC policy). This is one of the most difficult tasks of the modeling effort. The assertions are *post-conditions* for each instruction. We prove that an empty sequence of instructions satisfies the assertions, then we prove that, for each instruction, if an arbitrary sequence of length  $n$  satisfies the assertions, then the sequence with the chosen instruction appended also satisfies the assertions. This is the only formal proof required in the assurance argument for a TCSEC B3 evaluation.

If these three steps are completed successfully, then by transitivity we can assert that the formal instructions satisfy the COMPUSEC policy. The formal instructions are a strong foundation for creating a secure design for the trusted system: they are expressed in a computational framework that models the trusted system, and they manipulate abstract representations of the data structures upon which the security of the system will be based. The primary task hereafter is demonstrating that the formal instructions are implemented correctly.

## 3.3 Mapping the DTLS to the SPM

The assurance mapping from the DTLS to the SPM is a demonstration that the kernel calls and system structures of the DTLS correspond to the formal instructions and computational framework, respectively, of the SPM. We must argue that the kernel calls are an appropriate refinement of the formal instructions. Then we will argue in Section 4 that the TCB code correctly implements the requirements specified in the kernel calls.

However, our task is not so straightforward if the SPM does not include behavior-generating instructions. McLean [7] notes that the term “security model” has two distinct uses: as a particular mechanism for enforcing confidentiality and as a specification of the system’s confidentiality requirements. The latter use, McLean continues, is not a “model” at all since it does not specify a particular mechanism. The *mechanism* to which McLean refers are the behavior-generating instructions. Millen [8] calls an SPM with instructions a *concrete* SPM, while an SPM without instructions is called an *abstract* SPM. Whether the SPM is concrete or abstract impacts the assurance mapping significantly.

The Bell and LaPadula SPM [1] is a concrete SPM. It contains eleven instructions, or rules of operation, that may be mapped to top-level TCB functions. Bell and LaPadula’s exposition of the SPM includes an interpretation for the Multics architecture. The interpretation is a mapping between the Multics TCB functions and the rules of the SPM. The Secure Military Message System (SMMS) SPM [5], on the other hand, is an abstract SPM. The SMMS SPM defines an abstract state transition function  $T$  that represents all possible transitions in which the system could engage.

There are several reasons for choosing an abstract SPM over a concrete SPM. Good software engineering suggests that design decisions should be delayed as long as possible. Formu-

lating the instructions so that the assurance mapping is simplified assumes some knowledge of the TCB architecture (at least at the DTLS level of abstraction), so if the SPM can be defined without these assumptions, it will represent more implementations.

A trusted system development — with many traps to trip its progress — may benefit more from a concrete SPM than from an abstract SPM. The analysis required to specify the instructions represents the kind of forethought that is needed to avoid the traps. However, if the developer chooses to define an abstract SPM, then one of the following scenarios must occur to complete the assurance mappings. The developer must

- create a set of instructions, show that they satisfy the SPM’s assertions, and then map the kernel calls to these instructions, or
- map the SPM’s computational framework to the system structures of the DTLS, interpret the formal assertions for the system structures using the mapping, then prove that the kernel calls satisfy the interpreted assertions.

As we shall see below, the developer engages the second scenario anyway when he demonstrates the sufficiency of the DTLS, but we still believe that creating the instructions initially is beneficial.

Given a concrete SPM then, the following steps complete the mapping from the DTLS to the SPM.

1. *Identify the kernel calls and system structures of the DTLS.*

Pay particular attention to the effect of each kernel call and the system structures that it manipulates.

2. *Identify the instructions and computational framework for the SPM.*

3. *Construct the mapping.*

Determine the correspondence between the DTLS system structures and the SPM computational framework first. If an instruction affects particular elements of the computational framework, then the corresponding kernel call should manipulate the corresponding system structures. Kernel calls that do not map to instructions must be justified.

4. *Demonstrate the sufficiency of the DTLS.*

Once the kernel calls that correspond to the formal instructions are identified, then they must be shown to exhibit the same behavior as the instructions. Given the mapping between the system structures and the computational framework, it should be possible to interpret the SPM’s formal assertions roughly for the DTLS and argue informally that the kernel calls satisfy those constraints.

## 4 Mapping the Implementation to the DTLS

The TCSEC imposes the following requirement on the assurance mapping between the TCB implementation and the DTLS.

The TCB implementation (i.e., in hardware, firmware, and software) shall be informally shown to be consistent with the DTLS. The elements of the DTLS shall be shown, using informal techniques, to correspond to the elements of the TCB.  
[9]

### 4.1 Overview

This section acts as a catalog of tools and techniques for performing the mapping. It covers a variety of techniques that are useful in achieving the goal of demonstrating that the code of the system TCB performs exactly and only the functions required by the DTLS.

The purpose of performing a code to DTLS mapping is to develop evidence that will satisfy the evaluators that the system in question is, in fact trustworthy. A large part of this process is organizing and presenting the evidence gathered during the actual mapping into a coherent and well documented argument that supports the developer's contention that the system is what it purports to be. This section will discuss ways in which the evaluator can apply critical judgment to the evidence that will be presented by the developer. It should also provide the developer with ample guidelines for organizing and presenting this evidence.

Recall our definition of an assurance mapping. It consists of four parts: (1) an identification of the elements in the TCB to which requirements can be mapped; (2) an identification of the elements of the DTLS, which are called requirements; (3) an actual mapping in both directions, that is a pairing of sets of elements in the TCB and the DTLS requirements they satisfy, and vice versa, along with an identification of those TCB elements which cannot be paired; (4) a convincing argument for each case that the elements of the TCB correctly do what is required by the DTLS, or contribute to satisfying some other system requirement and actually belong in the TCB.

The methods we describe address each of these components of an assurance mapping:

1. Identifying all mappable elements in the TCB.
2. Identifying all mappable elements in the DTLS.
3. Constructing and maintaining the actual mappings.
4. Making the implementation arguments.

#### 4.1.1 Baby steps vs. Giant Steps

For complex systems, the gap between the code and the DTLS may be too large to bridge with a single mapping. In this case, there is likely to be a hierarchy of increasingly complex specifications between the DTLS and the code. It may be necessary to construct a hierarchy of mappings, say code to "C-Spec", "C-Spec" to "B-Spec" and "B-Spec" to DTLS to develop an appropriately convincing argument.

Whether or not such an approach is necessary is a function of the size and complexity of the TCB, and of the abstractness of the DTLs. Because the intermediate representations require evaluation to ensure the accuracy and completeness of the final result, the hierarchical approach requires more effort. In fact, the hierarchical process consists of constructing a sequence of assurance mappings from one layer to the next, with appropriate arguments at each level for each collection in the mapping. As a result, it should only be used when attempts at a single-step mapping become too large or too complex to be convincing.

#### **4.1.2 Maintenance**

Few, if any, computer systems are ever finished. As the system evolves during use, the mapping must be kept up to date if assurance is to be preserved.

This process is not trivial. There is a tendency for developers and maintenance programmers to view documentation of any kind as someone else's job. We know of projects where large investments have been made in documenting the internals of systems only to have the investment lost when the documents were not maintained as the system evolved.

Because assurance mappings link several views of a system, it is critical to maintain them along with the code and specifications. Not only must the links between the representations be kept up to date, but the arguments that they support must be revalidated after each change.

Maintaining the assurance mappings requires effort. If this effort is not planned for, budgeted, and ultimately expended, the initial assurance will slip away, never to be recovered.

#### **4.1.3 The role of standards like 2167A and SDD**

The TCSEC deals with only the security aspects of system building and although it gives lip service to good software engineering practices, it does nothing to relate the activities that it requires to those required by other applicable software engineering standards. We advocate an integrated approach.

Although many developers view them as nothing more than a burden that causes the creation of unnecessary and unusable documentation forms, the purpose of standards such as 2167A and SDD is to impose a uniform process on development practices and to capture the results of the process in a uniform fashion.

If the process is followed, or the documentation is organized as though it had been [10], the use of these standards can make a direct and substantial contribution to the development of the mapping. On the other hand if the standards are given lip service and the required documents judged on format rather than on content, the standards will not play a substantive role in the development of the mapping and the information that they should contain will have to be obtained elsewhere, probably at considerable additional expense.

### **4.2 Decomposing the TCB**

This subsection deals with the problem of identifying those elements of the TCB, whether represented as code or data, that perform a discrete function that can be mapped to either an implicit or explicit requirement of the TCB or that must be contained within the TCB to satisfy some unstated meta requirement.

### 4.2.1 Analysis of the Code

In analyzing the code, there are a number of factors that should be kept in mind. In this section, we point out some of the key issues.

**This is a code to DTLS mapping.** The mapping process is driven by the code. Each segment of code is analyzed to determine what it does and how this helps to satisfy some DTLS requirement.

**Each component of the code does something.** The code analysis should abstract what the code is doing. If the code is well constructed, this will be simple. If the code is convoluted and complex, determining its functionality will be difficult and the mapping dubious.

**What it does ought to map to an identifiable DTLS derived specification.** This is the actual mapping process. The correspondence between code and specification should be manifest. In a cleanly implemented system, most mapping items will be one to one. Each code fragment will map to exactly one specification. It is possible that a given code fragment will map to more than one specification and that more than one fragment will map to a single specification.

**Code that maps to no spec is suspect.** Code that cannot be clearly mapped to some DTLS derived requirement is suspect. It requires careful justification and an explanation for its existence. If it represents some essential functionality not covered by the DTLS, consideration should be given to revising the DTLS and repeating the DTLS based assurance steps such as covert channel analysis and DTLS to FSPM mapping. If it cannot be justified, it should be removed.

**Specs without code to satisfy them are worse.** They represent a failure to implement some required security relevant functionality.

### 4.2.2 Graph based methods

Calling trees or graphs starting with each TCB entry point and showing the routines that can be reached from the entry point are a useful technique. Routines that appear in more than one graph need careful attention to ensure that they do not permit information flows between functions that should be isolated. In this section, we outline a process for developing the calling trees and consider some of the issues that may arise in using this approach.

**Calling tree analysis:** Under the assumption that each TCB function starts with a subroutine entry, the calling tree that starts with that entry identifies a chunk of code that is potentially executed by invoking the TCB function.

**Calling trees vs. slices:** Technically, we want the slice of code that can be executed as a result of a call on a TCB routine. This will be contained within the routines identified by the calling tree. If the code associated with the operation does not start with a call to a subroutine, we may have to explicitly identify the slice.

**Shared Subroutines:** If we have a calling tree, or the slice contains subroutine calls, we need to check to see if the same subroutines are being shared among the slices associated with more than one TCB function. If this is the case, we need to analyze for potential information flows between TCB functions. Subroutines that access or modify global variables or that retain values between calls require special attention.

**Difficult cases:** In some cases, we may need to identify and analyze separately the threads that make up a slice. This is likely to be the case when a cursory analysis seems to indicate a troublesome sharing of information between TCB routines. It may be the case that the parameters used in the individual invocations of the shared routines partition the execution paths in such a way that the unwanted sharing does not occur. Serious consideration should be given to partitioning the code into separate routines so that subsequent modifications do not inadvertently introduce sharing where none is intended.

### 4.2.3 Dealing with data structures

Active code is not the only portion of the TCB that must be mapped. The state of the TCB is represented by values contained in its data structures. These must also be described in the mapping and the relationship between the TCB data structures and the implicit or explicit state abstraction of the DTLS explained.

Implicit data structures may arise in the DTLS in a variety of ways. The DTLS may use adjectives such as “locked” as in “locked file” to talk about results that are represented by data structures or variables and their values in the TCB. An important part of the mapping process is the preparation of a list of variables, variable fields, etc. that represent the “state” of the TCB. A key characteristic of state data items is persistence. A persistent variable retains a value from one state transition to the next. Depending on the language used to implement the TCB, persistent variables may be represented in a variety of ways. One is as global variables, either scoped to be accessible within a module or accessible by any routine within the TCB. In Ada, these will typically be package-level variables. In C, they will be variables declared at the file scope or explicitly declared `static`. Other languages may use different constructs to obtain the same effect.

One subtle point in identifying persistent variables comes about because certain variables may be mapped onto particular hardware registers or memory locations, and may be affected in non-obvious ways as a result of hardware operations. Another arises when what appears to be a simple data item at the DTLS level is implemented as the result of a computation. For example, “device busy” at the DTLS level might be computed within the TCB from the current size of the device buffer — with a size of zero denoting a non-busy status.

### 4.2.4 Code and Data that do not map up

We have identified TCB code that does not map to explicit DTLS requirements. There are several possibilities:

- The code implements some functionality implicit in the DTLS, but not explicitly called for. For example, the DTLS may describe each of the TCB functions as though it were a subroutine callable by the user while the TCB implementation is actually invoked via a system call instruction, the arguments of which identify the function to be called. In this case, the code to decode the system call and dispatch to the correct routine responds to the implied requirement for a method to invoke the operations described by the DTLS but cannot be attributed to any particular function.
- Code that responds to an understood meta requirement. The DTLS describes the operations that a user may invoke in an operating system TCB but does not describe the

mechanisms that are used to decide which user process will be run at a given time. The scheduler code for the system responds to a meta requirement, but cannot be attributed to any security relevant functionality described in or implied by the DTLS.

The need for the TCB to contain functionality of this sort means that the Code to DTLS mapping will not always identify a specific DTLS requirement that is satisfied by a TCB element.

### **4.3 DTLS Decomposition**

The DTLS ought not be incomplete, but we may have to deal with non functional or distributed requirements such as:

- The TCB shall be implemented in a well structured manner.
- When invoked, a TCB operation shall be performed promptly.

#### **4.3.1 Identifying mappable specifications in the DTLS**

The DTLS can take many forms. Good DTLS representations are non-procedural. With rare exceptions, code is imperative and procedural. The the task at hand is to identify the specific sections of the DTLS to which particular segment of code responds. The approach is to decompose the DTLS to identify the specific functions that it requires the system to perform or the results that it requires the system to achieve.

Each function should be identified and listed separately. The purpose of this decomposition is to ensure that each requirement specified in the DTLS is explicitly identified. In a well constructed DTLS, developing the list will be straightforward, but even in this case it is possible for apparently simple abstract operations to give rise to multiple concrete requirements. A DTLS that is written in natural language may also contain implicit requirements that must be made explicit in the decomposition process. It is also likely to contain separable requirements conjoined in a single sentence.

The decomposition should be accompanied by a cross reference (possibly expressed in a tabular form as described below) that traces each individual requirement to an identifiable region in the DTLS. The coordinate system used to create the cross reference is immaterial, but it should take into account the fact that the DTLS may not be static. Using section, paragraph, and sentence offsets is less subject to drastic upset in the face of minor changes than is using line numbers.

The decomposition should include a placeholder for “null” or immaterial functionality that can be used to represent those portions of the DTLS (if any exist) that do not contain functions or requirements that will be implemented. If this is done, it will be possible to analyze the cross reference to show that it covers the entire DTLS. This is one part of showing the completeness of the code to DTLS mapping.

#### **4.3.2 Analysis of a DTLS**

A good DTLS is non procedural, in that it stresses the desired result rather than a way in which the result can be obtained. Performing the mapping often requires analyzing the DTLS to identify the individual requirements that are realized by the code. It may be necessary to transform

the DTLS substantially to obtain mappable requirements. This section examines some of the issues associated with the analysis of a DTLS.

**The typical DTLS is a textual document.** The DTLS is relatively informal. Even when it has a significant mathematical content, it relies on expository material for a substantial portion of its meaning.

**We must identify the individual specifications and requirements.** Extracting requirements from the DTLS may be trivial or complex, depending on the form of the DTLS and the degree of abstraction that it uses. Relatively abstract concepts such as mathematical sets and mappings may be used in the DTLS. The implementation code is required to realize equivalent functionality through an appropriate choice of data structures and algorithms. It is important to note that the code need not implement all the mathematical operations possible for the abstraction, only those that are actually used in the DTLS. This simplification is often overlooked.

**This implies a possible post processing step.** Post processing extracts from the DTLS the list of individual requirements that can be targets of mapping from code. This postprocessing can be facilitated with some care in the construction of the DTLS. Using constructs whose meaning is clear is a big help. Organizing the DTLS so that requirements are clearly and easily separable is also a winner. Ensuring that the DTLS is structured in such a way as to avoid distributing a single requirement across multiple sections is also a help.

**Software Engineering Environments can help.** An integrated Software Engineering Environment can provide a variety of support functions that will simplify extracting requirements from the DTLS. Among these are data dictionaries, cross reference facilities, etc. For a DTLS written in a pseudo code supported by the SEE, a variety of additional tools may be available.

**The result of The DTLS analysis must be a collection of identifiable specification items.** It is these items that will be the primary targets of the mapping process. They will also serve other roles in the development of the system.

**Spec items must be clear and testable.** Testable serves two purposes here. From the mapping point of view, a testable spec item is one for which inspection of the code will determine whether or not the code meets the specification. In addition, it should be possible to devise a suitable, system-level, test case to demonstrate that the system meets the spec requirement.

**Spec items may refer to operations performed by single functions or procedures or They may refer to operations performed by a thread of control.** The former are usually the easiest to map as the code to satisfy the spec is localized. A requirement that is satisfied by a thread of control or by multiple segments of code are harder to map as an argument for the collective satisfaction of the requirement needs to be presented.

## 4.4 Identifying TCB to DTLS Correspondence

This section describes the techniques to be used in the preparing the TCB and DTLS to facilitate the mapping process. It also discusses tools and techniques that will help in recording and presenting the mapping.

### 4.4.1 Hooks

The process of performing the mapping process is simplified by annotating both the code and the DTLS with cross references that facilitate the comparison of the two representations. We refer to these as hooks. Hooks may be in the form of absolute references to document sections,

to page numbers, or to specific routines in the code can cause maintenance problems as the system evolves. Or one may use symbolic hooks, relying on a tool such as a document processor to produce the appropriate entries in the final document. This is an area where mechanized support should be used if at all possible.

#### **4.4.2 Constructing hooks in the mapped code base**

A properly implemented TCB will be built to simplify the mapping process. Its structure will closely mimic the structure of the DTLS. Hooks to facilitate the mapping process can be inserted into the code as comments pointing out the structural relationship between the code and the DTLS and identifying the DTLS requirements that the coded is intended to satisfy.

When an explicit requirements list such as the one discussed above is combined with appropriate code hooks and data structure mappings, the evaluation of a code to DTLS mapping is greatly simplified. The comments should be designed to permit the evaluators to identify the particular DTLS requirement to which the code is responding. The comments should be scoped in a way that permits code that responds to no requirements to be identified. If the DTLS has been decomposed into a list of discrete requirements prior to inserting these comments, the comments should be directed to the list entries. If they are directed to the DTLS and a DTLS decomposition is subsequently produced, they should be changed to facilitate the mapping process.

#### **4.4.3 Hooks in the DTLS**

The placement of hooks in the TCB is simplified by the fact that the DTLS exists prior to the design and coding phases that produce the TCB. While it is not feasible to make explicit reference to yet to be written code in the DTLS, these references can be added, either during the coding process or during the reviews involved in preparing the mapping. The use of symbolic hooks that can be mechanically transformed into appropriate cross references is highly recommended.

#### **4.4.4 Tabular methods**

Constructing a convincing argument requires the components of the argument to be presented in some usable form. One simple form is by presenting the arguments and their support in a tabular form. These tables would provide a checklist to ensure that both the code and the DTLS are completely accounted for in the mapping process.

These tables are not an end in themselves. Rather, they are a means for focusing the evaluators attention on the arguments that provide the mapping process with the meaning that provides assurance. Analysis of these tables can identify regions of the DTLS, code, or data that have been omitted from the mapping, but simply having been included in a mapping among components is not sufficient to provide assurance. Ensuring that each component in the DTLS is represented in the TCB doesn't assure that every DTLS component is appropriately mapped to the components of the TCB to which they apply. It is the responsibility of the analyst to ensure that the meaning of the DTLS is represented in the TCB.

#### 4.4.5 Databases

Databases are useful for recording data about the mapping process and about the code being analyzed. They can support the analysis process by aiding the analyst in answering questions about the system under consideration.

There are a number of commercially available database systems that can be used to support mapping activities. They run on platforms ranging from PCs to mainframes. Using a database to maintain information about the mapping has a number of advantages. The foremost benefit is that a database approach provides a flexible way to generate reports about both the context and status of the mapping. A secondary benefit is the possibility of using the database system to answer questions about the mapping. For example, if DTLS, code, and data items are appropriately identified, it will be possible to determine whether code that responds to no discrete DTLS requirements exists.<sup>1</sup>

Developers and evaluators should be aware of developments in areas such as object oriented databases and hypertext that may simplify the mapping representation process in the future.

### 4.5 Making the Mapping Argument

In previous sections, we've described approaches and techniques for the analysis of the TCB and DTLS into appropriate elemental components, and techniques for building and documenting the associations. Having used some appropriate technique to define the collections of separable requirements from the DTLS, the appropriate components of the TCB, and a mapping among them, we must finally construct the assurance argument.

The assurance argument is the part of this process that must actually convince the evaluator; it is the most important part of the mapping. This argument must present a convincing case that, despite the differences in detail, level of abstraction, and notation, the TCB does exactly what the DTLS requires, and that it is doing nothing except what is required to meet the security-critical requirements of the system as a whole.

To be convincing, this argument needs to be sufficiently *rigorous* to allow the evaluator to feel confident that the argument is sound, and the argument needs to be sufficiently *complete* to allow the evaluator to feel confident that the argument leaves no gaps. The question of what is "sufficiently rigorous and complete" is one that must be settled on a case by case basis — what would be an appropriate level of rigor for a C2 system is quite different from what is required in an A1 system.

Thus these arguments may take many forms, depending on the degree of assurance that is required. In highly secure systems it may be appropriate to make a complete formal verification of the TCB (although this is a stronger degree of verification than is currently required of A1 systems.) Formal verification provides *very* strong assurance that the requirements of the DTLS have been satisfied by the TCB. However, a formal verification requires the specifications of the DTLS to be stated as or transformed into mathematical statements, and requires a full proof of correctness of the TCB. While this is not impossible, it requires substantial resources and careful planning early in the process.

---

<sup>1</sup>Exists may be too strong. What the database query can show is gaps in the indices, e.g., code regions that have not been mapped to some DTLS requirements.

One step back from performing a full formal verification of the TCB's meeting its requirements is to verify the TCB through *hand-proof* techniques. Where a full formal verification requires the use of various tools such as theorem provers or proof-checkers to provide assurance of the validity of the proof, hand-proof techniques use instead documented rigorous reasoning performed by a human. This is usually understood to provide less assurance than a fully-formalized machine-checked proof, but is it nothing to scoff at: mathematicians have managed to get along with only hand-proof for thousands of years.

Both full verification and hand-proof would be a very convincing way to approach the TCB to DTLS mapping. Both are stronger methods than those required for B3 evaluation. In place of these more rigorous methods, most evaluations at the B3 level will depend on mathematically-stated prose or straightforward prose arguments.

These can be used effectively, but *must be used with care*. It is entirely too easy for a prose argument to slip over into hand-waving. The evaluation of the TCB is one of the most important components of assuring trust in a trusted system. Evaluators should and must be somewhat suspicious; they should approach these mapping arguments with a critical eye.

## References

- [1] D.E. Bell and L.J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Mitre Technical Report MTR-2997, Mitre Corp., Bedford, MA, March 1976.
- [2] James W. Freeman and Richard B. Neeley. A structured approach to code correspondence analysis. In *Proc. 5th Annual COMPASS: Conference on Computer Assurance*, pages 109–116. IEEE, June 1990.
- [3] CTA Incorporated. Implementation evaluation guideline, a chapter of the handbook for the computer security evaluation of trusted systems. To be published as a NRL Technical Memorandum, October 1993.
- [4] Sue Landuer. Personal Communication, 1990. Telephone conversation with Carl Landwehr of NRL.
- [5] C. Landwehr, C. Heitmeyer, and J. McLean. A security model for military message systems. *ACM Transactions on Computer Systems*, 2(3):198–222, August 1984.
- [6] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws, with examples. Technical report, Naval Research laboratory, November 1993.
- [7] John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1), 1992.
- [8] Jonathan K. Millen. Models of multilevel computer security. Mitre Technical Report MTR-10537, The Mitre Corporation, January 1989. Also in *Advances in Computers*, Vol. 28, Academic Press.
- [9] National Computer Security Center, Ft. Meade, MD. *DoD 5200.28-STD, Trusted Computer System Evaluation Criteria*, December 1985.
- [10] David L. Parnas and Paul Clement. A rational design process: How and when to fake it. *IEEE Transactions on Software Engineering*.
- [11] Charles N. Payne, Judith N. Froscher, and Carl E. Landwehr. Toward a comprehensive INFOSEC certification methodology. In *Proceedings of the 16th National Computer Security Conference*, pages 165–172, Baltimore, MD, September 1993. NIST/NSA.
- [12] Jane Solomon. Specification-to-code correlation. In *IEEE Symposium on Security and Privacy*, 1982.
- [13] National Computer Security Center staff. Department of defense trusted computer system evaluation criteria. Department of Defense Computer Security Center, December 1985. DoD 5200.28-STD.
- [14] Daniel F. Sterne. On the buzzword ‘security policy’. In *Proc. Symposium on Research in Security and Privacy*. IEEE, June 1991.