Chapter 2
High Assurance Software Lessons and Techniques

Abstract To understand the principles needed to manage security in FPGA designs, this chapter presents lessons learned from the development of high assurance systems. These principles include risk assessment, threat models, policy enforcement, lifecycle management, assessment criteria, configuration control, and development environments.

2.1 Background

Since the early 1960s system developers have been concerned with problems caused by *unspecified functionality*. This can include errors introduced in the development process and extra features added by industrious engineers. Sometimes extra features are relatively benign. In other cases, the unspecified functionality is malicious.

Engineers tend to trust hardware more than software. Sometimes engineers assume the hardware to be trusted; however, most malware can be implemented in hardware. The objective of this chapter is to introduce some of the lessons learned about avoiding mistakes in system implementations.

2.2 Malicious Software

Malicious software is functionality intended to violate the security policy of the system. The taxonomy of malicious software and the vulnerabilities such software exploits is vast: a 2007 report from the Common Vulnerabilities and Exposures project listed 41 different system vulnerabilities ranging from weak authentication to cross-site scripting attacks [18]. The focus of this chapter is on high assurance software lessons learned, and the discussion will be limited to two types of malicious software: that which executes in unprivileged domains and that which executes in privileged domains.

T. Huffmire et al., *Handbook of FPGA Design Security*,
DOI 10.1007/978-90-481-9157-4_2,

2.2.1 Trojan Horses

School children know the story of the fall of Troy to the Greeks [37]. Despite nine years of fighting, the Greeks were unable to breach the walls of Troy and conquer the city. Odysseus devised a plan to give the Trojans a present and pretend to retreat. Taking their gift, a large wooden horse, within the city walls, the Trojans celebrated until the Greeks emerged from the horse to sack and burn the city. The Trojan Horse was the vehicle for violation of the Trojan security policy: no Greeks within the city walls. When the Trojans found the Horse, it was on the beach: it was the Trojans who dragged it into the city and then "activated" it by celebrating.

In terms of computer systems, a Trojan Horse is hidden functionality within software, where the latter provides some other desired service. So if a user downloads or otherwise installs an application or functionality of unknown provenance, a Trojan Horse may accompany it. When a Trojan Horse executes in the context of a user application, it is generally constrained by the privileges granted to that executable, which are derived from the privileges accorded the user. The adversary has no control over when the malware will execute: if the user never invokes the application that contains the Trojan Horse, then it may never execute.

Despite these constraints, in most systems a Trojan Horse can wreak havoc on confidentiality, integrity and availability objectives. Consider the situation illustrated in Fig. 2.1. Kathy has a file containing information that Sean should not access. She has set the access controls on her file so that Sean will not be able to access the information directly. Unfortunately, Kathy is executing an application that contains a Trojan Horse devised by Sean and his nefarious gang. Although the legitimate application may make legitimate use of Kathy's file, the Trojan Horse writes her information into Sean's file. She has no idea that this is occurring.

Modern Trojan Horses may not exhibit behavior as simplistic as that illustrated in Fig. 2.1. Instead they may send information to remote sites. The activities of these applications are often rather complex, and the mechanism used to transmit may not be visible to kernel-level auditing mechanisms.

As will be seen in Sect. 2.5.1.2, the activities of Trojan Horses can be confined when mandatory policies are enforced.

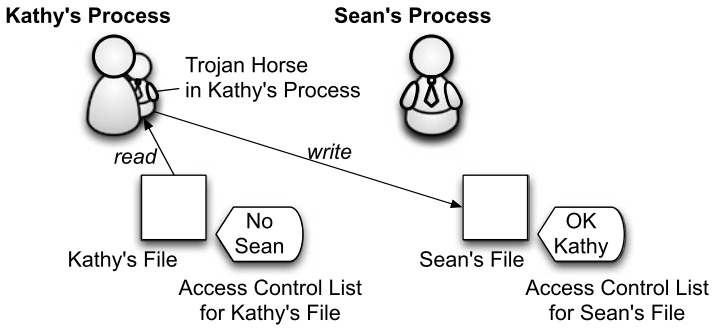

Fig. 2.1 Although Sean has no access to Kathy's file, the Trojan Horse in Kathy's process can write all of her information to Sean's file

2.2.2 Subversion

A well known example of subversion is the simple flight simulator in early versions of the Excel spreadsheet [100]. It was activated by a set of conditions that were highly unlikely to occur during typical use of the spreadsheet. It allowed the user to fly around over a gloomy landscape that featured a tombstone upon which credits scrolled, presumably containing the names of members of the development team. A less benign example of unspecified functionality is a hypothetical backdoor inserted into an operating system by its compiler [53, 101]. Anderson provides a worked example of system subversion [4]. Although no attempt is made to obfuscate the artifact, it consists of a total of eleven lines of code and allows the attacker undetected access to the entire Linux file system. Because the attacker may not know the exact nature of the ideal attack when a trapdoor is installed, it may be prudent to develop a chained subversion that consists of a toehold for subsequent system exploitation, a loader for putting the malicious payload in place, and the payload itself for the attack *du jour* [58, 72, 81].

Subversion differs from a Trojan horse in the following ways. First, a subversion can be activated by the adversary at will, whereas a Trojan horse requires the cooperation of the victim. A corollary is that the adversary can choose the time of activation, usually via triggered activation and deactivation, but, for a Trojan Horse, the time of activation depends upon the victim's use of the software. Third, a low-level subversion will bypass the security controls, and, in contrast, a Trojan horse will be constrained by the controls placed upon the victim executing it. Finally, Trojan Horses generally execute in the application domain, whereas the ideal execution domain for a subversion is the operating system.

Attackers can choose a number of system lifecycle phases, both developmental and operational, during which to target a subversion attack. The objective is to implant an artifice in the system that can execute with unlimited privileges. Myers identifies a number of lifecycle opportunities for subversion [74]. A system's lifecycle can be divided into three major stages: development, operation, and retirement. Developmental threats result in the incorrect construction of the system such that the high level policy and specifications are not faithfully reflected in the system implementation. Both unintentional errors and intentionally inserted or unspecified functionality fall into this category.

Operational threats can expose information assets to a variety of attacks. Imprecise interface specifications involving groundless assumptions can be exploited. Exploitable flaws may result from poor design and implementation. Chained attacks may allow attackers access to critical information. If the system is constructed in a way that permits its operational state to be manipulated, covert channels [60] may be possible. In the case of hardware, it may be possible to extract information using side channel attacks involving power or timing analysis, e.g. [56].

After further subdividing these phases, Myers identifies subversion threats for each phase and recommends rigorous design and development methodologies applied to people, processes and tools, accompanied by lifecycle assurance as a holistic approach to the mitigation of these threats. Table 2.1 extends this analysis to include system retirement.

Table 2.1 Lifecycle opportunities for subversion

Phase	Threat
Design	Inclusion by high-level designers of exploitable design elements
Implementation	Introduction of flaws and artifices in code base
Distribution	Additions to product and bogus updates
Installation	Untrusted installers insert artifices or misconfigure the system
Operation	Exploit flaws to install trap doors
Retirement	Analyze system and media to extract information

Subversion of an FPGA can occur at many levels ranging from developmental attacks on the hardware and software to operational attacks. Of course, the devices can be subverted at the IC level; however, given that the IC manufacturer does not know the use to which the base array will be applied, such attacks are probabilistic and lack the guarantees desired for a well designed subversion. As posited by both Karger and Schell [53] and Thompson [101], the tools used to construct the FPGA offer a vector for developmental subversion. Trimberger [102] describes some of the challenges associated with non-destructive validation of the equivalence between the original design and the implemented design. He recommends the use of layout-versus-schematic (LVS) comparison tools as a means to detect subversion. Despite the finite nature of the systems, such detection schemes are extremely challenging. Techniques that may be considered for countering subversion in FPGAs include: testing and validation of design tools, verification of design flows, and static analysis of HDL code.

2.3 Assurance

A system's *functional* security mechanisms are those that implement the access control rules, the login mechanism, the audit trail, and various security administrator functions. In contrast, *assurance* relates to the trustworthiness of the system. Just because a user might trust a system does not mean that it merits that trust: it must be trustworthy. This maps to a confidence that the system is doing what it is supposed to do and nothing extra. If the system contains errors or is implemented in a way that permits unintended use of its interfaces, then it contains unspecified functionality. In addition, a system might contain functionality that is intentionally inserted by a malicious adversary.

It is up to system owners to determine the adequate amount of assurance. The trustworthiness or assurance of systems is elevated through careful lifecycle management from the elicitation of requirements through retirement. At any step along the way, an adversary may try to enter the system to add some *special* functionality. Careful system security engineering, configuration management, trusted delivery mechanisms, testing, both internal and external reviews, and the application of formal methods contribute to system trustworthiness.

Design Tip: *Assurance Requirements*. Security is not for free. Spend your security dollars wisely. Your analysis should consider the assets that require protection and the resources available to the adversary. A higher level of trustworthiness requires greater effort to properly design, implement, test, deliver, configure, operate, and audit. While formal methods may be necessary for high assurance, they are not sufficient, in and of themselves.

2.4 Commensurate Protection

In early work with shared computer systems, when various conceptual models for computer processing (e.g., processes and scheduling) and information protection were formed, vendors presented different approaches as secure, and the computing community in ad hoc or organized [53] efforts would discover and report the incompleteness or incorrectness of the system's protection features. Today, the relationship between computer vendors and their customers remains largely the same, although many vendors may no longer assert that their systems are "fit" for security purposes [119], lest they be held liable for the product's adequacy [27]. A vendor releases a product; users discover and report some of its vulnerabilities; the vendors patch the vulnerabilities and add new features; and the vendor releases the product anew. The patches and new features, especially in combination, may add new vulnerabilities. While this *penetrate-and-patch* approach is not a satisfactory process, it avoids the up-front cost of building in security. Since vendors have found that users are willing to put up with vulnerable products, the cycle continues.

Security is expensive to build into a product, as it increases the design, documentation, configuration management, and testing efforts. While this careful approach to development (see Sect. 2.3) may reduce the *overall* cost of product maintenance, those long-term savings may not be persuasive to vendors who compete on a "first-to-market" basis. In any event, since security is not free, the question arises for data owners as to how much security is enough.

Common wisdom about the protection of any property is that one should not spend more on protection than the value of that which is protected. Another maxim is to not gamble (i.e., leave unprotected) that which you cannot afford to lose. Cost-benefit and risk analysis methods can be used to quantify the level of protection required based on the value of the information: i.e., the damage to the owner if the information is violated (compromised, corrupted, or made unavailable [61]). FEMA [50] uses the following generic formula to calculate financial risk, assuming that threat and vulnerability ratings range from 0 to 1:

$$Risk = Asset\ Value \times Threat\ Rating \times Vulnerability\ Rating$$

With respect to information protection, other risk factors may include the *proportion* of the Asset Value that will be lost if the information is violated, the value

of the information to potential attackers (which may be different than the value to the owner), and mitigations to vulnerabilities. The combination of a computing system's vulnerabilities and its mitigations to those vulnerabilities can be viewed as the inverse of the *protection* it provides with respect to defined assets and threats. Various approaches have been presented for measuring the protection provided by IT systems, including different evaluation criteria [14, 110].

Assets protected by IT systems may include people, the valuation of which can include factors such as life and liberty, which may have a subjective relationship to their monetary value. The *threats* to assets can also be difficult to quantify, as discussed next.

2.4.1 Threat Model

Looking at the FEMA risk formula, enough protection (i.e., mitigation to vulnerability) must be provided to keep risk to the assets within acceptable limits, given the perceived or defined threats. If there is no threat (e.g., if one has assets that no one else wants to attack), or the protection system has no vulnerability (assets are protected completely and continuously), there is no risk. However, a conservative assumption is that attackers' motivation and resources are proportional to the value of the information resources: highly valued information requires high assurance of protection. A *threat model* provides a more systematic evaluation of threat [57, 66, 73], including factors such as:

- The nature of the asset—whether its value derives from its confidentiality, integrity, and/or availability; and if there is a temporal quality to the value (e.g., some intellectual property or strategic military data may need to be secured for decades).
- The potentially vulnerable components or interfaces of the system through which assets can be accessed; and the attacks known to be pertinent to each type of the component or interface throughout the product life-cycle (e.g., design, development, delivery, and maintenance).
- The adversary's motivations, including monetary, competitive advantage, industrial espionage, revenge, prestige, and notoriety. This can be related to the nature of the assets.
- The adversary's capabilities—technical expertise, access to the protected system, and the availability of funding and other resources such as computer time and exploitation tools.

In addition to the primary IT access control features, potential attack surfaces include the procedural as well as the automated instantiation of various design assumptions and *supporting policies*, including: identification, authentication, audit, physical security, and the education and vetting of users. For example, the most cost effective means of attacking a system could be through the use of social engineering and bribes.

A threat model is useful for the assessment of components and products as well as for system deployment environments, which can be more specific regarding the characterization of threats, assets and adversaries. The following relates threats to components and products.

2.4.1.1 FPGA Interfaces

For FPGA components, a threat model should consider both internal and external interfaces. Cores may be connected directly or by a bus. Network interfaces may include malicious traffic. The FPGA reconfiguration interface must be considered—e.g., some FPGAs can be remotely updated in the field—as well as interfaces to shared computing resources such as system and cache memory.

2.4.1.2 FPGA Assets

General classes of information assets on FPGAs include: cryptographic keys, private information, and proprietary logic designs.

2.4.1.3 FPGA Attacks

For systems that allow untrusted modules or applications to share the processing environment, it must be assumed that they are malicious. Attack analysis must include those attacks related to system design vulnerabilities (if any) that are described in the product security evaluation and the open literature. Other potential FPGA attacks include:

- Uploading a malicious design through the FPGA reconfiguration interface. A malicious design could actually melt parts of the FPGA by causing a short-circuit [41].
- Exploiting the effects of using or contending for shared resources. For example, if one core can measure the effects caused by the use of a computing resource (e.g., delay in access to cache memory, change in temperature of the device, or change in the use of electrical power) by another core, a covert channel or side channel can result.

2.4.1.4 Other Threat Model Elements

Physical attacks, e.g., to read or destroy cryptographic keys, may require tamper protection, detection, and response techniques at the system, board, or chip level, depending on the value of the assets protected. Also, FPGA manufacturing and development tools are an attack vector in which the tools are subverted to weaken the FPGA designs they produce, as a second order effect. Chapter 3 discusses physical attacks in greater detail.

2.5 Security Policy Enforcement

Security requires the specification of a policy for a system and the translation of that policy to a system implementation that enforces the policy.

2.5.1 Types of Policies

When constructing a secure system it is essential to establish what *secure* means. Assuming that resources managed by the system must be protected, it is necessary to understand the kinds of protection that should be established. Security is always understood with respect to a policy. *Information assurance* is generally defined as a set of measures intended to protect and defend information and information systems. The five pillars of information assurance provide a way to categorize overarching objectives:

- Confidentiality—information is accessible only to those who need it and protected from unauthorized disclosure.
- Integrity—information is modified only by those with appropriate authorization and is protected from corruption.
- Availability—information is usable when needed in a reliable and consistent form.
- Authenticity—the recipient of information has knowledge of its genuine sender or source.
- Non-repudiation—irrefutable evidence of message transmission from its sender and receipt by its receiver can be provided.

The first three can be associated with system resources, whereas the last two are related to communications and are supported by well-designed protocols and the enforcement of some combination of the first three. For this reason we will focus only on the first three objectives. Sterne [99] describes an *organizational* security policy as one that may be stated in very general terms. Its translation to a system implementation results in an *automated* security policy. The automated policy is usually a subset of the overall policy, since policies related to physical and personnel security are beyond the scope of the computer system implementations.

We understand IT systems to be *secure* in the context of the policies regarding the confidentiality, integrity, and availability of the information resources. Different system security policies may combine confidentiality, integrity, and availability in different ways. For example, in a real-time control system the availability of high-priority events will be paramount, and the confidentiality of information may be secondary. A system intended to manage corporate financial records may focus on integrity and associated accountability controls. Finally, a system designed to protect state secrets will ensure that confidentiality policies are enforced. Other examples include voting systems, health record systems, and employee payroll systems. Security engineers often find a tension in the CIA triad: all three policies cannot

be perfectly enforced simultaneously! The existence of a tension between availability and both confidentiality and integrity is most evident, for example in military real-time systems that must manage both classified and unclassified information.

Before continuing with a more detailed discussion of policy enforcement, it is necessary to introduce certain terminology that has proved useful for several decades.

The underlying policy enforcement mechanism controls resources, and a subset of those resources will be exported at the mechanism interface in the form of abstract data types. These may include both active and passive entities. Those resources that can be read from or written to contain information and are called *objects*. The active entities in the system are called *subjects* and may be surrogates for typical users or for system owners. A typical example of the former might be an ordinary application process, and an example of the latter might be a service process [59].

Early pioneers in system security developed models to characterize policy enforcement mechanisms. For example, Graham and Denning [40] developed a tabular model that described the rights of each subject to objects. As depicted in Fig. 2.2, each cell in the matrix contains the access modes with which the corresponding subject is permitted to access the corresponding object.

The parameters used in these checks come from policy-relevant metadata associated with both the subjects and the objects. The nature of the metadata will be determined by the kind of policy being enforced. Any policy can fall into one of two major categories: discretionary and mandatory. So, in some systems enforcing discretionary access controls, user names or groups may be bound to subjects, and some metadata, such as a list of allowed users, may be associated with the object. In a system enforcing mandatory access control policies, the metadata may consist of sensitivity labels associated with both the subjects and objects. The most familiar types of labels are those used by the military to classify information, e.g. CONFIDENTIAL, SECRET, etc.

An interesting consequence of access control systems with interfaces that permit policy modification is that it is impossible to develop an algorithm to decide for an arbitrary protection system whether or not information will leak in an unintended manner [42]. Considerable research has explored the precise characterization of protection models that *are* decidable [2, 86, 87].

2.5.1.1 Discretionary Policies

A discretionary policy is dynamic and can be modified by unprivileged subjects during runtime, whereas a mandatory policy is *immutable* to those subjects. A non-technical example of each can be found in sentencing guidelines of the criminal justice system. For many crimes, the presiding judge is able to weigh a variety of factors associated with a particular case and can determine a punishment that *fits the crime*. Alternatively, judicial discretion might be constrained through the passage of a variety of sentencing mandates, such as *three strikes* laws. Where such constraints are in place, the sentencing judge has no choice regarding the punishment: there is no discretion.

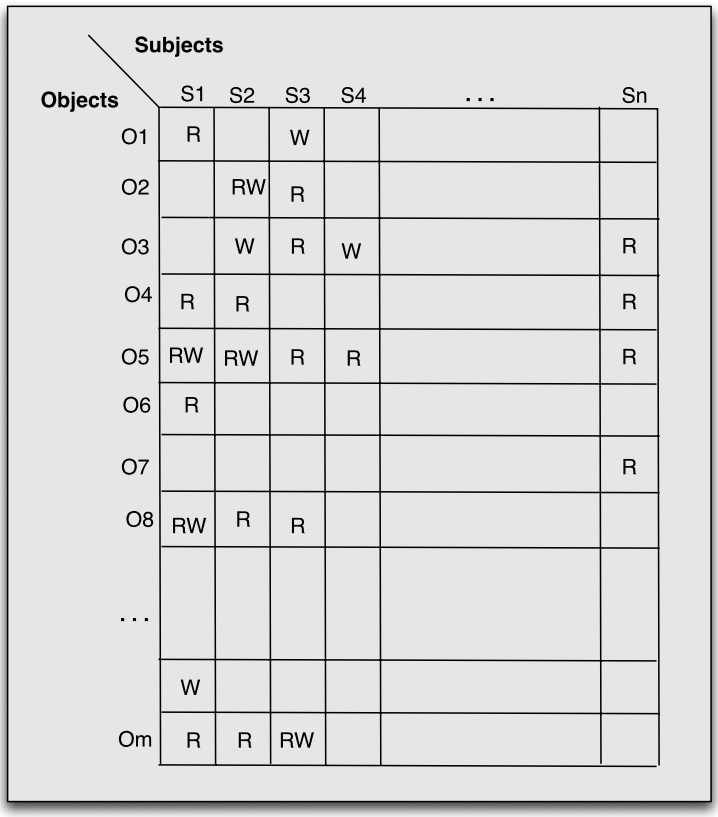

Fig. 2.2 The Graham-Denning model described access control in terms of a matrix where the rights of a subject to an object were given in the cell associated with the subject and object

In systems that enforce *discretionary policies*, an interface is provided that allows applications or users to modify the policy. Figure 2.3 illustrates how this can be a problem. Kathy's subject is executing a program that contains a Trojan Horse. As the Trojan Horse code is executed, Kathy's subject will use the runtime API to change the Access Control List (ACL) on her file to grant access to Sean. At this point, Sean's subject can read her information and store it for later use. Because the discretionary policy is *ad hoc*, it is impossible for Kathy to protect her information from access by Sean.

2.5.1.2 Mandatory Policies

In addition to being immutable, a *mandatory policy* is one that is both global and persistent: the policy is the same everywhere, and it does not change depending upon various conditions. If Jim's *secret barbeque sauce* is secret in Texas, it is also secret

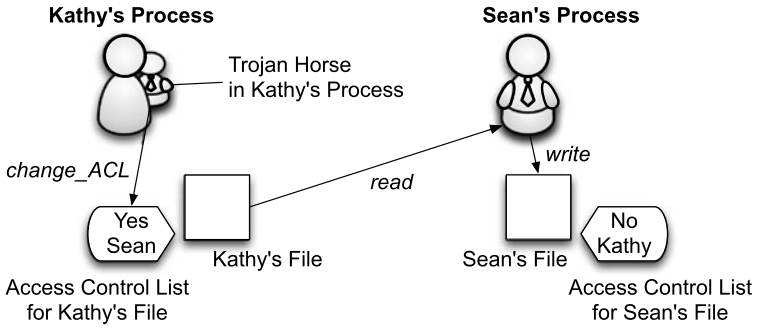

Fig. 2.3 A change to the ACL on Kathy's file permits Sean to read and store her information

in Berlin. In addition, Jim does not allow the recipe for the sauce to be available on Tuesdays from nine to ten o'clock in the morning: once the recipe is available, it is impossible to make it secret again. These global and persistent policies separate information and those who can access them into a lattice of partially ordered equivalence classes [30]. A well-understood mandatory policy is that of the military, which requires the classification of information based upon the harm its disclosure would cause to the nation. Typical military information classifications are: TOP-SECRET, SECRET, and UNCLASSIFIED. Individuals cleared for TOP-SECRET may access TOP-SECRET, SECRET, and UNCLASSIFIED; those cleared for SECRET may access SECRET and UNCLASSIFIED; and uncleared individuals may only read UNCLASSIFIED information. Because these classifications can be hierarchically ordered, this is called a multilevel security policy. However, as Denning pointed out, a partial ordering may have non-comparable equivalence classes, so sets of information may be organized in a MLS policy as well, as illustrated in Fig. 2.4. The arrows show the direction of information flow. Thus the reader must possess either a label of $\{b, c\}$ or $\{a, b, c\}$ to read information labeled b, c.

Two state machine models capture the intent of mandatory confidentiality and integrity policies respectively. The Bell and LaPadula model [7, 8] describes the secure state of a system and includes three properties: the *simple security property*, the **-property*, and the discretionary property. In this chapter, only the first two properties are of interest. If a system exhibits the simple security property, then an entity will only be able to read information at and below its confidentiality level. Another way of stating this is that it reflects typical confidentiality policies that prohibit individuals from reading information at a higher classification level than that for which they are cleared. The second property that must hold for the system is the *-property, which accidentally has this unfortunate name. It is often also referred to as the *confinement property* to reflect the notion of information confinement [60], and accounts for the challenge posed by Trojan Horses in user applications. As a result of confinement, it is impossible for an entity at a high confidentiality level to write to an information repository at a lower confidentiality level.

In the context of mandatory policies, integrity forms a dual of confidentiality. High integrity information should only be modified by high integrity entities,

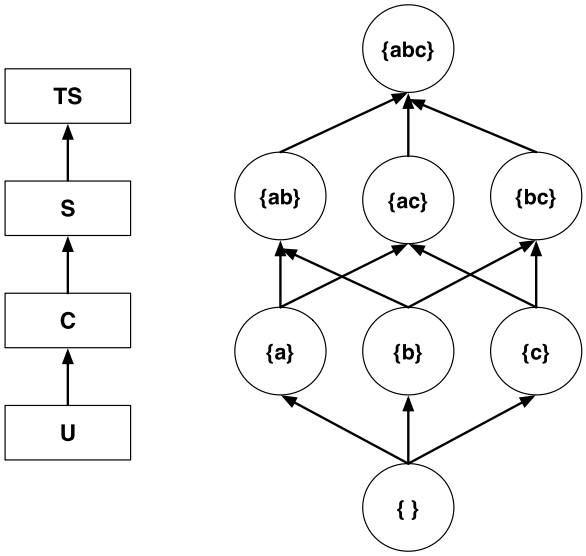

Fig. 2.4 A hierarchical ordering is shown in the lattice on *the left*, and a lattice of sets is shown on *the right*

whereas high integrity information should be accessible to entities at all integrity levels, even the lowest. Thus the Biba model [9] includes properties that constrain observation, modification and invocation.

2.5.1.3 Least Privilege and Its Policies

The *Principle of Least Privilege* is one of the cornerstones of secure system design, implementation, and management. It appeared in a codified form in the seminal paper by Saltzer and Schroeder [85] in which they described eight design principles that can guide the construction of secure systems. Their definition of least privilege stated that "every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur." Hardware mechanisms can create *protection modes* within a system that limit the privileges of applications with respect to those of the kernel (see Sect. 2.5.2.1).

Least privilege will be reflected in system design and implementation through the use of layering, modularity, and information hiding, all of which are constructive techniques that, when applied to the internal architecture of a system, improve the system's resistance to penetration. The system interface can export access control and fine-grained execution domains such that subjects may only perform authorized tasks.

2.5.2 Policy Enforcement Mechanisms

To enforce an access control policy, a mechanism must be in place that checks the access of the subjects to objects.

Since its introduction, the *Reference Monitor Concept* [3] has served as a useful abstract model for systems enforcing security policies. As an idealization of such systems, it can be used as a standard of perfection against which those designing protection mechanisms can measure their implementations.

The Reference Monitor Concept does not refer to any particular policy to be enforced by a system, nor does it address any particular implementation. Instead it articulates three properties of an ideal access mediation mechanism:

- The access mediation mechanism is always invoked: every access is mediated. If this were not the case, then it would be possible for an entity to bypass the mechanism and violate the policy.
- The access mediation mechanism is tamperproof. Thus, it is impossible for a penetrator to attack this ideal access mediation mechanism so as to disable the required access checks.
- The access mediation mechanism itself "must be small enough to be subject to analysis and tests, the completeness of which can be assured" [3]. This means that the mechanism must be understandable. It is necessary to ensure that it is doing what it is supposed to do and no more.

This articulation of a mechanism has met the test of time and continues to be an effective tool for describing the abstract requirements that drive secure system design and implementation. No viable alternative has been introduced, and it has proven effective, even under close scrutiny.

The minimal requirements for protecting the most privileged system elements from less privileged applications were described by Saltzer and Schroeder [85]. They include privilege bits, a memory management mechanism, controlled entry points to privileged functions, and a trusted way to bind user attributes to those of the active entities executing on behalf of the user. Each of these requirements will be discussed in greater detail in the next sections.

2.5.2.1 Privileged Instructions, Rings, and Gates

Systems are organized in terms of privilege. Hardware resources are managed by the most privileged software components, which organize and export abstract data types at an interface used by the next less privileged components. The simplest privilege hierarchy is that of a two-state processor that provides two privilege domains. The *privileged* domain is used by the operating system or kernel, and applications occupy the *unprivileged* domain. More elaborate hardware architectures support several hierarchical privilege domains, such as those of the Intel x86 family of processors, which has four hardware privilege levels [47].

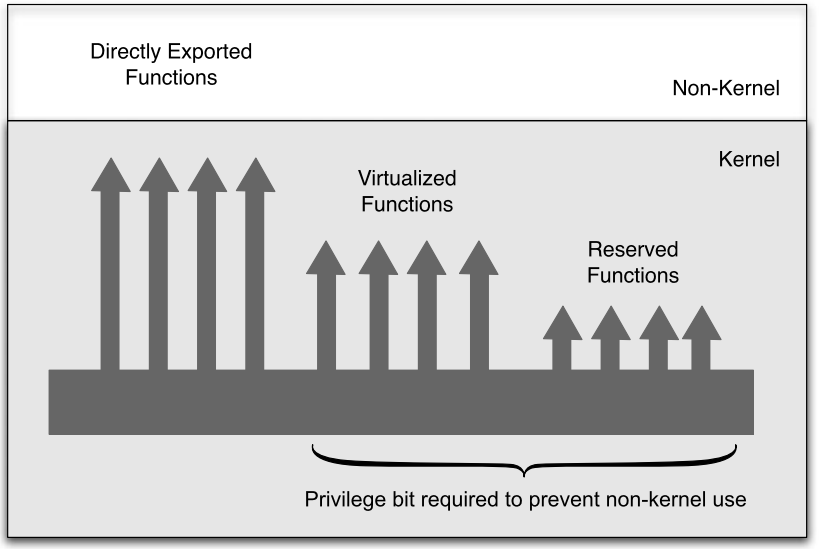

Fig. 2.5 Hardware instructions may be exported directly at the kernel interface, virtualized and exported as new abstract data types, or reserved for the exclusive use of the kernel

The ability of the processor to check the privilege level of an active entity within a task when attempts are made to access resources, transfer to a different privilege domain, or execute selected instructions, is an essential element of the overall protection mechanism provided by the hardware. *Privileged instructions* will only be executable by entities in the most privileged domain; if an application attempts to execute a privileged instruction, the hardware will issue a protection exception, and processing may be forced into an exception handler. Instructions that affect the state of the processor such as those to manage hardware memory resources, manage control and other registers, halt the processor, and perform a limited number of other critical functions will be privileged.

For performance reasons, most instructions are directly accessible by all privilege domains. Certain privileged instructions may be virtualized such that abstract data types are exported at the kernel interface. For example, the kernel may export an abstraction of the memory subsystem in the form of files, segment handles, or other objects. Finally, certain instructions will be reserved for kernel use alone, e.g. the *halt* instruction. Figure 2.5 illustrates these instruction differences.

2.5.2.2 Memory Protection, Process Address Space and Virtual Memory

To protect itself and protect processes from each other, the kernel must manage memory. Exclusive access to the memory management instructions ensures that the kernel can allocate memory for its own use and for the processes it creates. Depending upon the processor, this may involve management of segments, page tables,

or both. Obviously, the address space accessible to the kernel is that of the entire processor, whereas that of the non-kernel applications is limited. When memory is accessed, the hardware checks the privilege level associated with the memory with that of the executing entity. For the memory access to be valid, the address must be within the address space of the process and must be accessible by the privilege level making the access. A kernel handler can return an exception if the address is not part of the valid address space.

Virtual memory adds another level of complexity to address space management. Virtual memory allows processes to have the appearance of an address space larger than the physical memory resources available in hardware. The virtual memory of each process is divided into small, equal-sized pages. Secondary storage, called swap space, is used to maintain the pages while the process is executing, and pages are swapped into and out of primary memory as needed. A process may only access pages that have been allocated to it. To maximize performance, this virtual-to-physical address space mapping is managed using combinations of hardware and software support. Detailed descriptions of virtual memory can be found in many articles, texts and manuals [29, 43, 47, 95].

2.5.2.3 Object Reuse Mechanisms

One way for adversaries to obtain information is through data scavenging. Although rummaging through the garbage to find papers that might contain sensitive information is a classic form of scavenging in the real world, digital data scavenging is an attractive corollary. Thus, no matter what policy is enforced, it is necessary to ensure that resources that may be reallocated to different processes are purged of information associated with their previous usage.

Objects encompass all information containers in a system, and the general term for this aspect of secure system implementation is *object reuse*. Because objects are pervasive elements in systems and are often shared, many techniques for ensuring their reusability have been developed [76]. Examples of memories that must be purged between use by different processes include primary memory, caches, secondary storage, buffers used by I/O devices, and various registers: essentially anything that could contain residual information from its use by a different process.

Ensuring that the objects to be purged are identified and managed correctly requires a systematic methodology, such as that discussed by Wichers [116]. Consideration must be given to how information objects are implemented in terms of combinations of initialization, allocation, deallocation and protection of the resource pool from which objects are created. For example, purging can be systematically performed before each new allocation of a resource, or after each deallocation. To determine the completeness of the object reuse implementation, a careful study of the system should be conducted. It should be noted that object reuse analysis differs from covert channel analysis because objects exported by and directly accessible via the system interface are intended as containers for information that must be shared or confined according to the security policy, whereas the system metadata used to establish covert channels are not intended to be accessible information containers.

2.5.2.4 Controlled Entry Points

If applications could invoke kernel functions arbitrarily, then the resource management services provided by the kernel would be obviated. Chaos would ensue: process isolation could not be guaranteed, and the kernel's internal resources could not be protected from manipulation by applications. Any hope of security policy enforcement would evaporate. To encapsulate the kernel and ensure that the use of the function is allowed and that only intended kernel functions are invoked by non-kernel entities, the kernel must provide a mechanism so that all calls to it can be controlled.

Hardware support is needed to accomplish this task. In the simplest case, a special instruction is invoked by the application layer that causes control to transfer to a special location in the kernel. There, the kernel will examine a predefined location for the parameter list. In addition to the usual parameters, an identifier for the desired function will be provided. The kernel should validate the parameters to ensure that pointers and address ranges are associated with the domain of the application and not that of the kernel. The kernel may then transfer control to the function that will process the call.

The mechanisms just described are sufficient for systems that have only two privilege domains, but a problem arises if the system has multiple privilege domains. If all the kernel can tell is that it has been invoked from a less privileged domain, then it is impossible to determine whether intermediate privilege domains are being protected from accidental or intentional abuse by even less privileged domains. An elegant solution to this problem of controlling inward calls is to use a gate mechanism [77], e.g. *call gates* [47]. These gates are placed at the boundary of each domain and control the transfer of execution from a less privileged domain to a more privileged one. They can be set up so that all calls into more privileged domains must cascade through a series of gates or so that a call can skip intermediate domains. This allows each domain to export its functions to selected lower privilege layers in the system. For example, the kernel may export certain kernel management functions only to the next most privileged layer and not to those with lesser privilege, thus providing the capability for the system builder to provide trusted code external to the kernel for management activities. In contrast, less trusted applications would be unable to invoke the kernel management functions. Intermediate domains may export abstract data types and the gates needed to invoke the type managers at the domain interface.

2.5.2.5 User Attribute Binding: The Trusted Path

Since the attributes bound to subjects acting on behalf of users are the basis for access control decisions, it is clear that a well-defined user identification and authentication policy is essential for secure systems. It was this observation that led Saltzer and Schroeder to include a trustworthy technique for identification and authentication in their list of fundamental requirements for protection [85].

Assuming that both the user and the identification and authentication mechanism are trustworthy, how can the user be sure that security-critical information being

entered is not captured between the human interface and the I&A mechanism by a man-in-the-middle or some other malicious entity? An unforgeable connection that assures protected user communication with a trusted system mechanism is required. This is called a *trusted path*. Users invoke the trusted path using a *secure attention key*: a single key or special combination of keys or other input device intended solely for the purpose of establishing a trusted path. The secure attention key signal is received by the system's trusted mechanisms, and the I&A interface is displayed in a trustworthy manner to the user. Subsequent interaction is protected, and users have confidence that passwords and other critical authentication information is protected.

It is worth noting that a trusted path need not be restricted to the input of passwords: other critical information might require similar protection. The entry of bank account and credit card numbers, on-line confirmation of large financial transactions, electronic access to certain health records, or other high-value activities constitute examples where a trusted path provides enabling technology to organizations.

On a single platform, a trusted path requires that the interface presented to the user be constructed such that it depends only upon trustworthy mechanisms. This means that the use of large graphical user interface libraries of unknown or questionable provenance should not be within the dependency hierarchy of the trusted path mechanism. As a part of the system's overall security architecture, the trusted path must be as trustworthy as the components enforcing critical security policies.

A trusted path always refers to the interface between the user and the machine. Of course, in distributed systems, trusted communications between systems is also necessary. The term *trusted channel* is used to describe the protection of inter-system communications. In distributed systems, both trusted paths and trusted channels may require the use of cryptography. Care must be taken to ensure that the cryptographic functions and key management mechanisms are trustworthy.

User authentication to the system can be based upon any of three types of attributes: physical characteristics of the individual, something the individual knows, or something the individual has. Biometrics encompasses the use of physical characteristics for user authentication based upon physical characteristics. These include a number of modalities, common examples of which include fingerprints, voice recognition, retinal scans, iris scans, and facial recognition. Use of biometrics for access control requires that an initial biometric be enrolled. The template of the biometric of a claimant is compared to the enrolled template. Because of variations associated with biometric collection, an exact match of the two is extremely improbable, and statistical methods are used to determine whether to accept or reject a match. A number of significant research challenges, such as security, scalability, privacy, interoperability, and social aspects, need to be addressed to enable confident use of biometric technology for verifying identities [35, 49]. Passwords are something that the user knows and presents to the system to obtain access. Passwords do not suffer from the variations in collection described above, but they are vulnerable to guessing and brute-force attacks. A balance between the complexity of the password and user acceptability must be achieved. An example of an authentication mechanism involving something a user has is a physical token such as an ID card that may be presented to the system. Because tokens are susceptible to loss or

theft, their use is often coupled with a second authentication mechanism. To provide higher confidence that only valid authentications occur and to limit the complexity of the individual mechanisms used, organizations often choose to combine authentication techniques. When two different methods are used to authenticate users, this is called *dual-factor authentication*, which naturally leads to *multi-factor authentication*, where the number of attributes is further expanded.

A wide range of authentication mechanisms is available, so system developers need to consider the effectiveness of the technical mechanisms against their usability, the context in which they will be employed, their cost, and their maintainability from both a physical and technical perspective.

If a user has one password for access to all systems, then the compromise of that single password renders the information being protected in all of the systems vulnerable. To mitigate this threat, it is recommended that users have different passwords for each different account. As the number of accounts proliferates, users must memorize an increasing number of passwords. A second problem may be encountered in enterprise systems where users may have to authenticate many times to access various services during a given session. In such cases, user frustration can be addressed through the implementation of *single sign-on* mechanisms. However, single sign-on has both advantages (e.g., fewer passwords to remember and enter) and disadvantages (e.g., greater damage if credentials are compromised).

2.5.2.6 Discretionary Policy Enforcement Mechanisms

Discretionary access controls are enforced by two kinds of mechanisms: access control lists and capabilities.

Access control lists (ACLs) itemize the access permissions of subjects to objects, such as files, directories, or devices. Each ACL entry consists of the name representing an entity, such as an individual user or group, and the rights accorded to that entity. Groups are convenient because access control lists for a large number of similar users (for example, all students enrolled in Psychology 101) can be simplified, thus reducing the possibility of administrator error. The largest possible group is everyone, which in many systems is termed *public*. Extremely simple ACLs are found in UNIX [5] and its descendant systems, such as Linux, where only a short set of *permission bits* are used to determine access to a file: owner, group, and public. For decades, systems intended for commercial use have had more sophisticated ACLs in which the permissions of particular individuals may be defined.

Not only can ACLs be used to permit access, but they may support the ability to deny access to particular subjects. Consider a file that provides answers to the exam to students following an exam. If Andy was out of town and must take the exam this week, the professor can explicitly deny Andy access to the answers until after he has completed the test. Thus the ACL may contain read permission for the group consisting of all members of the class and an additional entry that denys Andy

access. The astute reader may have noticed that Andy as a member of the group *class* has been given access to the answers, so rules must be established regarding the precedence of the ACL permissions. In this case, the intent of the instructor is met if the ACL entries associated with individuals take precedence over the permissions accorded groups. Issues to be considered when determining ACL precedence rules are discussed by Lunt [68].

The simplest permissions found in ACLs may be merely read, write, and execute. Thus various users and groups will have one or more of these access rights to the object. Because discretionary access controls are often implemented for sophisticated applications, other types of access permission may be created. For example, it may be useful to allow certain users only append access to a file, for example a log file. In this case, writes are restricted to the end of the file. To implement append access, the underlying protection system will use a combination of both read and write. Without the user's knowledge, the system will open the log file, set the write pointer to the end of the file, and then write the next log record to the file. It is possible that the system or application programming interface will allow neither read nor write access explicitly to the user.

If ACLs contain the permissions to the objects, how are the permissions to the ACLs managed? This question is important because the runtime interface that permits modification of ACLs is what distinguishes them as elements of a discretionary access control mechanism. It is possible to associate *control* access rights with ACLs. The users or groups with control access to the object may be designated and will determine who can grant or deny permission for other access rights. These control access rights may be highly granular; for example, a particular individual might be given the ability to control a particular access right, e.g. read, within the ACL. Furthermore, the concept of control can be extended upward one more level so that certain individuals have control-of-control access rights. This rich set of access rights allows organizations to tailor discretionary access controls to meet specific requirements.

When new objects are created, it is important to ensure that the initial value of each ACL reflects the intended security policy. In some systems a template may be used to associate a default ACL with each new object. Such defaults may be system-wide or may be determined with higher granularity, for example, in the case of files, on a per-directory basis. Lunt [68] provided an analysis of discretionary control defaults, which can range from no access (i.e., minimized access) to complete access. In the context of least privilege, a default of limited or no access is a wise choice.

ACLs are attractive because all permissions associated with a particular object are localized. This allows policies to be managed easily. It is worth noting, however, that policy changes are not effective immediately. ACLs are used to check access permissions once the object is *opened*, prior to the actual read or write. The results of the access check are cached, and as long as the subject keeps the object *open*, the rights obtained at that *first* access are retained. Thus, revocation of access is not immediate and will only be effective the next time the user attempts to *first* access the object unless more sophisticated mechanisms are in place.

2.5.2.7 Capability Systems

Capabilities provide another way to implement discretionary access controls. In capability-based systems, which were first described by Dennis and van Horn [32], the list of access rights to objects are associated with the subjects, rather than the objects. In addition to defining access rights, capabilities provide a way to name objects, thus providing the basis for capability-based addressing [36]. For example, when a user logs onto the system, an initial set of capabilities is bound to the subject executing on the user's behalf. As execution progresses, subjects may accumulate additional capabilities. When a subject attempts to access an object, the RVM checks the access rights in the capability, and permission is granted if the requested access is included in those rights. Thus, once a subject possesses a capability for a particular object, that object may be accessed with the rights specified in the capability; all the subject needs to do is present the capability. A detailed discussion of capability systems is provided by Levy [65]. A notable implementation of a highly granular capability mechanism in an operating system was found in the CAP system [117].

Because a capability-based system distributes the access rights to each object among the subjects and since the rights may be stored as initialization data for each subject, revocation presents challenges. In addition, if subjects are able to copy and store capabilities, the revocation problem is further exacerbated. Also, there is no central location that can be inspected to determine which subjects have potential access to a particular object. Instead, the capability list for each subject must be inspected. If one decided to revoke access to an object, potentially every capability list in the system would require inspection to ensure that the revocation was complete. Again, as with ACLs, revocation would not take effect if the subject were already actively accessing the object. An approach to solve the revocation problem was proposed by Redell [80]. Capabilities can be particularly troubling in systems where mandatory policies are to be enforced because, in typical capability systems, no distinction is made between the access right and the ability to grant that access right [10]. The extension of capability systems to support lattice-based security policies was explored by Karger and Herbert [51, 52].

Although capability systems can be implemented, they are notoriously complex, and their lack of a conceptually simple policy-enforcement mechanism caused this approach to be eclipsed in terms of high assurance approaches. However, capability systems continue to be of interest, e.g. [92, 118].

2.5.2.8 Mandatory Security Policy Enforcement Mechanisms

Separation of domains requires the isolation of subsystems as well as mechanisms to allow controlled sharing between these domains.

2.5.2.9 Types of Mandatory Mechanisms

A *security kernel* binds internal sensitivity labels to exported resources and mediates access by subjects to other resources according to a partial ordering of the

labels defined in an internal policy module [88]. The label space may support confidentiality and integrity policies as well as non-hierarchical categories [69]. A security kernel usually provides a hardware-supported ring abstraction [91, 93] and can host trusted subjects [89]. The rings can separate processes within a privilege level. Thus, a subject is a process-ring pair. All high assurance security kernels to date have utilized segmented memory, which provides persistent hardware based process-local memory-protection attributes [33, 38, 89, 90] as opposed to dynamic, global, hardware attributes based on memory paging mechanisms.

The security kernel mediates external communication via network devices that are each dedicated to a given sensitivity level, or via multilevel devices, in which a sensitivity label is bound to each network protocol data entity (e.g., datagram). Security kernels generally support full resource and resource-allocation configurability during runtime.

A *separation kernel* [83], which is sometimes referred to as a partitioning kernel [67], maps its set of exported resources onto partitions:

$$resource\_map : resource \rightarrow partition$$

Multiple *subject* resources and *object* resources may be mapped to a given partition, but a partition is an abstraction and is not itself a subject. Resources in a given partition are treated equivalently with respect to the inter-partition flow policy, and subjects in one partition can be allowed to access resources in another partition. Separation kernels enforce the separation of partitions and allow subjects in those partitions to cause flows, each of which, when projected to partition space (per the *resource_map* function), comprises a flow between partitions (which may be between different or identical partitions). The allowed inter-partition flows can be modeled as a *partition flow matrix* whose entries indicate the mode of the flow, similar to that of Fig. 2.2, discussed earlier

$$partition\_flow : partition \times partition \rightarrow mode$$

The mode indicates the direction of the flow, so that

$$partition\_flow(P_1, P_2) = W$$

means that subjects in P_1 are allowed to write to any resource in P_2. The assignment of resources to partitions and the access control or *flow* rules are passed to the separation kernel in the form of configuration data that the kernel interprets during system initialization. Since configuration data correctness is critical for the enforcement of the intended security policy, a configuration tool is often described for the construction of flow rules. Although not part of the kernel itself, this tool can help the security administrator or system integrator to organize and visualize complex data. This helps to ensure that user inputs reflect the intended policy.

Least privilege separation kernels (LPSKs) provide two important enforcement benefits beyond basic partitioning kernels. First, LPSKs increase the granularity of privileges accorded subjects as described in the Separation Kernel Protection Profile (SKPP) [46]. Second, unlike a partitioning kernel, an LPSK extends reference monitor features so that it is the locus of control for all inter-partition flows. In addition

to the *resource_map* and *partition_flow* functions of a partitioning kernel, an LPSK supports the principle of least privilege in a manner than can be represented as a *subject-resource* flow matrix,

$$subj\_res\_flow : subject \times resource \rightarrow mode$$

It is possible to allow the subject-resource flow matrix to override the rules of the partition flow matrix [46]; however, a more restrictive interpretation, where a given flow is allowed by the LPSK only if both matrices allow it, is more intuitive and ultimately more likely to be correctly configured in system implementations [63]:

$$\begin{aligned} &allow\_flow(subject, resource, mode) \\ &\quad \rightarrow mode \in subj\_res\_flow(subject, resource) \quad \& \\ &\quad mode \in partition\_flow(subject.partition,\ resource.partition) \end{aligned}$$

The SKPP requires that

1. each secure configuration include an identification of a *base* partial ordering of flows between partitions to identify the strict MLS policy, and
2. subjects allowed to cause flows between partitions in addition to those base flows are treated as trusted subjects.

Figure 2.6 shows how the granularity of an MLS security policy can be refined through the application of the two policies. The baseline partial ordering appears in (a) and illustrates the partial ordering of the partitions: information may flow, as shown by the heavy arrows, from P1 to P2 and from P2 to P3, where the subjects within a partition are the entities that cause the flow. Least privilege is illustrated in (b). In this context, only certain subjects may cause flows, designated by lightweight arrows, to and from particular resources. For example, $S_{2,2}$ can only read from both O_{21} and O_{22}. A trusted subject, perhaps a specialized tool that downgrades only certain information, is shown in (c). It is permitted to cause a flow from O_{32} to O_{12}. As is the case for all trusted subjects, it is trusted to honor the intent of the system security policy and is thus shaded and has a dashed arrow to indicate a flow in opposition to those articulated in the base policy. Depending on the policy, an explicit partition rule allowing flow from P3 to P1 may be required for the S_{32} to O_{12} flow to be allowed.

A review of the relative merits of these three approaches is provided by Levin et al. [64].

2.5.2.10 Audit Mechanisms

A record of security-relevant events can be provided by an audit mechanism. If it includes dynamic rule-checking, the audit mechanism may provide alerts of impending security violations. Policies must be established to determine what should be audited. For example, audit might include: only accesses to a particular object; all activity on the system; the activities of subjects at a particular sensitivity level;

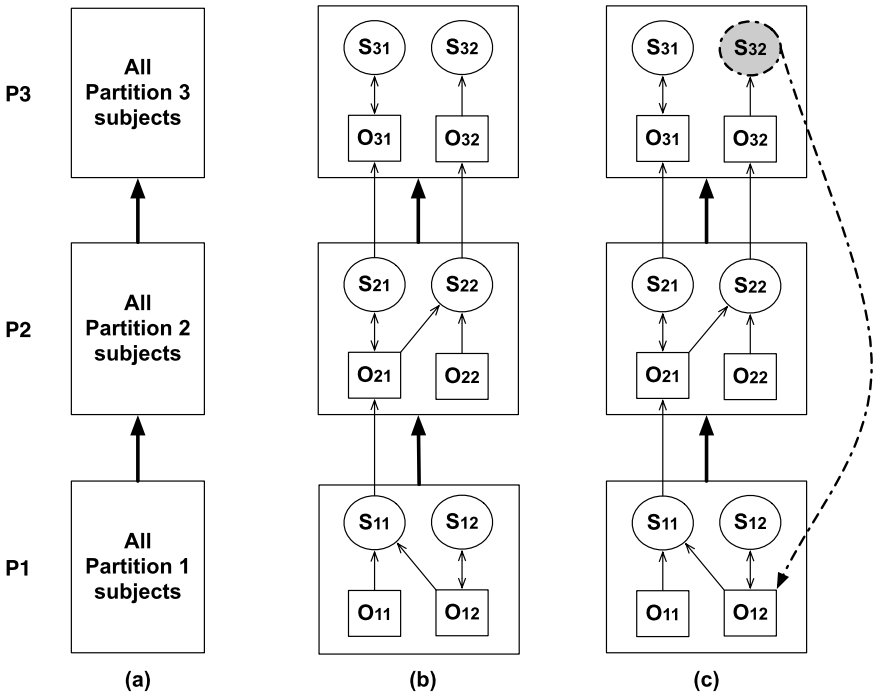

Fig. 2.6 SKPP policies. The partition_flow policy is shown in (**a**), the more granular subject_resource policy in (**b**), and (**c**) illustrates a trusted subject

the use of selected system calls; etc. Because the security administrator and other trusted individuals engage in security-critical activities, an audit of their activities should be maintained. Also, good audit reduction tools are needed, otherwise voluminous audit records are not likely to be particularly useful.

Intrusion detection systems (IDS) constitute a dynamic form of auditing. Suggested by Anderson in 1980 [19], the next work on intrusion detection systems, published in 1987, provides a general IDS model [31]. Since that time, a wide variety of systems have been developed for network intrusion detection, e.g. Snort [96] and Bro [79], as well as host-based intrusion detection, e.g. Tripwire [55]. In the Snort and Bro systems, network traffic is closely monitored for patterns that would indicate the prelude to or initial steps of an intrusion, whereas the latter systems introspectively observe the activities on a single platform in an attempt to catch malicious activity prior to the completion of an attack. A fundamental limitation associated with intrusion detection systems is that they detect only what they are encoded to look for. Thus, however artful the encoding, the adversary can find a way around it. Security administrators are often presented with a choice between the reduction of false positives and the reduction of false negatives. IDSs can be thought of in terms of the following dichotomies: host-based vs. network based;

after-the-fact vs. real-time vs. predictive; and modeling misbehavior/detecting similarities vs. modeling good behavior/detecting deviations.

2.5.3 Composition of Trusted Components

To reduce costs in the construction of large systems, it is desirable to use existing commercial components as much as possible. In the parlance of FPGAs, this translates to the reuse of exiting IP. Composing secure embedded systems from multiple components presents several challenges.

2.5.3.1 Composition Problems

A classic example of problems introduced by composition is illustrated by the *cascade problem* [71]. The problem can be described as follows. Consider an MLS system where labels are linearly ordered by a comparison operator ($\geq$), and two labels are *adjoining* if there is not a label between them in the ordering. If a component enforces the security policy sufficiently to keep separate the information in two adjoining sensitivity levels, S_i and S_j, but no more, the component is said to have a level of trustworthiness of T_{two}. Let there be two T_{two} components, C_1 and C_2. Suppose that the organizational security policy requires the separation of three adjoining sensitivity levels: S_1, S_2 and S_3. The trustworthiness required for this separation is T_{three}. If C_1 separates S_1 and S_2 and C_2 separates S_2 and S_3, the architecture can be considered sufficiently trustworthy. However, if the components C_1 and C_2 are subsequently connected at the S_2 level as shown in Fig. 2.7, their combination forms a system—a virtual component—that spans three levels and yet has only T_{two} trustworthiness. Trustworthiness is not additive, so two serially linked T_{two} components are insufficient for a network policy that requires the separation of three sensitivity levels with a T_{three} level of trustworthiness.

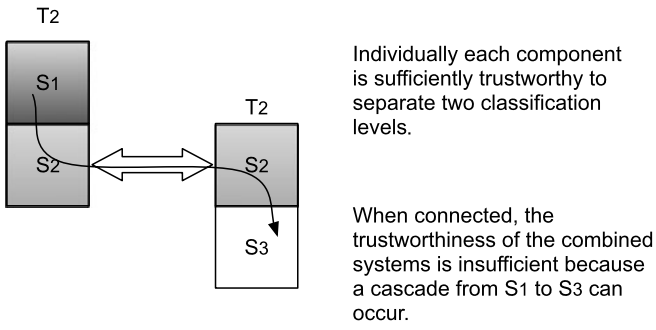

Fig. 2.7 The cascade problem. Separately the components are sufficiently trustworthy, yet when combined, their level of trustworthiness is insufficient

Analysis of the cascade problem shows that the algorithmic identification of a cascade within a network involves a time complexity of $O(an^3)$ and space complexity of $O(an^2)$, where a is the number of security levels and n is the number of nodes in the network [45]. Furthermore, the cost of calculating a correction (i.e., a policy-preserving reorganization of the network) is NP-Complete [45]. While reorganizing a network may only be a one-time cost, the conclusion to be drawn from this analysis is that it is better to avoid cascades in the first place.

An approach to avoiding the ad hoc nature of the composition problem is to provide a framework of rules under which pre-analyzed conjunctions of components may safely occur. For example, under TCB subsets [94], the system security policy is decomposed into a set of monitors, each of which enforces a subset of the overall policy. For example, one monitor might enforce the mandatory confidentiality policy, another the mandatory integrity policy, and yet another the discretionary confidentiality policy. A subject's access to objects is granted only when access is permitted by all three of the monitors. If the system can be subdivided such that separate components contain the monitors, then through appropriate engineering of a strict set of design and interface requirements, it may be possible to construct an architecture of these subset components that will generally satisfy the overall system policy. The goal, here, is to make it possible to construct the TCB subsets such that they may be evaluated independently, yet their composition results in enforcement of the larger system policy [75]. The result has been called a *partitioned TCB*.

> *Design Tip*: *Composition*. Trustworthiness is not additive and may in certain circumstances be degenerative. Two components that are individually trustworthy are not necessarily trustworthy when put together. The TCB subset abstraction involves decomposing the security policy into a set of enforcement mechanisms, each of which enforces a subset of the overall policy. All subpolicies must be in agreement for access to be granted. However, the general result of *evaluation by parts* is still a hard problem because unintended behavior can *seep out of the box*.

2.6 Assurance of Policy Enforcement

Software and configurable hardware have many similarities as one examines assurance and lifecycle management practices employed for each class of technologies over a product's lifecycle.

An FPGA contains a set of logic elements that perform a specific function, and the programmable nature of an FPGA requires a means to specify the logic that defines the FPGA's behavior. Just as in software, which is defined in terms of a program expressed in a programming language, logic elements of the FPGA are typically expressed in a hardware description language. Furthermore, common FPGA

logic elements may be expressed in libraries that are combined to produce progressively more complex functions. In this regard, an FPGA image may be considered a persistent and statically loaded program. Such a program can and should be subject to all the same analysis and assurance practices that are routinely applied to software.

Not unlike the offensive line on a football team, assurance is an often overlooked, but vital, element of maintaining a product through its entire lifecycle. Just as the line consistently performs the complex and unglamorous dirty work that allows the quarterback, running backs and receivers to be lauded for advancing the team down the field, the consistent, rigorous and successful application of sound assurance practices that result in a successful development effort is rarely recognized. Conversely, just as the offensive line often receives attention only when the quarterback gets sacked, assurance and configuration management practices typically receive the most scrutiny when flaws are uncovered.

But those knowledgeable about the sport know that the offense's success starts with the ability of the line to consistently open holes in the defense and provide the pass protection that allows the team to advance. The same is true of the assurance and configuration management processes that must be applied throughout the lifecycle of a robust and sound product.

2.6.1 Life Cycle Support

Life cycle management is an indispensable set of development and maintenance disciplines that helps define the assurance of a product, and thus the core competency of the manufacturer. Well-defined and efficiently managed life cycle models are the cornerstone to achieving design security for large and complex software projects. The actual life cycle processes vary among different organizations and depend on the product type (hardware versus software) and desired level of protection (high assurance versus low assurance). Nevertheless, these processes should apply the cradle-to-grave security principles during the entire life cycle of a product, viz. requirements engineering, design, development, manufacturing, testing, distribution, remediation and end-of-life disposal [108]. Although it is not a traditional focus of life cycle management and is often neglected, requirements engineering is an important aspect of security. Both functional and assurance (non-functional) security requirements must be correctly defined to avoid providing the wrong functionality or protecting the right functionality wrongly.

2.6.1.1 Assessment Criteria

The Common Criteria (CC), an internationally-recognized security evaluation framework, emphasizes the fundamental aspect of requirements engineering and prescribes a security requirements derivation methodology that is centered on a

thorough analysis of both real and perceived threats to be mitigated by the end product [23]. Life cycle support plays an important role in the CC paradigm as evidenced by the large number of requirements devoted to life cycle modeling, configuration management (CM), secure delivery, developmental security, and flaw remediation [25]. Life cycle security issues related to programmable integrated circuits (e.g., FPGAs) are further addressed in a CC supplementary document [22] which provides guidance on how to apply the base CC evaluation methodology to hardware IC products that must be evaluated under the CC. In the US, all national security systems are mandated to be evaluated in accordance with the CC or NIST Federal Information Processing Standard (FIPS) validation program by the overarching National Security Telecommunications and Information Systems Security Policy No. 11 [20]. These systems often include programmable circuitry.

In the case of commercially available hardware cryptographic modules to be used in sensitive but unclassified environments, FIPS Publication 140-2 [104] is presently the official evaluation criteria which levies similar life cycle (albeit somewhat mislabeled as design assurance) requirements, i.e., configuration management, secure delivery and installation, developmental evidence, and operational guidance. FIPS 140-2 defines four hierarchical levels of security and explicitly refers to the CC for security requirements levied on trusted software used in the target crypto modules. This tie to the CC has been removed in the current draft FIPS 140-3 [107] which has been in a public review phase since July 2007. In this draft, design assurance was renamed to life cycle assurance, which includes additional requirements on the use of an automated CM system, vendor testing, and more rigorous development processes, e.g., the use of a high-level HDL for custom ICs starting at Security Level 2 [107].

2.6.1.2 Use of Trustworthy Tools

The draft FIPS 140-3 also requires that if software is included in the crypto module then information about the compilers, configuration settings, and methods used to generate the executable code must be provided, even at the lowest Security Level 1. This relates to the vexing *trustworthy tools* problem, i.e., how users can ascertain the correctness of the tools used to create executable code or to fabricate hardware circuits. For FPGAs, the problem is exacerbated due to the complexity of the tools used to design, manufacture, assemble, test, and distribute FPGA products. These tools are typically made by different vendors (both foreign and domestic), and there are no standardized metrics or criteria to assess the integrity of their implementation. In theory, formally verifying every tool would provide a high level of confidence that the end product is not subverted by the tools, but in practice, doing so would be prohibitively expensive. These challenges and other issues related to the trustworthiness of integrated circuits (both ASIC and FPGA) have been investigated and documented in the Defense Science Board study on High-Performance Microchip Supply [113]. This report had prompted DARPA to issue the Trust in Integrated Circuits research solicitation in 2007 [114], which focuses on "developing technologies that offer rigorous validation of IC hardware and its design regardless of

where the design or manufacturing processes take place." The article entitled *The Hunt for the Kill Switch* [1] highlights the Trojan Horse attack and summarizes myriad vulnerabilities that are inherent in today's highly sophisticated hardware.

For secure software development, a *best practices* approach includes the following steps: (1) carefully selecting the tools based on common empirical analyses of quality factors such as provenance, maturity, stability and wide use, (2) performing a thorough black-box testing and security analysis of the tools' functional interfaces, and (3) maintaining the tools under strict configuration control. This process, if implemented properly, can help mitigate the threats of malicious subversion and accidental misuse. The testing and analysis results provide evidence to support the assertion that the selected tools do not introduce malicious functionality. Chapter 8 discusses as future work the application of this idea to the FPGA design, using a similar process to pick and manage the tools used in the different stages of the FPGA design flow (e.g., logic synthesis, place & route, etc.).

2.6.1.3 Applying Security Principles to Life Cycle Process

An effective life cycle methodology should incorporate the following high assurance software security principles as part of a defense-in-depth strategy:

- Audit
- Least privilege
- Separation of duties

Audit is a discipline of continuous inspection and assessment for accountability purposes. To detect security violations and deter penetration attempts, an audit framework should include both automated technical measures and manual actions. When applied to life cycle management, audit can ensure that all design and manufacturing activities conform to the life cycle control policies and procedures which can subsequently help offset the impacts of security breaches committed by malicious insiders. Deterrence is an effective risk management mechanism, and employing an audit policy that requires both random and periodic audit actions in all phases of a system's life cycle can also discourage potential adversaries from launching attacks. Configuration management (discussed below) is one form of auditing. It addresses the control of developmental and operational configuration changes that could affect the assurance disposition of a system.

Adherence to security principles such as least privilege and separation of duties that were suggested by Saltzer and Schroeder in their seminal work on protection of information [85] also affords additional protection and damage control. They defined least privilege as a design restriction that can limit the damages caused by both programmatic and operational errors, and separation of privileges (i.e., duties) as a protection mechanism that can reduce the risk of being compromised by colluding and maligned entities. The FPGA design and manufacturing flow is a complex series of different activities involving many actors and interdependencies, and verifying that the implementation of various components (e.g., netlist) has not deviated

from the intended design (e.g., HDL design files) is a daunting task. While the use of cryptographic techniques can protect some parts of this flow [102], procedural safeguards based on the principles of least privilege and separation (i.e., isolating critical steps, enforcing distinct roles, and restricting privileges to the task at hand) can help strengthen the assurance posture of the end product.

2.6.2 Configuration Management

Technology is constantly evolving, and changes are unavoidable. Configuration management (CM) is a well-established practice to control changes that should be assimilated early in the life cycle for both software and hardware products. CM retrofitting (viz. adding CM as an after-thought) is costly and can severely impact a product's integrity since maintaining a complete and unmodified change history for traceability purposes is the bedrock of CM. CM can be viewed as an active defense mechanism that, when implemented properly, can help to mitigate inherent security risks associated with evolutionary changes.

Most developers are aware of CM, mostly as a versioning control mechanism, but they often choose to either ignore or marginalize the importance of CM for fear of being burdened by the CM controls and procedures. It is exactly those rigorous safeguards that, if properly practiced, could enable the establishment of the initial baselines of hardware and software components and the subsequent change control of those components. Change control plays a critical role in CM as its main objective is to prevent unauthorized modifications (including accidental errors) to the baseline configuration items. For high assurance software development, a thorough security analysis of the proposed changes (e.g., modifications of existing components and additions of new components) to assess the security impact on other parts of the system must be performed and reviewed prior to the approval of the change request. A system of checks and balances must be employed to deter collusion, e.g., clear separation of CM and the development environment, and to ensure the validity of the security analysis, e.g., to ensure that the analysis is performed by a trained security analyst and that the review is done by a Change Control Board. For FPGA development, the same CM objectives and requirements apply, especially for complex FPGA implementations that contain a large amount of code developed and maintained by a multitude of principals (core designer, system developer, manufacturer, etc.).

Configuration change control is important but is not enough. NIST has defined a set of CM requirements (i.e., security controls) that addresses a wide range of concerns ranging from establishing CM policy and procedures to maintaining a current inventory of the components used in a system [106]. Different combinations of these requirements are levied on information systems based on the system's potential impacts on an organization in the event of a security compromise. FIPS Publication 199 defines three levels of potential impacts—low, moderate, and high—based on the severity of the effect on the operations and assets of the organization, i.e., limited

for low impact, serious for moderate impact, and severe or catastrophic for high impact [105]. NIST Special Publication 800-53 recommends the following eight CM requirement categories for moderate-impact and high-impact systems [106]:

- Configuration Management Policy and Procedures
- Baseline Configuration
- Configuration Change Control
- Monitoring Configuration Changes
- Access Restrictions for Change
- Configuration Settings
- Least Functionality
- Information System Component Inventory

These requirements cover all life cycle phases, i.e., planning, development and deployment, and they apply to all components of a system, including FPGAs. In other words, besides its effect on the developmental assurance of individual products, CM also contributes to a system's mission assurance if its use is included in the system's security strategy and plans. The security posture of a system is partially based on a set of approved configuration settings that, if changed without proper analysis and traceability, would invalidate the system's accreditation, i.e., license to operate. This is also true for FPGA-based embedded systems. Changes to a bitstream file in the field may have detrimental effects on both performance and security of a system; thus, CM policy and processes should be established and enforced to minimize the risks associated with bitstream reconfiguration.

2.6.3 Independent Assessment

Accountability and transparency are central elements in building and sustaining trust. Juvenal's poignant observation about trust[1] has been used over the years to emphasize the need for having external oversight to provide greater accountability in governance. It is easy to draw a parallel between this need for transparency and the need for security evaluation in secure product development since the security posture of a product could be strengthened if the product underwent an independent security assessment.

To be credible, the security evaluation of a product should be performed by an objective third party, preferably a government-sanctioned organization. This is because impartiality and independence are essential to guard against bias and collusion, respectively. In general, user confidence will increase if a vendor could demonstrate that their product passed the scrutiny of official organizations with legal oversight responsibility. For example, doctors and patients in the US would feel safer if the prescribed medicines for a life-threatening condition were approved by the Food and Drug Administration. Similarly, security-minded IT users in the US would be more

[1] "Quis custodiet ipsos custodies?" ("Who guards the guardians?")—Juvenal, Satires VI.347.

inclined to use security products that have been validated by official evaluation authorities such as the National Information Assurance Partnership (NIAP) Common Criteria Evaluation and Validation Scheme (CCEVS) and NIST.

CCEVS oversees and validates the evaluation of security products by CCEVS-approved commercial testing laboratories that are accredited by NIST [12]. The CC testing labs evaluate security products in accordance with the CC and NIAP-recognized Protection Profiles [13]. The CC is an international standard, and there are different evaluation schemes in other countries that provide the same CC evaluation oversight as CCEVS. In the US, the evaluation of cryptographic modules is performed by NIST, not CCEVS. NIST oversees the Cryptographic Module Validation Program (jointly with the Communications Security Establishment of the Government of Canada) which validates cryptographic modules in accordance with FIPS cryptographic standards, e.g., FIPS 140-2 [103].

Product evaluation authorities such as CCEVS and NIST only assess the trustworthiness of individual products (e.g., operating system, firewall, web server), not the trustworthiness of the end systems that use evaluated products. From a system acquisition viewpoint, independent security evaluation of individual products is a critical part of the technical due diligence which, when properly exercised, can help mitigate risks throughout the system's life cycle. However, a system that is composed of different evaluated products is not necessarily secure since the interactions among security functions provided by the evaluated products may result in new vulnerabilities. Product evaluation is performed based on security assumptions (e.g., physical and personnel security) and threats of specific operating environments for which a product is intended to be used. When integrated into an end system with a different threat model, the evaluated protection mechanisms may be inadequate to mitigate the threats manifested at the system level.

An independent critical examination of the integrated protection mechanisms at different dimensions of implementation (e.g., hardware, operating system, application software) could help identify adverse emergent behaviors prior to deploying the composed system for operational use. In the federal government, the process that federal agencies use for security and risk assessment before authorizing a system for operation is known as certification and accreditation (C&A). When there is a change in the functionality of an authorized system or its operational environment, subsequent C&A activities might ensue, depending on the organizational C&A policy, to determine and mitigate risks resulting from the change.

Although the complexity of the FPGA design in a product is typically hidden (encapsulated) in higher level functional components (e.g., processor cores and device controllers), it is important to not overlook the malleability of FPGAs in the security assessment of the overall system. The use of FPGAs should be inspected with the same depth and rigor that are used to assess the security of critical software in a product. Dynamic reconfiguration is an inherent benefit of using FPGAs, but it is also a double-edged sword. When FPGA-based products are used in a mission-critical system, it would be prudent to include, as part the system's C&A process, system-level architectural and design analyses to look for unintended side effects caused by poor, incorrect, or unanticipated use of FPGA-based components.

2.6.4 Dynamic Program Analysis

Dynamic program analysis generally refers to the testing and analysis of a program under execution. A target program is subjected to a specifically constructed set of input data, and instrumentation is used to examine and validate the program behavior. Input data may be constructed and instrumentation applied to:

- test for functional behavior
- test for performance
- test for timing constraints
- test for resource usage

Functional testing may be considered a common form of dynamic program analysis in which a program is subjected to a set of input data designed to exercise every interface of a program and validate all outputs in terms of effects, errors and exceptions. Functional testing is often conducted in conjunction with code coverage analysis to ensure that all program code gets exercised by the input data. The input data set is driven by code coverage and specifically designed to stimulate specific responses from the program.

In many environments, thorough testing of functional interfaces may be considered sufficient because properties such as performance, timing constraints or resource usage are not particularly demanding or may not be specified at all. Towards the other end of the spectrum are embedded, real-time systems in which resource and/or timing constraints may be absolute, severe and critical to the proper operation of the program. It is often difficult to know a priori how a program will behave in terms of performance or resource usage. In these environments, dynamic program analysis is applied to ensure the program behaves properly in response to a range of real world and pathological conditions.

2.6.4.1 Testing

Testing occurs in various contexts and at different phases during the development and certification of a software product. Unit and integration testing is typically conducted during development by the developers themselves. The rigor applied by developer-implemented testing can vary widely across organizations.

Once an overall software product is developed, it is subjected to a system test, typically conducted by a distinct Quality Assurance group. The test requirements are derived from a specification that completely describes the interface to the product. Testing procedures at this level are generally recognized to include:

- Thoroughly and completely exercising all interfaces to the program.
- Validating behavior under all externally visible states and conditions.
- Testing for both successful and unsuccessful conditions, including generation of all errors and exceptions.

If a product is subject to certification, then an evaluator may conduct additional testing. The examination and testing of a product can vary widely depending on the nature of the certification [26, 109]. Such testing may be comprised of simply running a defined test suite against the product to verify compliant functional behavior. However, often the certification also seeks to assess the compliance of a product at not just a single point it time, but over the lifetime of the product. Under this type of requirement, the certification process must examine not only the product itself but also all the software development practices applied to develop and maintain the product. Often the evaluator cannot examine, much less repeat, all the testing conducted by the developer. Instead, the evaluator examines testing processes, test records, documentation, and other materials that demonstrate software lifecycle assurance.

While testing is recognized as a vital process in software development, it is often compartmentalized to discrete phases within the development lifecycle, typically after a software unit or even a complete program has been coded.

The quality of a software product can be greatly improved by addressing testing requirements throughout the development lifecycle, not only by conducting testing at appropriate points within the development process, but also by actively considering testing impacts during the design and implementation of a program. A *design for test* strategy includes both application of design principles that promote simple and appropriately constructed interfaces and application of coding techniques that facilitate testing.

Application of design principles, including developing a sound abstraction and creating an appropriately modular architecture, tend to promote more intuitive and simpler programs that are thus easier to test. Interestingly, while principles of abstraction and modularization might not be easily grasped, more rote examination of an implementation can yield equally valuable feedback. For example, an interface regarded as *hard to test* or having *too many test cases* suggests an interface that might be unnecessarily complex given an understanding of a particular functional requirement.

Application of techniques that facilitate testing can enable development of extensive unit and integration test suites that may be used to support initial development and regression testing. Unit tests are typically conducted using a test driver that can exercise the unit under test by not only invoking the interfaces exposed by the software unit but also by applying code practices that allow one to selectively expose and control the internal state of the software unit.

Similar to the traditional software development process, the FPGA development process typically involves several iterative steps in which the output of each progressive step of development is fed back to verify the functional behavior of the circuit. To support this iterative model, the FPGA development tools have evolved to support sophisticated hardware description languages and testing techniques that can be used to construct test fixtures to exercise implementation logic and interfaces of an HDL circuit description at each stage of the FPGA design flow.

2.6.5 Trusted Distribution

It is important that the delivery of trusted products is protected against counterfeiting and subversion during transit from the vendor site to the user site. The user must have the following guarantees on the received product:

- It is of the correct version as specified by the vendor. If the product had been evaluated, the distributed version must also match the evaluated configuration.
- It comes from the vendor, not from a fraudulent source, and
- It arrives unmodified.

This type of assurance requirements is characterized in evaluation criteria as *trusted distribution* [110] and *secure delivery* [25], respectively.[2] The need for these requirements stems from the fact that any unauthorized changes to a product's security mechanisms during its life cycle could have an adverse effect on the system's ability to enforce its security policy. While configuration management provides protection against subversion during the development phase, trusted distribution addresses threats of subversion and forgery during the distribution phase.

In TCSEC, trusted distribution requirements are only levied on Class A1 products since it was considered too costly for lower assurance classes to provide assurance measures for ensuring secure delivery [111]. Specifically, the TCSEC requires the vendor to implement a distribution system that can ensure the integrity of the delivered product and to provide procedures for users to validate that the received version is the same as the distribution master's version [110]. These requirements apply to both the initial delivery and subsequent updates of a product. Accompanying the TCSEC is a series of technical guidelines whose purpose is to clarify the TCSEC requirements and to provide implementation guidance. *A Guide to Understanding Trusted Distribution in Trusted Systems* [111] is one such document. This guide explains why trusted distribution is an important life cycle assurance measure and provides insights on different approaches to implementing an effective trusted distribution mechanism.

The Common Criteria, on the other hand, imposes trusted distribution requirements starting at Evaluation Assurance Level 2 (EAL2), the second lowest level of a seven-level assurance scale. In previous CC versions (Version 2.3 and older), trusted distribution requirements were grouped into one family (ADO_DEL) and expressed in terms of *delivery procedures* (EAL2 and EAL3), *detection of modification* (EAL4 through EAL6), and *prevention of modification* (EAL7) [21]. These categories are linearly hierarchical, i.e., detection of modification requires delivery procedures, and prevention of modification requires both detection of modification and delivery procedures. These requirements are similar to the TCSEC requirements in that they focus on the use of the vendor's master copy and address both procedures and technical measures employed at both ends of the delivery channel.

In the current CC Version 3.1, trusted distribution requirements are expressed in terms of *delivery procedures* and *preparative procedures* [25]. The former re-

[2] For the purpose of this discussion, the two terms are considered to be equivalent.

quires the vendor to document and use the delivery procedures for distributing the product. The latter requires the vendor to provide acceptance procedures for users at the user site. The CC assurance requirements underwent a major rewrite after Version 2.3, and trusted distribution requirements are now defined as two separate families (ALC_DEL and AGD_PRE), making it harder for CC novices to follow. Anti-subversion and anti-counterfeit safeguards that were explicitly specified in the previous trusted distribution requirements have been made into application notes which are not normative in the CC paradigm.

Although the trusted distribution requirements in the TCSEC and CC are different in scope and form, they all have the same objective of protecting the product against post-development subversion and theft. Similar threats also exist in FPGAs, and just as in high assurance software, stringent delivery mechanisms should be employed to mitigate these threats at different development stages of an FPGA-based system.

2.6.6 Trusted Recovery

A secure system (implementing a state-machine model) must ensure that each state transition after an initial *secure state* results in another secure state [8]. Although the definition of secure state depends on a system's security policy model, in general a secure state can be viewed as a system state in which the system data is consistent and uncorrupted, and the system can correctly enforce the security policy represented by the its security model [46].

When a system detects that it is no longer in a secure state, it must attempt to self-recover to a secure state without further protection compromise while recovery is in progress. The concept of recovering in the presence of abnormity while ensuring continuity of protection is known as *trusted recovery*. The TCSEC and CC further characterize these exceptions as either *failure* or *discontinuity of operations*. A failure can be an error condition in the system's *security functionality*[3] that causes the system to behave incorrectly (e.g., inconsistent values in system data structures caused by transient hardware failure) or a media failure (e.g., a disk crash). A discontinuity of operation, on the other hand, is an error caused by inappropriate human actions, e.g., inappropriate shutdown of a system [24, 112].

While a system is running (as opposed to halted), it can be in one of two modes: operational or maintenance. Trusted recovery mechanisms must be supported in both modes [46]. These mechanisms must be able to determine whether the current system state is secure or not and to initiate mode-specific recovery actions to repair the system if the system is not in a secure state. Certain error conditions can be recovered by automated mechanisms (e.g., remapping of a bad disk sector) while

[3]The term *security functionality* is based on the term *TOE Security Functionality (TSF)* which is defined in the CC as a set consisting of all hardware, software, and firmware of the TOE that must be relied upon for the correct enforcement of the security functional requirements [23].

others require manual recovery actions (e.g., system crash due to an unexpected error). Recovery methods also vary depending on the operational environment of the system. For example, the recovery method used in an embedded real-time system would require more complex processing than the method used in a traditional computer due to resource constraints of the embedded system (e.g., processor overhead and response time) [62].

Although self-testing is a system integrity requirement, it is also relevant to trusted recovery. Its use during system initialization and normal operation can detect abnormal conditions that require recovery. Moreover, self-tests can be invoked by an automated recovery mechanism as part of the recovery process or performed by an administrator to verify that the system is indeed in a secure state after completing a recovery action.

Regarding life cycle assurance, recovery mechanisms must meet the same development assurance requirements levied on other security-relevant functions, since they are parts of the system's security functionality. Their design and implementation must be critically reviewed to ensure that architectural properties such as self-protection, least privilege, modularity and minimization are upheld, and that no Trojan horses or trap doors exist in the code. Furthermore, security testing and vulnerability analysis must be performed to determine potential vulnerabilities that could be exploited to bypass security enforcement during recovery [26].

Recovering from certain failures may require complex administrative actions. Since administrative users typically have more privileges than regular users, the principle of least privilege should be applied to the assignment of recovery privileges such that only authorized administrative users (e.g., the security administrator, not the system operator) can perform recovery functions. The operational guidance documentation must describe all types of failure conditions, recovery procedures, and tools, and for each type of failure, specific guidance on how best to recover from the failure. It is paramount that user documentation on recovery is complete and correct; otherwise, misuse of recovery functions may affect the system's ability to fail securely, resulting in compromised protection.

2.6.7 Static Analysis of Program Specifications

As opposed to testing the properties of a program[4] in execution, *static analysis* provides assurance of the properties of a program based on an objective examination of some specification of the program.

Programs can be specified at different levels of abstraction, for example, user manuals, design specs, source code, and executable code are different abstractions of a program, and different forms of static analysis examine different levels of abstraction. Often, the term *static analysis* is used to refer specifically to the analysis of source code [17], but in this context, it is used more broadly.

[4]Where *program* could be a module, component, monolithic system, or distributed system.

2.6.7.1 Code Reviews and Bug Checking

A basic form of static analysis occurs during *code reviews*, in which program authors along with peers, designers, managers, and customers, etc., look at source code. A code review can focus on properties such as elegance, faithfulness to a higher level specification, and coding errors like buffer overflows. Automated static analysis tools [6, 15, 16, 28] can perform some of these analyses of specifications, as long as the properties in question can be defined in terms of an *effective procedure* (e.g., a rule) understandable by the tool. For example, while a tool might be able to search for a well-defined buffer overflow, many concepts of program elegance are subjective and beyond the scope of current tools. Some source code properties that are statically checked by automated tools include: correct syntax and format, memory leakage, improper stack or general-memory access, memory leaks, overuse of privilege, buffer overflow, unused or duplicate functions, unused variables, uninitialized variables, lack of encapsulation/data-hiding, and time-of-check to time-of-use errors.

Automated code reviews and bug checking, like automated testing, may not provide adequate assurance that a property is completely or correctly implemented in a given program (the theoretical difficulty of writing a program to understand other programs is related to the *halting problem* [11]). The use of formal methods, discussed next, can lead to greater assurance of program security.

2.6.7.2 Formal Methods

Human languages lend themselves to ambiguity and lack of precision, whereas mathematics provides a basis for clear and precise description and reasoning about those descriptions. *Formal methods* is not, itself, a formally-defined or standardized term of art in computer science, but in general, it refers to the application of mathematics in various aspects of the software and hardware system development process. In particular, the use of formal methods to verify *security* properties is required for *high assurance* or *high robustness* [14] product ratings.

Some formal methods are:

- general mathematical models of
  - computation and processing [44]
  - security [8, 9]
- specification languages [84, 97, 98] with precise semantics that can express:
  - system behavior
  - security properties
  - refinement relationships between specifications
  - theorems of conformance to properties
  - theorems of conformance to more abstract specifications
- executable security languages with precise semantics:
  - in which security properties such as correct MLS flow can be described with first order language constructs

  - successful compilation of a program guarantees that it conforms to the properties that it includes [30, 115]
- automated systems for manipulating the logic of formal specifications, such as:
  - automatic or interactive theorem provers [54, 78, 84]
  - model checkers, model executers, and SAT solvers [48]
  - tools that automatically generate theorems based on properties in a formal specification [34]
- information flow analysis tools [34]

2.6.7.3 Refinement and Preservation of Properties

Formal verification of a secure system includes formalization of key specifications, at different levels of abstraction, as well as a series of correspondence demonstrations showing that each specification preserves the security properties of the next most abstract level—resulting in a transitive argument that the implementation preserves the security policy (see Fig. 2.8). The more the elements of this chain are formalized, the more formal the resulting argument. The particularization of a given specification to one that is more concrete (i.e., less abstract) is called *refinement*; whereas the translation of a concrete specification to one that is more general is called *abstraction*.

The prevalent criteria for high assurance verification [14, 110] have required several items in common: a formal specification of both the security policy model and the top level functional specification (an interface specification that includes the inputs, outputs, processing, and internal effects of each interface); a proof that the formal model is consistent with its own security properties; and a proof that the formal specification preserves the properties of the model. Formal methods may also be used in the analysis of covert channels and in the demonstration that the source code is consistent with the formal top level specification.

The usual expectation is that the natural language security policy and the security policy model are simple enough that their consistency can be ensured through inspection, with a high degree of confidence. The security policy model is often a

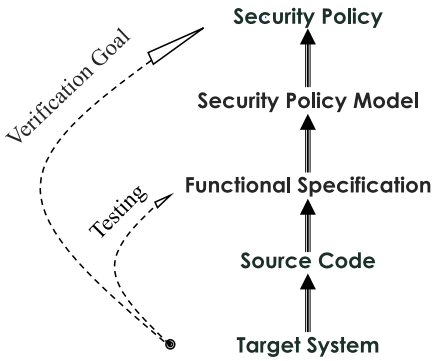

Fig. 2.8 Formal verification chain of evidence

refinement of the security policy, where an organizational-level policy that is "independent of the use of a computer" [110] is interpreted in the computer technology domain—i.e., the security policy model helps to transition between the security policy and the formal specifications. On the other hand, verified translation of source code to machine code (i.e., *trusted compilers*) and the automatic translation of formal functional specifications to source or machine code [97] are topics of current research.

In what has been called the *refinement paradox*, [70, 82] it has been shown that the refinement of an *information flow* model [39] does not, in general, preserve the security properties of the model—e.g., the addition of detail (viz., in the formal specification) may introduce flows not included in the abstract formal model. In this case, covert channel analysis can be performed to ensure that the information flow in the refined specification is correct. Similarly, if an access control model [8] is used, a covert channel analysis of the formal specification ensures that information flow that is extraneous to the model does not violate the security policy.

The correct correspondence of source code to the formal specification can be demonstrated through exhaustive enumeration of the source code, in which each element of code is mapped to its representation in the formal specification and is accompanied by a rationale as to why the semantics of the formal specification are preserved in the refinement. One of the goals of research to provide verified automatic translation of the formal specification to source code is to avoid the arduous manual code-correspondence task as well as to reduce the error rates associated with manual coding.

References

1. S. Adee, The hunt for the kill switch. IEEE Spectrum **45**(5), 34–39 (2008)
2. P. Ammann, R.S. Sandhu, The extended schematic protection model. J. Comput. Secur. **1**(3, 4), 335–385 (1992)
3. J.P. Anderson, Computer security technology planning study. Tech. Rep. ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972. Also available as vol. I, DITCAD-758206. Vol. II, DITCAD-772806
4. E.A. Anderson, C.E. Irvine, R.R. Schell, Subversion as a threat in information warfare. J. Inf. Warfare **3**(2), 52–65 (2004)
5. M.J. Bach, *The Design of the UNIX Operating System* (Prentice Hall, Inc., Englewood Cliffs, 1986)
6. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K.R. Jamani, A. Ustuner, Thorough static analysis of device drivers. SIGOPS Oper. Syst. Rev. **40**(4), 73–85 (2006)
7. D.E. Bell, L. LaPadula, Secure computer system: unified exposition and multics interpretation. Tech. Rep. ESD-TR-75-306, MITRE Corp., Hanscom AFB, MA, 1975
8. D.E. Bell, L. LaPadula, Secure computer systems: mathematical foundations and model. Tech. Rep. M74-244, MITRE Corp., Bedford, MA, 1973
9. K.J. Biba, Integrity considerations for secure computer systems. Tech. Rep. ESD-TR-76-372, MITRE Corp., 1977
10. E.W. Bobert, On the inability of an unmodified capability machine to enforce the *-property, in *Proceedings DoD/NBS Computer Security Conference*, September 1984, pp. 291–293

11. G. Boolos, R. Jeffrey, *Computability and Logic* (Cambridge University Press, Cambridge, 1974)
12. CCEVS, Publication #4: guidance to CCEVS approved Common Criteria testing laboratories, version 2.0. National Information Assurance Partnership Common Criteria Evaluation and Validation Scheme, September 2008
13. CCEVS, Publication #1: organization, management and concept of operations, version 2.0. National Information Assurance Partnership Common Criteria Evaluation and Validation Scheme, September 2008
14. CCMB, Common Criteria for information technology security evaluation, revision 3.1, revision 1, no. CCMB-2006-09-001. Common Criteria Maintenance Board, September 2006
15. B.E. Chelf, S.A. Hallem, A.C. Chou, Systems and methods for performing static analysis on source code. US Patent 7,340,726, Coverity, Inc., 2008
16. H. Chen, D. Wagner, MOPS: an infrastructure for examining security properties of software, in *Proc. 9th ACM Conf. Computer and Communications Security (CCS 02)*
17. B. Chess, G. McGraw, Static analysis for security. IEEE Secur. Priv. **2**, 76–79 (2004)
18. S. Christy, R.A. Martin, Vulnerability type distributions in CVE. http://cve.mitre.org/docs/vuln-trends/index.html, May 2007
19. J.P.A. Co, Computer security threat monitoring and surveillance. Tech. Rep., James P. Anderson Co., Fort Washington, PA 19034, February 1980
20. Committee on National Security Systems, NSTISSP no. 11, revised fact sheet. National Information Assurance Acquisition Policy, July 2003
21. Common Criteria Maintenance Board, Common Criteria for information technology security evaluation, part 3: security assurance components, version 2.3, CCMB-2005-08-003. Common Criteria Maintenance Board, August 2005
22. Common Criteria Development Board, The application of CC to integrated circuits, version 2.0, revision 1, CCDB-2006-04-003. Supporting document, mandatory technical document. Common Criteria Development Board, April 2006
23. Common Criteria Maintenance Board, Common Criteria for information technology security evaluation, part 1: introduction and general model, version 3.1, revision 1, CCMB-2006-09-001. Common Criteria Maintenance Board, September 2006
24. Common Criteria Maintenance Board, Common Criteria for information technology security evaluation, part 2: security functional components, version 3.1, revision 2, CCMB-2007-09-002. Common Criteria Maintenance Board, September 2007
25. Common Criteria Maintenance Board, Common Criteria for information technology security evaluation, part 3: security assurance components, version 3.1, revision 2, CCMB-2007-09-003. Common Criteria Maintenance Board, September 2007
26. Common Criteria Maintenance Board, Common Criteria for information technology security evaluation, evaluation methodology, version 3.1, revision 2, CCMB-2007-09-004. Common Criteria Maintenance Board, September 2007
27. M.A. Cusumano, Who is liable for bugs and security flaws in software? Commun. ACM **47**, 25–27 (2004)
28. M. Das, S. Lerner, M. Seigle, ESP: path-sensitive program verification in polynomial time, in PLDI 02: Programming Language Design and Implementation, June 2002, pp. 57–68
29. P.J. Denning, Virtual memory. ACM Comput. Surv. **2**(3), 153–189 (1970)
30. D.E. Denning, A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976)
31. D.E. Denning, An intrusion-detection model. IEEE Trans. Softw. Eng. **13**, 222–232 (1987)
32. J.B. Dennis, E.C.V. Horn, Programming semantics for multiprogrammed computations. Commun. ACM **9**(3), 143–155 (1966)
33. DigitalNet Government Solutions, Security target version 1.7 for XTS-6.0.E, March 2004
34. P. Eggert, D. Cooper, S. Eckmann, J. Gingerich, S. Holtsberg, N. Kelem, R. Martin, FDM user guide. No. TM-8486/000/04, Reston, VA: Unisys Corporation, June 1992
35. European Commission, Biometrics at the frontiers: assessing the impact on society. Tech. Rep., European Commission Joint Research Center (DG JRC), Institute for Prospective Technological Studies, 2005

36. R. Fabry, Capability-based addressing. Commun. ACM **17**, 403–412 (1974)
37. R. Fitzgerald, trans. *Homer: The Odyssey* (Vintage, New York, 1961)
38. L.J. Fraim, Scomp: a solution to the multilevel security problem. Computer **16**, 26–34 (1983)
39. J. Goguen, J. Meseguer, Security policies and security models, in *Proc. of 1982 IEEE Symposium on Security and Privacy*, Oakland, CA (IEEE Comput. Soc., Los Alamitos, 1982), pp. 11–20
40. G.S. Graham, P.J. Denning, Protection—principles and practice, in *Proceedings of the Spring Joint Computer Conference*, May 1972, pp. 417–429
41. I. Hadzic, S. Udani, J. Smith. FPGA viruses, in *Proceedings of the Ninth International Workshop on Field-Programmable Logic and Applications (FPL'99)*, Glasgow, UK, August 1999
42. M. Harrison, W. Ruzzo, J. Ullman, Protection in operating systems. Commun. ACM **19**(8), 461–471 (1976)
43. J.L. Hennessy, D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th edn. (Morgan Kaufmann, San Mateo, 2006)
44. C.A.R. Hoare, Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
45. J. Horton, R. Harland, E. Ashby, R.H. Cooper, W.F. Hyslop, B. Nickerson, W.M. Stewart, O. Ward, The cascade vulnerability problem, in *Proceedings IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1993, pp. 110–116
46. IAD (Information Assurance Directorate), US Government protection profile for separation kernels in environments requiring high robustness. National Information Assurance Partnership, version 1.03 edn., 29 June 2007
47. Intel, Intel 64 and IA32 architectures software developer's manual, vol. 3A: system programming guide, part 1. Intel Corporation, Denver, CO, 253668-022us edn., November 2006
48. D. Jackson, *Software Abstractions: Logic, Language, and Analysis* (MIT Press, Cambridge, 2006)
49. A.K. Jain, S. Pankanti, S. Prabhakar, L. Hong, A. Ross, J.L. Wayman, Biometrics: a grand challenge, in *Proceedings of the 17th International Conference on Pattern Recognition*, August 2004, pp. 935–942
50. M.J. Kaminskas, *Risk Assessment/Risk Management*. Building Design for Homeland Security, vol. 5. FEMA, Risk Management Series ed. (2007). http://www.fema.gov/library/viewRecord.do?id=1939
51. P.A. Karger, Improving security performance for capability systems. Ph.D. thesis, University of Cambridge, Cambridge, England, 1988
52. P. Karger, A.J. Herbert, An augmented capability architecture to support lattice security and traceability of access, in *Proceedings 1984 IEEE Symposium on Security and Privacy*, Oakland, CA (IEEE Comput. Soc., Los Alamitos, 1984), pp. 2–12
53. P.A. Karger, R.R. Schell, Multics security evaluation: vulnerability analysis. Tech. Rep. ESD-TR-74-193, vol. II, HQ Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, MA 01731, June 1974
54. M. Kaufmann, J. Moore, An industrial strength theorem prover for a logic based on common Lisp. IEEE Trans. Softw. Eng. **23**(4), 203–213 (1997)
55. G.H. Kim, E.H. Spafford, The design and implementation of Tripwire: a file system integrity checker, in *Proceedings of the 2nd ACM Conference on Computing and Communications Security (CCS)*, Fairfax, VA, November 1994
56. P. Kocher, Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems, in *Proceedings of the 16th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, August 1996
57. M. Kurdziel, J. Fitton, Baseline requirements for government and military encryption algorithms, in *MILCOM*, vol. 2, Oct. 2002, pp. 1491–1497
58. L. Lack, Using the bootstrap concept to build an adaptable and compact subversion artifice. Master's thesis, Naval Postgraduate School, Monterey, CA, June 2003
59. B.W. Lampson, Protection, in *Proc. 5th Princeton Conf. on Information Sciences and Systems*, Princeton, NJ, 1971
60. B.W. Lampson, A note on the confinement problem. Commun. ACM **16**(10), 613–615 (1973)

61. C.E. Landwehr, Formal models for computer security. ACM Comput. Surv. **13**(3), 247–278 (1981)
62. K. Lee, L. Sha, Process resurrection: a fast recovery mechanism for real-time embedded systems, in *Proceedings of 11th IEEE Real Time and Embedded Technology and Applications Symposium 2005* (RTAS 2005), March 2005, pp. 292–301
63. T.E. Levin, C.E. Irvine, T.D. Nguyen, Least privilege in separation kernels, in *E-business and Telecommunication Networks; Third International Conference*, ed. by J. Filipe, M.S. Obaidat. ICETE 2006, Set'ubal, Portugal, 7–10 August 2006. Communications in Computer and Information Science, vol. 9 (Springer, Berlin, 2008)
64. T.E. Levin, C.E. Irvine, C. Weissman, T.D. Nguyen, Analysis of three multilevel security architectures, in *Proceedings 1st Computer Security Architecture Workshop*, Fairfax, VA, November 2007, pp. 37–46
65. H.M. Levy, *Capability-based Computer Systems* (Digital Press, Bedford, 1984)
66. S. Lipner, The trustworthy computing security development lifecycle, in *Proceedings 20th Annual Computer Security Applications Conference* (IEEE Comput. Soc., Los Alamitos, 2004), pp. 2–13
67. Lockheed-Martin/The Open Group, Protection Profile for PKS in environments requiring high robustness. Draft Version 1.3, submittal for NSA approval, 09 June 2003. http://www.csds.uidaho.edu/pp/PKPP1_3.pdf. Last accessed: 15 March 2009
68. T.F. Lunt, Access control policies: some unanswered questions. Comput. Secur. **8**, 43–54 (1989)
69. T.F. Lunt, P.G. Neumann, D.E. Denning, R.R. Schell, M. Heckman, W.R. Shockley, Secure distributed data views security policy and interpretation for DMBS for a Class A1 DBMS. Tech. Rep. RADC-TR-89-313, vol. I, Rome Air Development Center, Griffiss, Air Force Base, NY, December 1989
70. J. McLean, Security models and information flow, in *Proceedings of the IEEE Symposium on Security and Privacy* (IEEE Comput. Soc., Los Alamitos, 1990), pp. 180–189
71. J. Millen, The cascading problem for interconnected networks, in *Fourth Aerospace Computer Security Applications Conference*, 1988, pp. 269–273
72. J. Murray, An exfiltration subversion demonstration. Master's thesis, Naval Postgraduate School, Monterey, CA, June 2003
73. S. Myagmar, A. Lee, W. Yurcik, Threat modeling as a basis for security requirements, in *Proc. Symp. Requirements Engineering for Information Security (SREIS 05)*, 2005
74. P. Myers, Subversion: the neglected aspect of computer security. M.S. thesis, Naval Postgraduate School, Monterey, CA, 1980
75. National Computer Security Center, Trusted network interpretation of the trusted computer system evaluation criteria, NCSC-TG-005, July 1987
76. National Computer Security Center, A guide to understanding object reuse in trusted systems. Tech. Rep. NCSC TG-018, National Computer Security Center, Fort George G. Meade, MD, 1991
77. E.I. Organick, *The Multics System: An Examination of Its Structure* (MIT Press, Cambridge, 1972)
78. L.C. Paulson, *Isabelle: A Generic Theorem Prover*. LNCS, vol. 828 (Springer, Berlin, 1994)
79. V. Paxon, Bro: a system for detecting network intruders in real-time. Comput. Netw. **31**(23–24), 2435–2463 (1999)
80. D. Redell, R. Fabry, Selective Revocation of Capabilities, *International Workshop on Protection in Operating Systems*, IRIA, 1974
81. D. Rogers, A framework for dynamic subversion. Master's thesis, Naval Postgraduate School, Monterey, CA, June 2003
82. A. Roscoe, CSP and determinism in security modelling, in *Proceedings of the IEEE Symposium on Security and Privacy* (IEEE Comput. Soc., Los Alamitos, 1995), pp. 114–127
83. J. Rushby, Design and verification of secure systems. *ACM SIGOPS Operating Systems Review*, vol. 15, December 1981, p. 12
84. J. Rushby, S. Owre, N. Shankar, Subtypes for specifications: predicate subtyping in PVS. IEEE Trans. Softw. Eng. **24**(9), 709–720 (1998)

85. J.H. Saltzer, M.D. Schroeder, The protection of information in computer systems. Proc. IEEE **63**(9), 1278–1308 (1975)
86. R. Sandu, Analysis of acyclic attenuating systems for the SSR protection model, in *Proceedings of the 1985 IEEE Symposium on Security and Privacy*, April 1985, pp. 197–206
87. R.S. Sandhu, The schematic protection model: its definition and analysis for acyclic attenuating schemes. J. ACM **35**, 404–432 (1988)
88. R.R. Schell, P.J. Downey, G.J. Popek, Preliminary notes on the design of secure military computer systems. Tech. Rep. MCI-73-1, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA, 73
89. R. Schell, T.F. Tao, M. Heckman, Designing the GEMSOS security kernel for security and performance, in *Proceedings 8th DoD/NBS Computer Security Conference*, 1985, pp. 108–119
90. D.D. Schnackenberg, Development of a multilevel secure local area network, in *Proceedings of the 8th National Computer Security Conference*, October 1985, pp. 97–101
91. M.D. Schroeder, J.H. Saltzer, A hardware architecture for implementing protection rings. Commun. ACM **15**(3), 157–170 (1972)
92. J.S. Shapiro, J.M. Smith, D.J. Farber, EROS: a fast capability system, in *SOSP'99: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (ACM, New York, 1999), pp. 170–185
93. L.J. Shirley, R.R. Schell, Mechanism sufficiency validation by assignment, in *Proceedings 1981 IEEE Symposium on Security and Privacy*, Oakland (IEEE Comput. Soc., Los Alamitos, 1981), pp. 26–32
94. W.R. Shockley, R.R. Schell, TCB subsets for incremental evaluation, in *Proceedings Third AIAA Conference on Computer Security*, December 1987, pp. 131–139
95. A. Silberschatz, P.B. Galvin, G. Gagne, *Operating System Concepts*, 7th edn. (Wiley, New York, 2005)
96. Snort.org, Snort. http://www.snort.org/, last referenced 22 March 2009
97. Specware 4.2 Manual, Kestrel Technology, http://www.specware.org/documentation/4.2/languagemanual/SpecwareLanguageManual.pdf, 3 November 2008
98. J.M. Spivey, *Understanding Z: A Specification Language and Its Formal Semantics* (Cambridge University Press, Cambridge, 1988)
99. D.F. Sterne, On the buzzword "security policy", in *Proceedings of the IEEE Symposium on Research on Security and Privacy*, Oakland, CA (IEEE Comput. Soc., Los Alamitos, 1991), pp. 219–230
100. The Easter Egg Archive, Excel Easter Egg—Excel 97 flight to credits. http://www.eeggs.com/items/718.html, last accessed 19 February 2009
101. K. Thompson, Reflections on trusting trust. Commun. ACM **27**(8), 761–763 (1984)
102. S. Trimberger, Trusted design in FPGAs, in *Proceedings of the 44th Design Automation Conference*, San Diego, CA, June 2007
103. US Department of Commerce and Communications Security Establishment of the Government of Canada, Implementation guidance for FIPS PUB 140-2 and the cryptographic module validation program, initial release: 28 March 2003, last update: 10 March 2009. National Institute of Standards and Technology, Gaithersburg, MD, March 2009
104. US Department of Commerce, Security requirements for cryptographic modules, Federal Information Processing Standards Publication 140-2. National Institute of Standards and Technology, Gaithersburg, MD, May 2001
105. US Department of Commerce, Standards for security categorization of federal information and information systems, Federal Information Processing Standards Publication 199. National Institute of Standards and Technology, Gaithersburg, MD, February 2004
106. US Department of Commerce, Recommended security controls for federal information systems, NIST Special Publication 800-53 Revision 2. National Institute of Standards and Technology, Gaithersburg, MD, December 2007
107. US Department of Commerce, Security requirements for cryptographic modules, Federal Information Processing Standards Publication 140-3 (Draft: 07-13-2007). National Institute of Standards and Technology, Gaithersburg, MD, July 2007

108. US Department of Commerce, Security considerations in the system development life cycle, NIST Special Publication 800-64 Revision 2. National Institute of Standards and Technology, Gaithersburg, MD, October 2008
109. US Department of Commerce, Derived test requirements for FIPS PUB 140-2, Security requirements for cryptographic modules, 24 March 2004, Draft, CMVP program staff (NIST, CSE and CMVP laboratories). National Institute of Standards and Technology, Gaithersburg, MD. http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-2/fips1402DTR.pdf. Cited 7 April 2009
110. US Department of Defense, Trusted computer systems evaluation criteria (Orange Book) 5200.28-STD. National Computer Security Center, Fort Meade, MD, Dec. 1985
111. US Department of Defense, A guide to understanding trusted distribution in trusted systems, version 2, NCSC-TG-008. National Computer Security Center, Fort Meade, MD, December 1988
112. US Department of Defense, A guide to understanding trusted recovery in trusted systems, version 1, NCSC-TG-022. National Computer Security Center, Fort Meade, MD, December 1991
113. US Department of Defense, Defense Science Board task force on high performance microchip supply. Office of the Under Secretary of Defense For Acquisition, Technology, and Logistics, Washington, DC, February 2005
114. US Department of Defense, TRUST in integrated circuits, presolicitation notice, solicitation number: BAA07-24. Defense Advanced Research Project Agency, Microsystems Technology Office, Arlington, VA, March 2007. http://www.darpa.mil/mto/solicitations/baa07-24/index.html, cited 27 Mar 2009
115. D. Volpano, C. Irvine, Secure flow typing. Comput. Secur. **16**(2), 137–144 (1997)
116. D.R. Wichers, Conducting an object reuse study, in *Proceedings of the 13th National Computer Security Conference*, October 1990, pp. 738–747
117. M.V. Wilkes, R.M. Needham, The Cambridge model distributed system. ACM SIGOPS Oper. Syst. Rev. **14**(1), 21–29 (1980)
118. E. Witchel, J. Cates, K. Asanovic, Mondrian memory protection, in *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, October 2002
119. C. Zymaris, A comparison of the GPL and the Microsoft EULA. 2003. Cybersource. Retrieved 15 September 2008, from http://www.cybersource.com.au/cyber/about/comparing_the_gpl_to_eula.pdf