Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance[1]

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Technical Report
March 1999

[1]This work was sponsored by the FAA Technical Center and NASA Langley Research Center through contract NAS1-20334, and by DARPA and NSA through Air Force Rome Laboratory Contract F30602-96-C-0204.

Abstract

Automated aircraft control has traditionally been divided into distinct "functions" that are implemented separately (e.g., autopilot, autothrottle, flight management); each function has its own fault-tolerant computer system, and dependencies among different functions are generally limited to the exchange of sensor and control data. A by-product of this "federated" architecture is that faults are strongly contained within the computer system of the function where they occur and cannot readily propagate to affect the operation of other functions.

More modern avionics architectures contemplate supporting multiple functions on a single, shared, fault-tolerant computer system where natural fault containment boundaries are less sharply defined. *Partitioning* uses appropriate hardware and software mechanisms to restore strong fault containment to such integrated architectures.

This report examines the requirements for partitioning, mechanisms for their realization, and issues in providing assurance for partitioning. Because partitioning shares some concerns with computer security, security models are reviewed and compared with the concerns of partitioning.

Acknowledgments

I wish to acknowledge my deep appreciation for the support of Pete Saraceni of the Flight Safety Research Branch of the FAA Technical Center, and Ricky Butler of the NASA Langley Research Center. As often before, they provided encouragement and excellent technical advice throughout preparation of this report, and displayed stoic patience.

I am also very grateful to other researchers, and to engineers in several aerospace companies, who took the time to explain their concerns and approaches to me. I particularly benefited from extensive discussions with David Hardin, Dave Greve, and Matt Wilding of Collins, with Kevin Driscoll and Aaron Larsen of Honeywell, and with Hermann Kopetz of the Vienna Technical University. Ben di Vito of NASA Langley provided excellent comments and posed taxing questions on a draft version of this report.

Contents

List of Figures

Chapter 1

Motivation and Introduction

Digital flight-control functions in current aircraft are generally implemented by a *federated* architecture in which each function (e.g., autopilot, flight management, yaw damping, displays) has its own computer system that is only loosely coupled to the computer systems of other functions. A great advantage of this architecture is that fault containment is inherent: that is to say, a fault in the computer system supporting one function, or in the software implementing that function, is unlikely to propagate to other functions because there is very little that is shared across the different functions. To be sure, some functions interact with others, but these interactions are accomplished by the exchange of data, and functions can be designed to detect and tolerate a faulty or erratic data source.

The obvious disadvantage to the federated approach is its profligate use of resources: each function needs its own computer system (which is generally replicated for fault tolerance), with all the attendant costs of acquisition, space, power, weight, cooling, installation, and maintenance. Integrated Modular Avionics (IMA) has therefore emerged as a design concept to challenge the federated architecture [1,77]. In IMA, a single computer system (with internal replication to provide fault tolerance) provides a common computing resource to several functions. As a shared resource, IMA has the potential to diminish fault containment between functions: for example, a faulty function might monopolize the computer or communications system, denying service to all the other functions sharing that system, or it might corrupt the memory of other functions, or send inappropriate commands to their actuators. It is almost impossible for individual functions to protect themselves against this kind of corruption to the computational resource on which they depend, so any realization of IMA must provide *partitioning* to ensure that the shared computer system provides protection against fault propagation from one function to another that is equivalent to that which is inherent to the federated architecture.

The purpose of this report is to identify the requirements for partitioning in IMA and to explore topics in achieving those requirements with very high assurance. The next chapter, therefore, is concerned with the general requirements for IMA and partitioning, and the one following with issues in the implementation of IMA and the mechanisms for partitioning. The discussion in these chapters is deliberately more general than that in ARINC 651 ("Design Guidance for Integrated Modular Avionics") [1]: the ARINC document reflects aircraft practice, whereas we take a computer science perspective—in the hope that this will cast a new or different light on the issues. For this reason, our terminology is also more generic (e.g., we speak of processors and other basic components rather than line replaceable modules (LRMs)), and so are the component properties that we consider (e.g., we consider buses in general, not just avionic buses such as ARINC 629 [3]). In Chapter 4, we consider methods developed for specifying and analyzing computer security policies, since these share some concerns with partitioning, and have been the object of considerable study. We end with conclusions and suggestions for future work.

Chapter 2

Informal Requirements

To gain insight into the requirements for partitioning, we first need to examine the context provided by IMA and related developments in avionics.

2.1 Integrated Modular Avionics

It can be argued that the simplest interpretation of IMA envisions an architecture that technology has already rendered obsolete: an embedded systems version of the centralized time-shared "mainframe." Thanks to recent technological developments, powerful processors, large memories, and high-bandwidth local communications are all available as reliable and inexpensive commodity items, and these developments surely favor *less* rather than more centralization. Thus, this argument proceeds, a modern avionics architecture should be more, not less, federated, with existing functions "deconstructed" into smaller components, each having its own processor.

There is some plausibility to this argument, but the distinction between the "more federated" architecture and centralized IMA proves to be moot on closer inspection. A federated architecture is one whose components are very loosely coupled—meaning that they can operate largely independently. But the different elements of a function—for example, vertical and horizontal flight path control in an autopilot—usually are rather tightly coupled (and it is argued below that they should become even more tightly coupled), so that the deconstructed function would not be a federated system so much as a *distributed* one—meaning a system whose components may be physically separated, but which must coordinate to achieve some collective purpose. Dually, a centralized IMA architecture would not be a simple mainframe—for a computer system supporting flight functions must provide replicated and physically distributed hardware for fault tolerance, together with mechanisms for redundancy management. Consequently, a conceptually centralized architecture will be, internally, a distributed system, and the basic services that it provides will not differ in a significant way from those required for the "more federated" architecture.

Another contrarian point of view is that neither centralized IMA nor the "more federated" architecture offers significant benefits over current practice; the present federated architecture has been validated by experience, and modern hardware technology will reduce its cost penalty—so there is no reason to change it. The argument against this point of view is that it takes a very narrow interpretation of the costs associated with the current architecture, and therefore grossly underestimates them. One neglected cost is safety: the federated architecture has the advantage of natural fault containment, but it imposes a cost in poorly coordinated control, and complex and fault-prone pilot interfaces.

The current allocation of flight automation to separate functions is the result of largely accidental historical factors. Consequently, certain control variables that are tightly coupled in a dynamical sense are managed by different functions: for example, engine thrust is managed by the autothrottle, and pitch angle by the autopilot. Since a change in either of these variables affects the other, but there is no higher-level function that manages them in a coordinated manner, such conceptually simple services as "cruise speed control," "altitude select," and "vertical speed" have complex and imperfect implementations that are difficult to manage. For example, Lambregts [58, page 4] reports:

> "Because the actions of the autothrottle are not tactically coordinated with the autopilot, the autothrottle speed control constantly upsets the autopilot flight path control and vice versa, resulting in a notorious coupling problem familiar to every pilot. It manifests itself especially when excited by turbulence or windshear, to the point where the tracking performance and ride quality becomes unacceptable. The old remedy to break the coupling was to change the autopilot mode to ALTITUDE HOLD (e.g., the older B747-200/300). On newer airplanes, this problem has been reduced to an acceptable level for the cruise operation after a very difficult and costly development process, implementing provisions such as separation of the control frequency by going to very low autothrottle feedback gain, application of 'energy compensation,' turbulence compensation, and nonlinear windshear detections/compensation."

And again:

> "Due to the lack of proper control coordination, the autopilot ALTITUDE SELECT and VERTICAL SPEED modes never functioned satisfactorily... these problems resulted in development of the FLIGHT LEVEL CHANGE (FLC) mode that was first implemented on the B757/B767... however the mode logic depends on certain assumptions

> that are valid only for certain operations, so the logic can be tricked and cause an incorrect or poorly coordinated control response... as a result there have been a number of incidents where the FLC mode did not properly execute the pilot's command."

The lack of properly integrated control caused by the artificial separation of functions in the federated architecture is one of the factors that leads to the complex modes and submodes used in these functions, and thence to the "automation surprises" and "mode confusions" that characterize problems in the "flightcrew-automation interface." Numerous fatal crashes and other incidents are attributed to such human factors problems [27, Appendix D], but it is clear from their origins in the artificial separation of functions that these problems are unlikely to be solved by local improvements in the interfaces and cues presented to pilots. The plethora of modes, submodes, and their corresponding interactions also exacts a high cost in development, implementation, and certification. If this analysis is correct, the traditional federated architecture is a major obstacle to a more rational organization of flight functions, and IMA is the best hope for removing this obstacle.

The topics considered so far suggest that the appropriate context in which to examine partitioning for IMA is a distributed system in which flight functions (which might well be defined and subdivided differently than in the traditional federated architecture) are each allocated to separate processors (replicated as necessary for fault tolerance). In this model, we would need to consider partitioning to limit fault propagation *between* the processors supporting each function, but not *within* them. This model, however, overlooks a new opportunity that could be created by more fine-grained partitioning.

If functions have no internal partitioning, then all their software must be assured and certified to the level appropriate for that function. Thus, all the software in an autopilot function is likely to require assurance to Level A of DO-178B (this, the highest level of DO-178B, the guidelines for certification of airborne software [29,83], is for software whose malfunction could contribute to a catastrophic failure condition [28]), and this discourages the inclusion of any software that is not strictly essential to the function. While this may be a good thing in general, it also discourages inclusion of services that could have a positive safety impact, such as continuous self-test, or for-information-only messages to the pilot. More generally, partitioning within a processor could allow an individual function to be divided into software components of different criticalities; each could then be developed and certified to the level appropriate to its criticality, thereby reducing overall costs while allowing assurance effort to be focused on the most important areas. Without partitioning, the concern that a fault in less critical software could have an impact on the operation of more critical software necessarily elevates the criticality of the first to that of the second; partitioning would remove the danger of fault propagation, and allow the criticality of each software component to be assessed more locally.

The considerations of the previous paragraph suggest that for partitioning within a single processor it might be appropriate to limit attention to the case where the processor is shared by the components of only a single function. We might suppose that these components consist of one implementing the main function, and several others providing subsidiary services. Since a fault in the main component amounts to a fault in (this replica of) the overall function, there seems little point in protecting the subsidiary components from faults in the main component, and this suggests that partitioning could be asymmetric (the main component is protected from the subsidiary ones, but not vice versa). It is not clear whether such asymmetry would provide any benefit in terms of simplicity or cost of the partitioning mechanisms, but the point is probably moot since other scenarios require a symmetric approach. One scenario is support for several minor functions, for example undercarriage and weather radar, on a single processor. Where the functions are not required at the same time, partitioning could perhaps be achieved by giving each one sole command of its processor while it is active (this is similar to "periods processing" in the security context), but the more general requirement is for simultaneous operation with symmetric partitioning. The second scenario concerns very cost sensitive applications, such as single-engine general aviation aircraft. Here it may be desirable to run multiple major functions (such as autopilot and rudimentary flight management) on a single (possibly non-fault-tolerant) processor. There are even proposals to host these functions on mass-market systems such as Windows NT. Although one can be skeptical of this proposal (particularly if "free flight" air traffic control makes flight management data from general aviation aircraft critical to overall airspace safety), it seems worth examining the technical feasibility of symmetric partitioning for critical functions within a single processor.

The current federated architecture not only uses a lot of computer systems, it uses a lot of *different* computer systems: each function typically has its own unique computer platform. There is a high cost associated with developing and certifying software to run on these idiosyncratic platforms. Logically independent of IMA, but coupled to it quite strongly in practice, are moves to define standardized interfaces to the platforms that support flight functions, and to introduce some of the abstractions and services provided by an operating system. The ARINC 653 (APEX) [4] standard represents a step in this direction. Developments such as this could significantly reduce the cost of avionics software development and might stimulate creation of standard modules for common tasks that could be reused by different functions running on different platforms.

The design choices for partitioning interact with those for providing operating system services. The major decision is whether partitioning is provided above an operating system layer (Figure 2.1(a)), or above a minimal kernel (or executive) with most operating system services then provided separately in each partition (Figure 2.1(b)). The first choice is the way standard operating systems are structured (with partitions being client processes), but it has the disadvantage that partitioning then relies on a great deal of operating system software. The second choice is sometimes called the "virtual machine" approach, and it has the advantage that partitioning relies only on the kernel and its supporting hardware.[1]

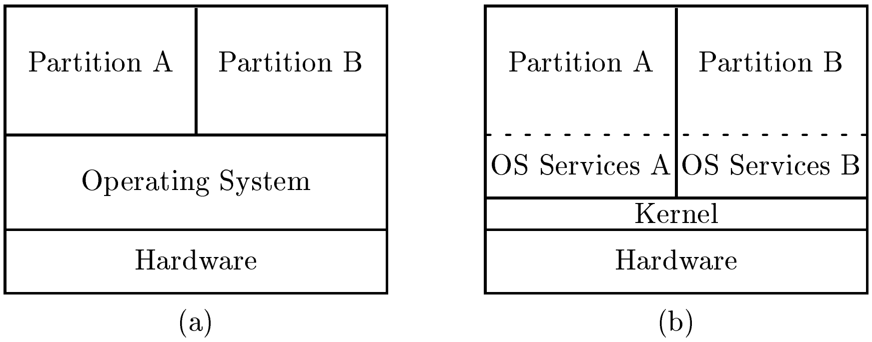

Figure 2.1: Alternative Operating System/Partitioning Designs

Another area where IMA has the potential to reduce costs is through improved dispatch reliability. Critical flight functions must tolerate hardware faults, and so they run on replicated hardware (typically quad-redundant or greater for primary flight control and displays, triple for autopilot and autoland, dual for flight management and yaw damping, and single for autothrottle). But despite the massive cost of providing a fault-tolerant platform for each function, and despite the large number of separate processors and other components available (there can be as many as 50 processors among the major functions of a large modern transport plane), the federated architecture does not provide a large margin of redundancy, nor operational flexibility. A single faulty processor in any function may be enough to prevent takeoff (thereby requiring maintenance in possibly less than ideal circumstances), and multiple faults afflicting such a function during flight might trigger a diversion, or have even more serious consequences. With IMA, in contrast, replicated processors are not bound to a specific function, but can be allocated as required: normal operation can continue as long as the total number of nonfaulty processors is sufficient to provide the required level of replication to each function. This increases

[1]Some operating systems use the second model. It was first employed in VM/370 [69], which served as the basis for a major early secure system development [7,35]. Fully virtualizing the underlying hardware is expensive, so later "μ-kernels" such as Mach and Chorus provided a more abstract interface. These also proved to have disappointing performance. Second-generation μ-kernels and comparable toolkits such as Exokernel [48], Flux [31], L4 [38], and SPIN [11] achieve good performance and introduce several implementation techniques relevant to the design of partitioned systems.

overall safety margins, while also allowing maintenance to be deferred (e.g., until the aircraft's schedule brings it to a major maintenance base) [40].[2]

The ability to exploit this increased redundancy and flexibility depends on a systematic approach to fault tolerance within functions (so that they are not tightly bound to a specific processor), and across the distributed coordination mechanisms of the IMA platform itself. Design of fault-tolerant systems is not only a massively difficult and expensive activity (the basic mechanisms of fault tolerance concern the coordination of distributed, real-time systems operating in the presence of faults, which are among the hardest problems in computer science) but is often a pervasive one: that is, mechanisms for fault tolerance and redundancy management in avionics are seldom encapsulated as an operating system or middleware service, and instead affect the design of every piece of software within the function. As a result, it is generally impossible to take software—or even the design for a piece of software—from one function and reuse it in another, or on another platform, even when standards such as APEX are used, for these standards concern only the mechanics of system calls and do not address the deeper concerns of systematic and transparent fault tolerance. Another reason for the pervasive influence of fault tolerance in current system designs is that the these mechanisms (and most others that involve coordination across multiple processors and functions) are seldom *compositional*—meaning that there is no *a priori* guarantee that elements that each work on their own will also work in combination. The massive resources expended on systems integration are a symptom of the lack of compositionality provided by current design practices.

Thus, full realization of the benefits of IMA requires adoption of modern concepts for systematic, compositional, fault-tolerant real-time system design [54]. These would reduce the pervasive impact of fault tolerance in avionics software development and provide cost savings and opportunities for reuse that could be much greater than those provided by lower-level standards such as APEX. Taken to their conclusion, such approaches could completely decouple the implementation of flight functions from that of their fault-tolerant platform, possibly enabling each to be certified separately. The impact of such developments on partitioning is, first, a requirement that the distributed partitioning mechanisms must themselves be robustly fault tolerant and, second, that these mechanisms must cooperate with operating system or kernel functions to provide the services required for systematic and transparent fault tolerance in the implementations of flight functions.

Summarizing this review of issues in IMA, we see that partitioning should be considered both within a single processor and across a distributed system, and that partitioning has interactions with the provision of operating system services and transparent fault tolerance. In the next section we examine the requirements for partitioning a little more closely.

[2] Current implementations of IMA allocate functions to processors at startup time; reconfiguration in flight is a future prospect.

2.2 Partitioning

The purpose of partitioning is fault containment: a failure in one partition must not propagate to cause failure in another partition. However, we need to be careful about what kinds of faults and failures are considered. The function in a partition depends on the correct operation of its processor and associated peripherals, and partitioning is not intended to protect against their failure—this can be achieved only by replicating functions across multiple processors in a fault-tolerant manner. After all, each function would be just as vulnerable to hardware failure if it had its own processor. Rather, the intent of partitioning is to control the additional hazard that is created when a function shares its processor (or, more generally, a resource) with other functions. The additional hazard is that faults in the design or implementation of one function may affect the operation of other functions that share resources with it.[3] Now a design or implementation fault in a flight function is surely a very serious event and it might be supposed that (a) such faults are so serious that it does not matter what else goes wrong, or (b) certification ensures that such faults cannot occur. Both suppositions would, if true, diminish the requirements for partitioning.

The first point is easily refuted: the whole thrust of aircraft certification is to ensure that failures are independent (and individually improbable) if their combination could be catastrophic. Thus, while a design fault in, say, the autothrottle function would be serious, appropriate design and system-level hazard analysis will ensure that it is not catastrophic, *provided* other functions do not fail at the same time. Allowing a fault in this function to propagate to another (e.g., autoland) would violate the assumption of independent failures. Thus, far from a fault in a critical function being so serious as to render concern for partitioning irrelevant, it is the need to contain the consequences of such a fault that renders partitioning essential (and elevates its criticality to at least that of the most critical function supported).

It could be argued that both functions will certainly be lost if their shared processor fails, so they surely would not be sharing if their correlated failure could be catastrophic. This overlooks a couple of points. First, malfunction or unintended function is often more serious than simple loss of function, and the consequences of a propagating fault (unlike those of a processor failure) may well be of these more serious kinds. For example, a buffer overflow in one function might overwrite data in another, leading to unpredictable consequences. (The Phobos I spacecraft was lost in just this circumstance—when a keyboard buffer overflowed into the memory of a critical flight control function [14, 17].) Second, the increased interdependency wrought by IMA may introduce shared resources—and hence paths for fault

[3] Partitioning can also limit the consequences of transient hardware faults (by containing them within the partition that is directly affected), but that is a side benefit, not a requirement.

propagation—that are less obvious and more easily overlooked than shared processors. For example, functions in separate processors, where correlated failure would not be anticipated (and would not occur in a federated architecture) might become vulnerable to fault propagation through a shared bus in an IMA architecture.

Returning to the second point raised above (that certification ought to ensure the absence of design and implementation faults), note that certification requires assurance proportional to the consequences of failure. In a federated architecture, such consequences are generally limited to the function concerned, so that assurance is related to the criticality of that function. But if the failure of one function could propagate to others, then a low-criticality (and correspondingly low assurance) function might cause a high-criticality function to fail. This means that either all functions that share resources must be assured to the level of the most critical (such elevation in assurance levels is directly contrary to one of the goals of IMA), or that partitioning must be used to eliminate fault propagation from low-assurance functions to those of high criticality. When different functions already happen to have the same level of assurance, the need for partitioning may not be so great, and it has been suggested that functions with software assured to Level A of DO-178B may be allowed to share resources without partitioning. Note, however, that a fault that causes one function to induce a failure in another might not affect the operation of the first (as noted above, a temporary buffer overflow can have this property). And although certification requires assurance of the absence of such unintended effects, as well as positive assurance that the intended function is performed correctly, it is generally much harder to provide the first kind of assurance than the second. Furthermore, shared resources create new pathways for the propagation of unintended effects, and these pathways might not have been considered when assurance was developed for the individual functions. Consequently, partitioning seems advisable even when the functions concerned are of the same level of criticality, and all software is assured to the same level.

Summarizing the discussion in this chapter, we may conclude that future avionics architectures will have the character of distributed, rather than federated, systems, and that multiple functions, of possibly different levels of criticality and assurance, will be supported by the same system. Resources, such as processors, communications buses, and peripheral devices, may be shared between different functions. Shared resources introduce new pathways for fault propagation, and these hazards must be controlled by partitioning.

Because partitioning is required to prevent fault propagation through shared resources, a suitable benchmark or "Gold Standard" for the effectiveness of partitioning would seem to be a comparable system (intuitively a federated one) in which there are no shared resources. This is captured in the following.

Gold Standard for Partitioning

> A partitioned system should provide fault containment equivalent to an idealized system in which each partition is allocated an independent processor and associated peripherals, and all inter-partition communications are carried on dedicated lines.

Although this Gold Standard provides a suitable mental benchmark for designers and certifiers of partitioning mechanisms for IMA, it is less useful as a "contract" with the "customers" of such mechanisms. These customers—that is, those who develop software for the functions that will run in the partitions of an IMA architecture—are assured that their software will be as well protected in a partition as if it had its own dedicated system, but they are not provided with a concrete environment in which to develop, test, and certify that software. The Gold Standard implies that the environment provided by the partitioned system to a particular application function must be indistinguishable from an idealized system dedicated to that function alone, but this idealized system is just that—an imaginary artifact—and not one suitable for testing and evaluating real-world software. The only environment actually available is the partitioned system itself, so its customers need a contract expressed in terms of that environment. This can be done as follows:[4] instead of comparing the environment perceived by the software in a partition to that of an idealized, dedicated system, we require that the environment (whatever it is) is one that is totally unaffected by the behavior of software in other partitions. This leads to the following alternative statement of our Gold Standard.

Alternative Gold Standard for Partitioning

> The behavior and performance of software in one partition must be unaffected by the software in other partitions.

This formulation is not only simpler and more direct than that involving an idealized system, but it also suggests how the customers of a partitioned system can develop and evaluate their software—for if software in one partition is unaffected by that in other partitions, it will run the same (in terms of both behavior and performance) whether the other partitions are inhabited or empty. Thus, in particular, individual software functions can be developed and certified using the real environment of the partitioned system, but with the other partitions empty (or, more likely, containing stubs to provide the data sources and sinks required by the function under examination). The Alternative Gold Standard ensures that the certified software will behave exactly the same when those other partitions are inhabited by real (and possibly faulty) functions.

[4] I am grateful to David Hardin, Dave Greve, and Matt Wilding of Collins Commercial Avionics for explaining this approach and its motivation to me [111].

A problem with the Alternative Gold Standard is apparent in the mention of "data sources and sinks" in the discussion above: software functions residing in separate partitions are seldom completely independent—some provide data or control inputs to others. This means that "unaffected by the software in other partitions" needs to be qualified in some way that allows the effects of intended communications while excluding those that are unintended. Thus, although the Alternative Gold Standard is more attractive than the original one as a requirements definition for partitioning *isolated* functions, it needs further development before it can serve as a gold standard for the more general case of partitioned but interacting functions. When restricted to isolated functions, the basic and the Alternative Gold Standards are very similar; indeed, if suitably formalized, each would be definable in terms of the other.

The original formulation of the Gold Standard has the advantage that it focuses attention on the structural differences between a partitioned system and a federated one. These structural differences introduce two classes of hazards into a partitioned system: a fault in one partition could corrupt code, control signals, or data (in memory or in transit) belonging to another, or it could affect the ability of another partition to obtain access to, or service from, a shared resource (such as the processor or a bus). In considering issues in the design and assurance of partitioned systems, it is therefore useful to distinguish two dimensions—spatial and temporal—corresponding to these two classes of hazards.

Spatial Partitioning

> Spatial partitioning must ensure that software in one partition cannot change the software or private data of another partition (either in memory or in transit), nor command the private devices or actuators of other partitions.

Temporal Partitioning

> Temporal partitioning must ensure that the service received from shared resources by the software in one partition cannot be affected by the software in another partition. This includes the performance of the resource concerned, as well as the rate, latency, jitter, and duration of scheduled access to it.

The mechanisms of partitioning must block the spatial and temporal pathways for fault propagation by interposing themselves between avionics software functions and the shared resources that they use. In this way, the partitioning mechanisms can control or "mediate" access to shared resources. In the next chapter, we consider the mechanisms that can be used to provide mediation in each of the two dimensions of partitioning.

Chapter 3

Issues and Mechanisms

As discussed in the previous chapter, issues in partitioning arise at two levels: within a single processor, and across a distributed system. Issues in partitioning also interact with those in fault tolerance. We consider these topics separately below, and further separate them into consideration of spatial and temporal partitioning.

3.1 Partitioning Within a Single Processor

We start by considering partitioning within a single processor. We sometimes use the neutral term *application* to refer to the computational entity within each partition; this could be a complete avionics function (e.g., a yaw damper), or a part of one. Depending on the implementation, an application could correspond to the operating system notions of *process* or *virtual machine*, or it could be some different notion. An application will generally be composed of smaller units of computation that are called or scheduled separately; we generally refer to these as *tasks*. Again depending on the implementation, these may correspond to an operating system notion such as *thread* or *lightweight process*. Partitioning must prevent applications interfering with one another, but the tasks within a single application are not protected from each other. We focus first on partitioning in the spatial dimension.

3.1.1 Spatial Partitioning

The basic concern of spatial partitioning is the possibility that software in one partition might write into the memory of another: memory is often pictured as a one- or two-dimensional grid, hence the reference to the spatial dimension for this aspect of partitioning. Memory includes that used to store programs as well as data, although in embedded systems it is sometimes possible to hold the former in ROM, where it cannot be overwritten by errant software.

Hardware mediation provided by a memory management unit (MMU) is the usual way to guard against violations of spatial partitioning. The details vary from one processor design to another, but the basic idea is that the processor has (at least) two modes of operation and, when it is in "user" mode, all accesses to memory addresses are either checked or translated using tables held in the MMU. A layer of operating system software (generally called the *kernel*) manages the MMU tables so that the memory locations that can be read and written in each partition are disjoint (apart, possibly, from certain locations used for inter-partition communications). The kernel also uses the MMU to protect itself from being modified by software in its client partitions, and must be careful to manage the user/supervisor mode distinctions of the processor correctly to ensure that the mediation provided by the MMU cannot be bypassed. (In particular, entry and exit from the kernel needs to be handled carefully so that software in a partition cannot gain supervisor mode; some processors have had design flaws that make this especially difficult [43].)

Software executing in a partition accesses processor registers such as accumulators and index registers as well as memory. Generally, the kernel arranges things so that the software in one partition executes for a while, then another partition is given control, and so on; when one partition is suspended and another started, the kernel first saves the contents of all the processor registers in memory locations dedicated to the partition being suspended, and then reloads the registers (including those in the MMU that determine which memory locations are accessible) with values saved for the partition that executes next. The software in the partition resumes where it left off and cannot tell (apart from the passage of time while it was suspended) that it is sharing the processor with other partitions.

The description just given resembles classical time-sharing, where partitions can be suspended at arbitrary points and resumed later. Some variations are possible for embedded systems. For example, if partitions are guaranteed an uninterruptible time slice of known duration, they can be expected to have finished their tasks before being suspended and can then be restarted in some standard state, rather than resumed where they left off. This eliminates the cost of saving the processor registers when a partition is suspended (but at least some of them—including the program counter—must be restored to standard values when the partition is restarted). We can refer to the two types of partition swapping arrangements as the *restoration* and *restart* models, respectively.

In either case, the requirement on the mediation mechanisms managed by the kernel is that the behavior perceived across a suspension by the software in each partition is predictable without reference to anything external to the partition. In the "restoration" model, the processor state must be restored to exactly what it was before suspension; in the "restart" model, it must be restored to some known state. It may be acceptable in the latter case to specify that some registers may be "dirty" on restart and that the software in a partition is required to work correctly without making assumptions on their initial contents—this saves the cost of restoring these registers to standard values (obviously, the program counter and MMU registers must be restored).[1] The requirement to make behavior predictable across the suspension and resumption of a partition generates in turn the requirement that the operation of the processor must be specified precisely and accurately with respect to all of its registers—for it is important that register saving and restoration or reinitialization should not overlook visible minor registers such as condition codes and floating point/multimedia modes, and that hidden registers, such as those associated with pipelines and caches, really are hidden. (Again, processors often have design glitches, or errors and omission in documentation, that make it difficult to accomplish this [98].)

In the approach just outlined, the mechanisms of spatial partitioning comprise the processor and its MMU, and the kernel. There is much advantage, from the point of view of assurance and formal specification, if these mechanisms are simple. Unfortunately, commodity processors, their MMUs, and associated features such as memory caches, are generally designed for high performance and extensive functionality rather than simplicity. Although a fast processor is often desired, the functionality of MMUs and cache controllers generally exceeds that required for embedded systems; MMUs, in particular, are usually designed to provide a flexible virtual memory and contain large associative lookup tables—whereas for partitioning, a simple fixed memory allocation scheme would be adequate.[2] The latter would also be far less vulnerable to bit-flips caused by single-event upsets (SEUs) than a traditional million-transistor MMU. However, because they are usually highly integrated with their processor, it can be difficult or even impossible to replace MMUs and cache controllers with simpler ones, but consideration should be given to this issue during hardware selection.

An alternative to spatial partitioning using hardware mediation is Software Fault Isolation (SFI) [108]. The idea here is similar to array bounds checking in high-level programming languages, except that it is applied to all memory references, not just those that index into arrays. By examining the machine code of the software in a partition, it is possible to determine the destinations of some memory references and

[1]Although partitioning has much in common with computer security, this is one aspect where they differ: "dirty" registers are anathema in computer security because they provide a channel for information flow from one partition to its successor. The issues underlying this difference are considered on page 53 in Chapter 4.

[2]MMUs are also heavily optimized for speed: in some architectures, the MMU will start a read from the memory using the current page map before it has determined whether that is still valid; if it is not valid, the MMU squashes the bus read transaction before it completes. Also, for efficiency, multiple copies may be maintained for some of the associative lookup tables, and these must be kept consistent with each other. All this is done in the context of speculative out-of-order execution, where providing assurance for correctness of these optimizations is nontrivial. A separate problem is the timing uncertainty introduced by these optimizations: ratios of 2 to 1 between average-case and worst-case timings are not uncommon [51] (see also `http://www.intelligentfirm.com/`).

jumps and hence to check, statically, whether they are safe. Memory references that indirect through a register cannot be checked statically, so instructions are added to the program to check the contents of the register at runtime, immediately prior to its use. By using more elaborate static analysis or program verification techniques (e.g., to ensure that an index register has not been changed since last checked), it is possible to minimize the number of runtime checks; by using modest optimizations of this kind, an overhead of just 4% has been reported for the runtime checks of SFI [108].

Static (i.e., compile-time) analysis of information flow within individual programs written in high-level languages has long been a topic in computer security. In its simplest form, some of the variables used by the program are labeled HIGH and some LOW, and the goal is to check whether information from a HIGH variable can ever influence the final value of one labeled LOW. Techniques for information flow analysis include approximate methods similar to typechecking [21, 107] or to data flow analysis [6], as well as exact methods [62] and those that rely on formal proof [80]. It is possible that approaches based on these techniques could reduce, or even eliminate, the runtime overhead of SFI.

Although SFI usually imposes a small overhead on memory references within a partition, it can greatly reduce the cost of controlled references or procedure calls across partitions (compared with hardware mediation, since the cost of a partition swap is avoided). However, for reasons discussed later (page 27), such cross-partition references may not be acceptable in some partitioned architectures, so the advantage would be moot in those cases.

A disadvantage of SFI compared with hardware-mediated partitioning is that it imposes an additional analysis and certification cost on every program, whereas hardware mediation has the one-time cost of designing, implementing, and certifying the partitioning mechanisms of the kernel and its supporting hardware. On the other hand, the analysis required for SFI lends itself to powerful automation (cf. "extended static checking" [22], and "proof carrying code" [79]) where the certification cost would be transferred to the one-time cost of certifying the tools.

Even without automation, SFI may have advantages of cost and simplicity in "asymmetric" applications where a single function is allocated to a processor but it is desired to include some less critical "nice-to-have" features. These could be partitioned from the main safety-critical function by SFI, while the latter runs unchanged. SFI might also be cost-effective in partitioning functions of similar assurance levels that already require significant analysis (e.g., two Level A functions). And SFI could also be used to provide additional protection *within* partitions (i.e., among tasks) established by hardware mediation.

One concern about SFI, especially when static analysis is used to optimize away many of the runtime checks, is that it provides little protection against hardware faults (e.g., SEU-induced bit-flips) that cause memory addresses that were correct when analyzed to be turned into ones that are incorrect when executed. The bad memory reference will be caught only if a runtime check is in the right place; a hardware MMU, on the other hand, mediates *every* reference at its time of execution. It was earlier stated that the purpose of partitioning is to protect functions against faults of design and implementation in other functions, not to guard against hardware faults—since these could afflict the function even if it had its own dedicated processor—but a hardware fault that leads to a violation of partitioning is not a fault that would have afflicted the function if it had its own processor, so it seems that the concern is legitimate. However, a little analysis shows that the increased exposure to hardware faults is small. Suppose the function in which we are interested shares its processor with n other functions of similar size, and that the probability of an SEU hitting any one of them is p. Suppose further that the probability that an SEU in one function will cause it to violate SFI partitioning and to afflict some other function is q. Then the probability of an SEU directly or indirectly affecting the original function changes from p to $(1+q)p$ when the function is moved from a dedicated to a shared processor. (Notice that this is independent of n: the chance of an SEU hitting *somewhere* increases by a factor of n, but the chance that the consequent memory error affects the function concerned is reduced by the same factor.) This small increase in probability is unlikely to be significant, and we conclude that the possibility of SEU-induced addressing errors does not invalidate SFI.

Perhaps surprisingly, it is some implementations of hardware-mediated partitioning that seem more vulnerable to this kind of fault scenario. Although an SEU in an individual function cannot lead to a violation of partitioning when memory references are mediated by an MMU, an SEU in the MMU itself could be quite dangerous. If the MMU is a large device with millions of transistors, then the possibility of an upset cannot be overlooked, and a change to one bit in an address translation register may cause the memory references of one partition systematically to infringe on the memory of another. It seems to me that in designs where it is possible to provide a custom MMU, it would be prudent to ensure that this is either fault tolerant, or that it merely checks rather than translates addresses (so that a double fault would be needed to violate partitioning); best of all might be relocation or checking with hardwired values.

So far, our consideration of partitioning has considered only the processor and the memory, and has assumed that different partitions are meant to be isolated from each other; we now need to consider inter-partition communications, and devices. Like partitioning itself, there are two dimensions to inter-partition communication: the spatial dimension is concerned with where and how data is transferred from one partition to another, while the temporal dimension is concerned with whether and how synchronization is performed, and how one partition invokes services from another. We postpone consideration of the latter topics to the discussion of temporal partitioning in Section 3.1.2 and focus here on the spatial dimension.

The obvious way to communicate data from one partition to another is to copy it from a buffer in memory belonging to the first partition into a separate buffer in the memory of the second. Because only the kernel has access to the memory of both partitions, it must perform the copying and, since it generally runs without memory protection, it must check carefully against buffer overruns. A more efficient scheme uses a single buffer in memory locations that are among those the sending partition can write and the receiver can read (both MMU and SFI forms of memory protection can do this); data can then be copied into the shared buffer by the sending partition without the active participation of the kernel. The receiving partition must assume that the sending one can write arbitrary data anywhere in their shared buffers whenever it has control, and its verification must be performed under this assumption. It seems cleanest if separate buffers are used for each direction of transfer, but bidirectional buffers may also be acceptable. It is, however, important that separate buffers are used for each pair of partitions (otherwise, partition A could overwrite the data of B in C's single input buffer).

Observe that it is important to restrict inter-partition communications to those that are intended: one partition should be able to send data to another only if that communication is authorized in the specification of the system configuration (and the receiving partition must then have a buffer to receive it). A related topic is how one partition should name the other partitions with which it communicates. Absolute addresses (e.g., "send this datum to Partition 7") lead to a rigid and fragile system organization and are to be deprecated on this account. Functional addresses (e.g., "send this datum to the pitch autopilot") are little better: they build assumptions about the system structure into individual applications and limit the opportunities for reuse and reconfiguration. Relative addressing (e.g., "send this datum out on my Port 7") allows the binding of names to specific inter-partition communication channels to be postponed until system configuration time (and may allow some dynamic reconfiguration), but requires a database to record what type of data or service is provided (or expected) on a given port. The best arrangement may be one where partitions use the type of data or service provided or expected as the name of the port concerned (e.g., "send this datum out on my `air-data-samples` port," or "get me an `air-data-sample`"); the binding of these names to inter-partition channels can be done during system configuration, or at runtime. In the latter case, we have something like a publish-subscribe architecture [81]; this provides excellent support for dynamic reconfiguration, but its application to life-critical systems is still an issue for research. (Some avionics systems use this type of naming or addressing scheme, but not in a way that is tightly integrated with their fault-tolerance mechanisms.)

Software in one partition should not make assumptions about when tasks in other partitions are scheduled (tasks within some partitions may be dynamically scheduled); this, combined with normally asynchronous communication, means that care is needed when communicating time-sensitive data. For example, a task that collects from its input buffer a sensor sample contributed by another partition needs to know when that sample was taken. The usual arrangement is to attach a timestamp to the sample (since both partitions are running in the same processor, they have access to a common clock). However, the utility and interpretation of a sensor sample depends not only on its age, but also on its accuracy and the dynamics of the physical process being measured (e.g., an altimeter reading that is 1 second old is much less useful if the aircraft is landing than if it is in cruise). Some of these factors are likely to be much better known to the partition that provides the sensor sample than to the one that receives it, and duplicating the knowledge in both places is expensive and raises the problem of ensuring consistency. Instead, it seems best if the provider of the data also provides a compact description of its temporal interpretation. Kopetz has made an interesting proposal of this kind under the name *temporal firewall* [52,56], which exists in two variants. A *phase-insensitive* sensor sample is provided with a time and a guarantee that the sampled value is accurate (with respect to a specification published by the partition that provides it) *until* the indicated time. For example, suppose that engine oil temperature may change by at most 1% of its range per second, that its sensor is completely accurate, and that the data is to be guaranteed to 0.5%. Then the sensor sample will be provided with a time 500 ms ahead of the instant when it was sampled, and the receiver will know that it is safe to use the sampled value until the indicated time. This is much more useful than a timestamp that merely records when the sample was taken. A *phase-sensitive* temporal firewall is used for rapidly changing processes where state estimation is required; in addition to sensor sample and time, it provides the parameters needed to perform state estimation. For example, along with the sampled altitude it may supply vertical speed, so that altitude may be estimated more accurately at the time of use.

In addition to communications between partitions, we must examine communications between partitions and devices. Devices, which include sensors and actuators as well as peripherals such as mass storage, have implications for both temporal and spatial partitioning. Most devices raise an interrupt when data is available, or when they need service. Such interrupts affect the timing and locus of control, and consideration of their impact is postponed to the discussion on temporal partitioning in Section 3.1.2; here we concentrate on the relationship of devices to spatial partitioning. Devices impact spatial partitioning in three ways: they need to be protected against access by the wrong partition, they must not be allowed to become agents for violating partitioning, and they may themselves need to be partitioned.

The simplest case is where a device "belongs" to some partition and should not be accessed by others. Most modern processors use memory-mapped I/O, meaning that interaction with devices is conducted by reading and writing to registers that are referenced like ordinary memory locations. In these cases, the mechanisms (MMU or SFI) used to provide ordinary memory protection can also protect devices. If memory protection is insufficiently fine-grained to permit devices to be allocated to partitions as desired, then it will be necessary to create special device management partitions that own several devices but are trusted to keep them separate. Similar arrangements will be necessary if several devices are attached to a data bus or remote data concentrator (and may also be useful if multicast communication services are desired). Of course, the trust in such "multiplexing" partitions needs to be justified by suitable verification and assurance. An alternative to providing device management partitions is to perform these functions in the kernel. The argument against doing this is that the properties of the kernel must be assured to a very high degree, so there is much advantage to keeping its functionality as simple as possible. It should be easier to provide assurance for a kernel that provides memory protection, plus separate device management partitions, than for a kernel having both functions.

Some devices may be shared by more than one partition. Such devices come in two forms: those that need protection and those that do not. An example of the former is a sensor that periodically places a sample in a device register. There seems no harm in allowing two partitions both to have read access to the memory location containing that device register. Devices that accept commands are more problematical in that faulty software in one partition may issue commands that render the device inoperable or otherwise unavailable to other partitions. Protection by a special device management partition seems necessary to mediate access in these cases. (The Clementine spacecraft was lost when a software fault caused garbage to be sent over an unmediated bus, where it was interpreted by an attached device as a command to fire all the thrusters without limit.) Notice that such a device management partition must play a more active role in checking or controlling the device than the simple "multiplexing" device management partitions described earlier.

Device management partitions also are necessary to mediate access to truly shared devices such as mass storage. In these cases, it is usual for the device manager to synthesize a service (e.g., a file system) rather than just mediate access to the raw device (e.g., a disk), and to partition the service appropriately (e.g., with a separate "virtual" file system for each client partition). A device manager of this kind poses challenges to assurance that are similar to those of the main memory partitioning mechanism, since flaws could allow one client partition to write into areas intended for another.

Mass storage and other devices that transfer large amounts of data at high speed generally do so by direct memory access (DMA) rather than through memory-mapped device registers (which are limited to a few bytes at a time). Depending on the processor and memory architecture, DMA devices may be able to address memory directly, without the mediation of the MMU. This arrangement has the potential to violate partitioning since faulty software may instruct the device to use a region of memory belonging to some partition other than its own; a fault in the device itself could have a similar effect. A simple solution is to interpose some checking or limiting mechanism into the device's memory address lines (e.g., by cutting or hard-wiring some of them) so that the range of addresses it can generate is restricted to lie within that of the partition that manages it. Another solution is to isolate each DMA device to a private bus with a dual-ported memory bridging the private and main system buses.

3.1.2 Temporal Partitioning

Our context is real-time embedded systems, where correctness requires not only that the right results are produced, but that they are produced at the *right time*. The concern of temporal partitioning is to ensure that activities in one partition do not disturb the timing of events in other partitions.

The most gross concerns are that faulty software in one partition might monopolize the CPU, or that it might crash the system or issue a HALT instruction—effectively denying service to all other partitions. Other scenarios that can cause a partition to fail to relinquish the CPU on time include simple schedule overruns, where particular parameter values cause a computation to take longer than its allotted time, and runaway executions, where a program gets stuck in a loop.

Although their manifestations are in the temporal dimension, system crashes and instructions that halt the CPU are usually prevented by the mechanisms of spatial partitioning. In particular, HALT and other dangerous instructions usually cannot be issued (or, rather, they cause a trap to the kernel) when in user mode. There are reports, however, that some steppings of some commodity processors have untrapped instructions that can halt the CPU, or user-mode instructions that can "hang" when supplied with certain parameters (e.g., see `http://www.x86.org`; also [98] notes 102 bugs reported up to 1995 in various versions and steppings of the Intel 80X86 architecture, while [8] documents a comparable number in later processors). It is important to know these characteristics of the precise stepping of the processor employed (which may require a nondisclosure agreement), but it is difficult to provide a complete solution to such untrapped hardware flaws. Perhaps the best that can be done is to use SFI-like techniques and to scan the machine code of each application and insert runtime checks as necessary to prevent execution of dangerous instructions or parameter values (a purely static check will be inadequate if parameter values can be constructed or modified—either under program control or by an SEU—at runtime).

The last-ditch escape from a halted or locked-up CPU is a watchdog timer interrupt managed by the kernel. This is not certain to provide recovery, however, unless the basic kernel design is correct: for example, design faults in the Magellan spacecraft led to a runaway execution in which a program sat in a loop that did nothing but reset the watchdog timer [18, pp. 209–221] [25, 50],[3] and not all halted or "hung" processors respond to the timer interrupt. Recovery in these dire cases usually depends on a system reset (or cycling the power supply, which causes a reset), which may be invoked either manually or by other components in a distributed fault-tolerant system (which is how Magellan recovered).

Runaway executions in the kernel, lockups, and untrapped halt instructions could all afflict a processor dedicated to a single function, and so their treatment is more in the domain of system-level design verification or fault tolerance than partitioning. Overruns or runaways within a function, however, are genuinely the concern of partitioning and are usually controlled through timer interrupts managed by the kernel: the kernel sets a timer when it gives control to a partition; if the partition does not relinquish control voluntarily before its time is up, the timer interrupt will activate the kernel, which then will then take control away from the overrunning partition and give it to another partition under the same constraints.

Merely taking control away from an overrunning partition does not guarantee that other partitions will be able to proceed, however, for the overrunning partition could be holding some shared device or other resource that is needed by those other partitions. The kernel could break any locks held by the errant partition and forcibly seize the resource, but this may do little good if the resource has been left in an inconsistent state. These considerations reinforce the earlier conclusion that devices and other resources cannot be directly shared across partitions. Instead, a management partition must own the resource and must manage it in such a way that behavior by one client partition cannot affect the service received by another.

Another problem can arise if the overrunning partition is performing some service on behalf of another partition: it will generally be necessary to notify the invoking partition (the next time it is scheduled) of the failure of the service provided by the other. The invoking partition must have enough fault tolerance that it can do something sensible despite the failure of the service. It may also be necessary for the kernel to perform some remedial action on the partition that overran its allocation. This could force that partition to do a restart next time it is scheduled,

[3]The flaw in Magellan was in the design of its kernel (sensitive data structures were manipulated outside the protection of a critical section, so an interrupt could leave them in an inconsistent state). Such flaws would be unconscionable in a safety-critical system: the design of the core hardware and software mechanisms simply have to be correct in these systems. In addition to skilled and experienced designers, formal methods of specification and analysis may be valuable for this purpose (design diversity is implausible at these levels).

or could simply notify the partition of its failure and leave recovery (e.g., the killing of orphans) to the operating system functions resident in that partition.

Timeout mechanisms such as those just described ensure that each partition will get *enough* access to the CPU and other resources, but real-time systems need more than this: the tasks within partitions need to get access to the CPU and to devices and other resources at the *right* time, and with great *predictability*. This means that discussion of temporal partitioning cannot be divorced from consideration of scheduling issues. The real-time tasks within a partition generally consist of *iterative* tasks that must be run at some fixed frequency (e.g., 20 times a second) and *sporadic* tasks that run in response to some event (e.g., when the pilot presses a button); iterative tasks often require tight bounds on *jitter*, meaning that they must sample sensors or deliver outputs to their actuators at very precise instants (e.g., within a millisecond of their deadline), and sporadic tasks often have tight bounds on *latency*, meaning that they must deliver an output within some short interval of the event that triggered them.

There are two basic ways to schedule a real-time system: statically or dynamically. In a static schedule, a list of tasks is executed cyclically at a fixed rate. Tasks that need to be executed at a faster rate are allocated multiple slots in the task schedule. Even sporadic tasks are scheduled cyclically (to poll for input and process it if present). The maximum execution time of each task is calculated, and sufficient time is allocated within the schedule to allow it to run to completion: thus, one task never interrupts execution of another (although a task may be terminated if it exceeds its allocation). Notice that this means that a long-duration task may need to be broken into several smaller pieces to make room for short tasks with higher iteration rates. The schedule is calculated during system development and is not changed at runtime (although it may be possible to select among a fixed collection of different schedules at runtime according to the current operating mode).

In a dynamic schedule, on the other hand, the choice and timing of which tasks to dispatch is decided at runtime. The usual approach allocates a fixed priority to each task, and the system always runs the highest-priority task that is ready for execution. If a high-priority task becomes ready (e.g., due to a timer or external interrupt) while a lower-priority task is running, the lower-priority task is interrupted and the high-priority task is allowed to run. Note that this requires a context-switching mechanism to save and later restore the state of the interrupted task. The challenge in dynamic scheduling is to allocate priorities to tasks in such a way that overall system behavior is predictable and all deadlines are satisfied. Originally, various plausible and ad-hoc schemes were tried (such as allocating priorities on the basis of "importance"), but the field is now dominated by the rate monotonic scheduling (RMS) scheme of Liu and Layland [65]. Under RMS, priorities are simply allocated on the basis of iteration rate (the highest priorities going to the tasks with the highest rates) and, under certain simplifying assumptions, it can be shown that all tasks will meet their deadlines as long as the utilization of the processor does not exceed 69% (the natural logarithm of 2—higher utilizations are possible when the task iteration rates satisfy certain relationships). Some of the simplifying assumptions (e.g., that the context-switch time is zero, and that tasks do not share resources) have been lifted or reduced recently [61, 68, 96].

The choice between static and dynamic scheduling is a contentious one (Locke [66] provides a good discussion). The basic arguments in favor of static scheduling are its complete predictability and the simplicity of its implementation; the arguments against are that all tasks must run at a multiple of the basic iteration rate (so that some run more or less frequently than is ideal for their control function), the handling of sporadic tasks is wasteful, and long-running tasks must be broken into multiple, separately scheduled pieces (to make room for tasks with faster iteration rates). The arguments in favor of dynamic scheduling are that it is more flexible and copes better with occasional task overruns; the arguments against hinge on the difficulty of giving complete assurance that a given task set will always meet its deadlines under all circumstances. (The factors that must be considered are complex and not all are fully characterized; errors of understanding or judgment are not uncommon. For example, the much publicized communications breakdowns between the 1997 Mars Pathfinder and its Sojourner rover were due to priority inversions in its RMS scheduler.[4] Priority inversions are a well-understood problem in dynamically scheduled systems, with a well-characterized solution called "priority inheritance" [20, 96] that was available, but not used, in the commercial real-time executive used for Pathfinder.)

The mechanisms of both static and dynamic scheduling have to be modified to operate in a partitioned environment, and these modifications change some traditional expectations about the tradeoffs between the two approaches; in addition, partitioning creates opportunities for hybrid approaches that combine elements of both basic mechanisms. The traditional scheduling problem is to ensure satisfaction of all deadlines, given information about the rate and duration of the tasks concerned. It is assumed that this information is accurate; if it is not—if, for example, some task runs longer or requests service more often than expected—then the system may fail. When all the tasks in the system are contributing to some single application, such a failure may be undesirable but will not have repercussions beyond those consequent on the failure of the application concerned. In a partitioned system, however, it is necessary to ensure that faulty assumptions about the temporal behavior of tasks belonging to one application cannot affect the temporal behavior of applications in different partitions.

There seem to be two ways to achieve this temporal partitioning: one is a two-level structure in which the kernel schedules *partitions*, with the application in each

[4] See `http://www.research.microsoft.com/research/os/mbj/Mars_Pathfinder/Authoritative_Account.html`.

partition then responsible for locally scheduling its own tasks; the other is a single-level structure in which the kernel schedules *tasks*, but with a quota system to limit the consequences of any faults—or faulty assumptions—to the partition that is in violation.

The first approach usually employs static scheduling at the partition level: the kernel guarantees service to each partition for specified durations at a specified frequency (e.g., 20 ms every 100 ms) and the partitions then schedule their tasks within their individual allocations in any way they choose; in particular, partitions may use dynamic scheduling for their own tasks. Any partition that schedules its tasks dynamically must provide a mechanism for interrupting one task in favor of another. Such support for task swapping is one of the reasons for preferring dynamic over static scheduling: it simplifies application programming by allowing long-running, low-frequency tasks to be interrupted by shorter high-frequency tasks, whereas statically scheduled systems have to break long-running tasks into separately scheduled fragments that perform their own saving and restoration of local state data to create room for the higher-frequency tasks. If partition swapping uses the restoration model, however, it provides an alternative mechanism for dealing with long-running tasks within a statically scheduled environment: a single application can be divided into parts that are allocated to separate partitions that are scheduled at different rates. The partition-swapping mechanism then takes care of interrupting and restoring the long-running tasks, thereby simplifying their construction.

Opportunities such as this make static scheduling for both partitions and tasks relatively attractive. Conversely, the constraints of static partition scheduling render its combination with dynamic task scheduling rather less attractive. One of the conveniences of dynamic scheduling is that it allows new tasks to be introduced—or the frequency and duration of existing tasks to be changed—with relative ease. But this ease is vitiated when partitions are statically scheduled because, for example, a new 10-Hz task can only be fitted into a partition that is already scheduled at this rate (or some multiple of it), so that the rigidity of the partition-scheduling mechanism dominates any flexibility in task scheduling.

This drawback could be overcome, however, if partitions could be scheduled at iteration rates very much higher than those of any task—say 1,000 times a second. Under the restoration model of partition swapping, a partition that is scheduled at such a rate and that is guaranteed, say, one tenth of the CPU (i.e., 100 μs every millisecond) could, for most purposes, be regarded as running continuously on a CPU that has one tenth the power of the real one, and its tasks could be dynamically scheduled without regard to the underlying partition schedule. Partition swaps are relatively expensive on traditional processors (because there is a large amount of state information that has to be saved and restored) and this renders kilohertz partition schedules infeasible on such hardware (all the resources of the system would be expended in swapping). However, specialized processors are under development where partition swapping is performed at the microcode and hardware levels, and these are believed to be capable of supporting partition schedules in the kilohertz range with no more than 5% to 10% of the system resources expended on swapping. Notice that the task swapping required for dynamic scheduling within each partition can be relatively lightweight (since tasks within a partition are not protected from each other) and will be activated at a frequency comparable to the fastest task iteration rate and not the much faster partition swapping going on beneath it.

The radical combination of a static partition schedule operating at kilohertz rates and dynamic task scheduling within each partition is an attractive one: it seems to provide both the convenience of dynamic scheduling and the predictability of static scheduling. However, one of the conveniences of dynamic scheduling is the ease with which it can accommodate aperiodic activities driven by external events, such as operator (e.g., pilot) inputs and device interrupts, and it requires care to support this on top of static partition scheduling—even when this is running at kilohertz rates. The basic concern is that external events of interest to one partition must not disturb the temporal behavior of other partitions. If partitions are scheduled dynamically, use of suitable quota schemes can allow temporal predictability to coexist with aperiodic event-driven task activations (this is discussed on page 28), but static partition scheduling ensures predictability through temporal determinism and this imposes strong restrictions on event-driven activations.

First and most obviously, a static partition schedule does not allow an external event to initiate a partition swap: the partition schedule is driven strictly by the processor's internal clock, so that if an event requires the services of a task in a partition other than the current one, it must wait until the next regularly scheduled activation of the partition concerned. This increases latency, but may not be a problem if partitions are scheduled at kilohertz rates. Less obvious, perhaps, are the consequences of the requirement that the currently executing partition should see no temporal impact from the arrival of events destined for other partitions. Even the cost of a kernel activation to latch an interrupt for delivery to a later partition reduces availability of the CPU to the current partition and must be strictly controlled. It is possible to add padding to the time allocated to each partition to allow for the cost of kernel activity used to latch some predicted number of interrupts for other partitions. But this makes temporal correctness of one partition dependent on the accuracy of information provided by others (i.e., the number and rate of their external events)—and even originally accurate information may become useless if a fault causes some device to generate interrupts constantly.

This concern is a manifestation of a more general issue: temporal partitioning requires not only that each partition has access to the resources of the system at guaranteed intervals, but that those resources provide their expected performance. A CPU whose performance is degraded by the cost of latching interrupts for later delivery is just one example; others include a memory subsystem degraded by DMA transfers on behalf of other partitions, or an I/O subsystem that is busy on their behalf.

Under static partition scheduling, temporal partitioning is predicated on determinism: because it is difficult to bound the behavior of faulty partitions, the availability and performance of each resource is ensured by guaranteeing that no other partition can initiate *any* activity that will compete with the partition scheduled to access the resource. This means that no CPU or memory cycles may be consumed other than at the behest of the software in the currently scheduled partition. Thus, in particular, there can be no servicing of device interrupts, nor cycle-stealing DMA transfers other than those initiated by the current partition. These requirements can be violated in two ways: a previously scheduled partition may have had some I/O activity pending when it was suspended, or the external environment may generate an interrupt spontaneously (e.g., to indicate that a button has been pressed).

Draconian measures seem necessary to prevent these sources of temporal uncertainty. External events either should not generate interrupts (the relevant partition should poll for the event instead), or it should be possible to defer handling them until the relevant partition is running (whether this is possible depends on the nature of the device and the interrupt, and on how selectively the CPU architecture allows interrupts to be masked off). Similarly, interrupts due to pending I/O from a device commanded by a previous partition should be masked off. If interrupts cannot be masked with sufficient selectivity, we could require the kernel to issue commands that quiet the devices of the previous partition as part of the process of suspending that partition and starting the next. Alternatively, if devices go quiet when uncommanded for some short time, the kernel could make the device registers unavailable (e.g., by changing the MMU table) during the final few milliseconds of each partition's schedule.

The restrictions just described as necessary to ensure that temporal correctness of tasks in one partition are unaffected by software in other partitions have consequences for inter-partition communications. With static scheduling of partitions, a task that needs the services of software in another partition (e.g., to access a shared device) cannot simply issue a procedure call. In fact, there can be no synchronous services (i.e., where the caller blocks and waits for the service provider to reply) across partitions because (a) one partition should not depend on another (that may be faulty) to unblock its progress, and (b) it would impose a large performance penalty: the caller would block at least until its next slot in the schedule after the service provider's slot. Instead, all inter-partition communication must be asynchronous (where the caller places requests in the input buffers of tasks in other partitions and continues execution; when next activated, it looks in its own input buffers for replies, requests, and unsolicited data from other partitions). Because faulty software could generate an excessive number of requests for service by another partition, it seems necessary that fixed quotas should be imposed on the number or rate of service requests that will be honored from each partition.

Some of the restrictions that are necessary when partitions are scheduled statically may possibly be relaxed when they are scheduled dynamically. It makes little sense to schedule partitions dynamically and tasks statically, and when both partitions and tasks are scheduled dynamically there is little point in maintaining two levels of scheduling, so the unit of scheduling will actually be the task. However, the concern for temporal partitioning will influence which tasks are eligible for execution. Whereas static scheduling ensures temporal partitioning through strict preplanned determinism, dynamic scheduling relies on theorems from the mathematical study of (for example) RMS. There are two problems in applying this theory in the context of partitioning: one is that a faulty partition may violate the assumptions underlying the theorem concerned; the other (related) problem is that the simplest (and therefore, for life-critical applications, preferred) theorems make the strongest assumptions (e.g., that context switches take no time), whereas those with more realistic assumptions rest on more elaborate and less well-established theory. Both problems can probably be overcome by having the kernel and its scheduler enforce quotas.

For example, if schedulability of a task set is predicated on a given partition taking no more than 20% of the available time in each cycle, then the kernel can simply refuse to make any of its tasks eligible for scheduling once that 20% quota has been reached. The problem with this simple scheme is that a faulty partition may consume its quota in very many small bursts (or a device may generate interrupts at a rapid rate). The many partition swaps entailed thereby may have a more deleterious effect on the tasks of other partitions than the CPU time directly consumed by the faulty task. A plausible way to overcome this problem is to subtract the cost of a partition swap (and the performance degradation caused by disturbing the caches) from the quota of the task that causes it. Quotas managed in this way provide many of the guarantees of static scheduling while retaining some of the flexibility of dynamic scheduling. For example, such a scheme could allow synchronous as well as asynchronous inter-partition communications, together with the ability to service aperiodic events and interrupts. (Modern operating systems such as Scout use a somewhat similar approach, in which accounting for resource usage is performed on abstractions called *paths* [102].) However, many of the restrictions and concerns discussed for static partition scheduling remain relevant for dynamic scheduling: for example, it still seems necessary to eliminate cycle-stealing DMA transfers and other performance-degrading activities that cannot easily be controlled by quotas, and it is also necessary to ensure that interrupts for a partition that has exceeded its quota are masked or latched at truly zero cost. Other potential sources of cross-partition interference such as locks and semaphores must also be suitably controlled (probably by elimination).

Quota-based dynamic scheduling may provide simple guarantees that the tasks of nonfaulty partitions receive their expected allocations (i.e., they receive *enough* time), but guarantees that they will hit their deadlines (i.e., they get it at the *right* time) are more problematical (there are, for example, scenarios under RMS where the *early* completion of one task causes another to miss its deadline [84]). In practice, relatively few tasks may need to be scheduled with great temporal precision: it is generally necessary to sample sensors and control actuators with very low jitter, but it does not greatly matter when the control laws are evaluated provided their results are ready when needed. Thus, we can envisage a scheme in which certain tasks (those associated with sensors and actuators) are guaranteed to execute with great temporal accuracy, while others are guaranteed only to get their allocation of resources sometime during their period. To achieve this, the sensor and actuator tasks could run in separate processors that are statically scheduled (and communicate with the dynamically scheduled computational tasks through dual-ported memory), or they could run at the highest priority in the dynamically scheduled system; justification for the latter scheme would require deeper theorems than the former.

Whether partitions and tasks are statically or dynamically scheduled, the kernel must collaborate with other software to provide some of the services of an operating system—at the very least it will be necessary to service interrupts. Under static partition scheduling, interrupts from external devices are allowed only when their partition is running; this means it is possible to vector interrupts directly to handlers in the partition, rather than handle them in the kernel. The advantage of the former arrangement is that it minimizes the complexity of the kernel; its difficulty is that interrupts are often vectored in supervisor mode, which can threaten hardware-mediated spatial partitioning. Compromise arrangements have the kernel fielding the hardware interrupt, but then passing it in a safe way to the partition for service. Arguments against device handling in a partition are that this really is an operating system service that is better done by an operating system. A conventional operating system is unattractive in a partitioned environment because, as portrayed in Figure 2.1(a) on page 7, it is a large shared resource that must be shown to respect partitioning as well as to be free of other faults. A more suitable arrangement provides operating system services separately within each partition, as portrayed previously in Figure 2.1(b). This arrangement has the additional benefit that different partitions can use different sets of operating system services: for example (see Figure 3.1 on page 30), a critical function might use a minimal set of services (Partition C), while a less critical but more complex function might employ something close to a COTS operating system (Partition B), and a device management partition might consist largely of standardized operating system services for device management. Operating system services cannot affect basic partitioning in this arrangement; however, they must be used with great circumspection in partitions that encapsulate a shared service or resource (e.g., a partition that provides a shared file system). Such partitions are logically an extension of the kernel and must be shown to partition their service or resource appropriately—which is likely to be more difficult the more software they contain.

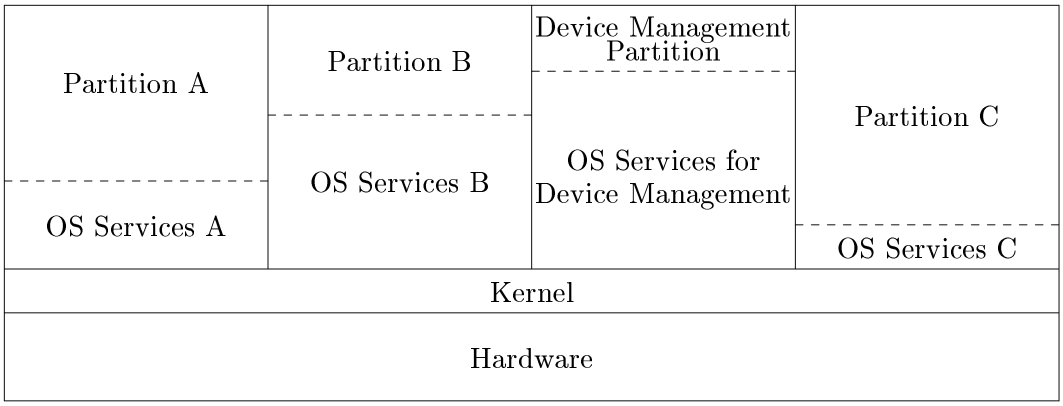

Figure 3.1: Different Operating System Software for Different Partitions

3.2 Partitioning Across a Distributed System

A distributed system resembles our original Gold Standard—a separate processor for each partition—more closely than a single shared processor, and might seem to raise few new issues with respect to partitioning: if we accept that the partitioning mechanisms employed within individual processors are sound, then connecting several such systems together surely cannot do any harm. This would be true if we could arrange dedicated physical point-to-point communications between *partitions* in different processors, but the only physical communications that can be provided are between *processors*. This limitation has a fairly significant impact, which is compounded when we consider shared communications, such as buses.

To start with, suppose we wish to communicate data from partition a_1 of processor A to partition b_1 in a different processor B, and that we have a suitable communications line from A to B. Interrupts will be generated at B as the data starts to arrive and, as we discovered in the previous section, some care is needed to ensure that these do not disturb temporal partitioning in B. If B is dynamically scheduled, the quota schemes discussed previously may be all that is needed, but matters can be more complicated when partitions are scheduled statically. Under static scheduling, we must require either that the interrupts can be latched at no cost until the scheduled execution of partition b_1 (partitions must be scheduled at high-frequency to make it feasible to service communications in this way), or that partition b_1 (or some device management partition that handles the communications line) is guaranteed to be executing when the interrupts arrive. The latter clearly requires synchronization between the partition schedules of processors A and B and, by extension to other processors, this implies global synchronization of schedules across all processors.

The only way to avoid these consequences when static partition scheduling is employed is to have a data concentrator device at B that buffers incoming data without imposing a load on the CPU or its buses. The partition b_1 can then retrieve incoming data from the data concentrator as part of its normally scheduled activity. A more aggressive design would allow the data concentrator to write incoming data directly into buffers associated with each partition using dual-ported RAM. Even these designs do not necessarily eliminate the need for global synchronization, however, because of the need to control "babbling idiot" failures in partitions and processors.

These are failures where a transmitter sends data constantly, possibly overwhelming its recipient, or denying service to other transmitters. One scenario would be a runaway in partition a_1 that causes it to transmit to b_1 throughout its scheduled execution. We need to be sure that this heavy load on the communications line from A does not affect the ability of the recipient (B or its data concentrator) to service its other lines. This requires either some kind of quota scheme at the recipient, or a global schedule that excludes simultaneous transmissions. A babbling partition can do so only during its scheduled execution, so a global schedule may be able to ensure that no two processors simultaneously schedule partitions that transmit to the same recipient. An alternative if a_1 does not drive the communications line directly, but instead sends data to a device management partition, is for the management partition to impose a quota on the quantity of data that it will accept from any one partition. A babbling processor is an even more serious problem than a babbling partition; either the recipient must be able to tolerate the fault, or it must be prevented at the transmitter—mechanisms to do this are discussed below in the context of bus communications.

The measures discussed above address temporal partitioning in inter-processor communications; we also need to consider spatial partitioning. The spatial dimension to partitioning requires mechanisms to ensure that partition a_1 of processor A can send data to partition b_1 in a different processor B only if that communication is authorized. No additional mechanisms are required to ensure this when a communication line is dedicated to a specific inter-partition channel; additional mechanisms are needed, however, when one line is shared among multiple receiving partitions. In this case, the address of the intended recipient must be indicated in each transmission. This can be done explicitly by including the address in the data transmitted, or implicitly through the time at which it is sent (the schedules of the sending and receiving processors must be coordinated in this case). A concern with explicit addresses is that a communications fault could transform a datum addressed to partition b_1 into one addressed to b_2. This is a fault-tolerance issue, and is generally handled by checksums or similar techniques to ensure the integrity of transmitted data. The related partitioning issue is the concern that a fault in the sending partition a_1 could cause it to address data directly to an unauthorized recipient b_2—this fault will not be detected by checksums, since it occurs outside their protection. The only certain way to contain this fault is to mediate the communication with some trusted entity that has independent knowledge of the authorized inter-partition communications. This can be performed either at the transmitter (e.g., if a device management partition is used to access the communications line) or at the receiver (e.g., in a data concentrator). A probabilistic method to contain the fault is to allocate partition addresses randomly from a very large space; the chance that a fault in a_1 will cause it to manufacture the legitimate address b_2 is then correspondingly small. In the case of implicit addresses, the concern is that by sending data at the wrong time, the transmitting partition will cause it to be received by an unintended recipient. Mediation is required to contain this fault, which is considered in more detail below, in the context of bus communications.

Some architectures allow the components of a distributed system to communicate without adding explicit addresses to name the intended recipient. In "publish-subscribe" architectures [81], for example, data is tagged with a description of its *content* (e.g., `air-data-samples`) and recipients "subscribe" to data carrying given tags. These issues of naming and binding were discussed earlier (page 18) in the context of individual processors, and similar considerations apply here, but with the added concern for fault tolerance with respect to communications faults.

Using separate communications lines to connect each pair of processors is expensive, so buses are generally used in practice. A bus is a departure from the Gold Standard—it is a resource shared by all processors and all partitions—and it is therefore crucial to provide partitioning so that a fault in one partition or processor cannot affect others. The faults of greatest concern with buses are those where a partition or processor either babbles or fails to follow the access protocol in some way, so that other partitions or processors are denied timely access to the bus.

A babbling or misbehaving partition cannot interfere with bus access by other partitions in its own processor (because a partition can access the bus only when it is scheduled), but it can interfere with access by other processors (by contending for the bus if this is mediated, or by sending transmissions that collide with those of other processors if it is not), and it may overwhelm its receivers. A babbling or misbehaving processor is even more disruptive than a babbling partition because it is not constrained by its own schedule and can monopolize the bus. Notice that processor faults such as this are partitioning—not fault-tolerance—issues, because their consequences would not be so serious if the buses were not shared. Dual or multiple buses can be used, in the hope that a babbler will confine itself to just one of them, but this cannot be guaranteed. The only certain way to prevent babbling is to mediate each processor's access to the bus by some component that will fail independently. The question then is how does the mediator know what is a legitimate transmission and what is babbling? The answer depends on whether communications are *time* or *event triggered*.

In a time-triggered system, transmissions are determined by a schedule, and the mediating component need only have an independent copy of its processor's schedule and an independent clock in order to determine whether its processor should be allowed to transmit on the bus. The schedules that govern time-triggered transmissions can be either local or global. A local schedule treats each processor independently, so that different processors may contend for the bus and the receiving partition need not be scheduled at the same time as the transmitter. A global schedule, on the other hand, coordinates *all* processor and bus activity, so that there is no bus contention. Although it is perfectly feasible to use global scheduling with contention buses such as Ethernet or CAN (global synchronization means that their ability to resolve contention will never be exercised, but the system benefits from the low cost and high performance of the network interface hardware), some specialized buses have been developed specifically to support and exploit static global scheduling. Examples include the ARINC 659 SAFEbus™ [2,41] and the Time Triggered Protocol and its associated Architecture (TTP/TTA) [57]. With global scheduling, there is no real need to include a destination address with the data (because this is implicit in the time the message is sent) and some globally scheduled buses (e.g., ARINC 659) do eliminate explicit addresses, thereby reducing the number of bits that need to be communicated and increasing the useful capacity of the bus.

The clock of a bus mediation component needs to be independent of that of its processor, but synchronized with it. With local scheduling, the purpose of the mediating component is to control the pacing of bus accesses, but not their absolute timing and for this purpose it is adequate for the mediator and its processor to synchronize locally (obviously, this must be done carefully to maintain plausibility of the independent failure assumption). With global scheduling, however, the clocks of all processors and mediators must be globally synchronized, and the mediating components should perform the synchronization independently of their processors. If clock synchronization is achieved by a high-level protocol, then the mediating components must be capable of interpreting the full protocol hierarchy, and this greatly complicates their design. For this reason, the mediating components in TTA (called *bus guardians*) do not perform independent clock synchronization, but take synchronizing signals from their host processors [103]. This design prevents babbling, but a processor that loses clock synchronization will take its bus guardian with it and will still be able to access the bus at the wrong time, though only for short periods. However, the unsynchronized processor/guardian pair will also be unable to receive messages correctly (because synchronization is required to satisfy the CRC checks on each message), and the guardian will shut off all bus access after failing to receive a set number of expected messages. An alternative approach performs clock synchronization as a low-level protocol that can be performed by simple mediating components. This approach seems to require suitable electrical properties of the bus and its drivers. In SAFEbus, for example, the signals from separate drivers are OR'ed together on the bus, and this allows a very simple synchronization protocol that is performed directly in the mediating components (they are called Bus Interface Units in SAFEbus) [2].

Whereas globally scheduled systems guarantee that the bus will be free when a processor is scheduled to transmit, locally scheduled and event-triggered systems must cope with contention between processors attempting to transmit on the bus. In buses intended for control applications, contention is not resolved probabilistically following collisions as it is in classic Ethernet, but deterministically using preassigned slots (as in Echelon's LON), a circulating token (as in PROFIBUS [Process Field Bus] [23]), or a priority arbitration scheme (as in CAN [Controller Area Network] [46]) to provide distributed mutual exclusion and thereby prevent collisions. This determinism does not provide very strong guarantees on how long a processor must wait to access the bus, however. In CAN, for example, a processor that wishes to transmit must first wait for any current transmission to finish and then it must contend with any other processors that also wish to transmit. In CAN, the lowest-numbered processor always wins the arbitration and may therefore have to wait only as long as the longest message transmission, while other processors also have to wait while any lower-numbered processors perform their transmissions.[5] It follows that only probabilistic guarantees can be given on the bus-access delay in such systems, and that these guarantees will be quite weak in the presence of faults [105], even if bus access is mediated to control the worst manifestations of babbling.

It is not straightforward to mediate a processor's access to the bus when that access is event triggered—that is to say, triggered by the processor's internal computations, possibly based on data it has received—for there is no way to know whether an event has legitimately occurred without independently copying the data received and reproducing the computation performed by that processor. A master-checker dual-processor arrangement such as this is a very expensive way to prevent babbling. Redundant processors are obviously required for fault tolerance in IMA, but such

[5]The Echelon LON protocol has similar characteristics: stochastic flow control is used to reduce the likelihood of collisions; if a collision does occur, processors back off and access the bus in order of their "contention slots." The main application of the LON protocol is in automating buildings, where tight real-time guarantees are unlikely to be required, but the Echelon web site `http://www.lonworks.echelon.com` reports that Raytheon uses this technology in its Control-By-Light$^{\mathrm{TM}}$ fault-tolerant fiber optic distributed control system, which is currently undergoing FAA Part 25 certification for use in commercial aircraft; however, it seems that mechanisms in addition to the LON protocol are employed in this application.

redundancy should be managed flexibly at the system level, not committed to pairing. Without master-checker pairs, the best that can be done to control babbling in an event-triggered system seems to be the imposition of some limit on the rate at which a processor may transmit on the bus. The ARINC 629 avionics data bus [3] has this capability (the bus uses time slots to control access, but it can be used in the larger context of an event-triggered system).

Because the purpose of partitioning is to control fault propagation, some aspects of partitioning are very close to fault tolerance—for example, the control of babbling discussed in the previous paragraphs has elements of both. Mechanisms such as these are needed to preserve the integrity of the service provided by an IMA architecture to the avionics functions that it supports. In addition, the avionics functions often need to be fault tolerant themselves, and an IMA architecture must therefore support the development of such fault-tolerant applications. There is a choice in how much fault tolerance should be provided by the IMA architecture, and how much by the functions themselves. Faults such as babbling, which are outside the control of any single function and that can have system-wide ramifications, must clearly be tolerated by mechanisms of the IMA architecture. Sensor failure, on the other hand, seems more naturally the responsibility of the function that uses the sensor, while failure of a processor seems to fall somewhere in between—the designers of the function may best know how to handle such a fault, but may need services provided by the IMA architecture to implement their strategy.

As mentioned in Section 2.1 (page 8), the trend toward IMA runs in parallel with another trend toward developing avionics functions on top of a layer that provides standard operating system services and, possibly, additional services to support systematic fault tolerance. Fault tolerance in critical systems is usually based on active redundancy; errors are detected or masked through comparison or voting of the redundantly computed values. Fault tolerant architectures differ in whether the redundant replicas perform the same or different computations, and in whether their states are synchronized (to allow exact-match voting). Some of the architectural choices for fault tolerance are strongly contingent on other choices—for example, that between time- and event-triggered architectures—that are themselves strongly tied to choices in partitioning mechanisms. Kopetz presents persuasive arguments that time-triggered architectures are the best choice for critical real-time applications [53–55] and this choice also fits well with the requirements and mechanisms, discussed in the previous section, for ensuring temporal partitioning in a distributed system.

3.3 Summary

The topics examined in this chapter show that partitioning interacts rather strongly with several other issues in system design: for example, scheduling, communication, distribution, and fault tolerance. By "interacts with" I mean that design freedom in these dimensions is curtailed when partitioning is a primary system goal. This is not necessarily a bad thing, however, because the restrictions imposed by partitioning are exactly those that prevent unexpected interactions among system components, thereby promoting compositionality (i.e., the property that components that work on their own continue to do so when combined with other components) and reducing integration costs.

Because partitioning is critical to the safe deployment of IMA, the design and implementation of its mechanisms must be assured to very high standards. Guidelines for the assurance and certification of safety-critical airborne software are specified in the document known as DO-178B in the USA and ED-12B in Europe [83]. These guidelines call for a very rigorous—if traditional—process of reviews, analysis, and documentation; however, an appendix includes *formal methods* among the "alternative methods" that "may be used in satisfying one or more of the objectives" described in the document. The idea behind formal methods is to construct a mathematical model of a software or system design so that calculations based on the model can be used to predict properties of the actual system—in much the way that finite element analysis of a structural model for an airplane wing can be used to predict mechanical properties of the actual wing. Because the appropriate mathematical domain for modeling software is mathematical logic, where "calculation" is performed by so-called "formal deduction" (as opposed to, say aerodynamics, where the appropriate mathematical domain is partial differential equations, and calculation is performed by numerical methods), this approach is referred to as use of "formal methods."

The utility of calculation—as an adjunct to, or replacement for, physical experimentation—is well understood in other branches of engineering, and is similar in computer science. In fact, its utility is potentially greater in computer science than in other engineering disciplines because computer science deals with discrete or discontinuous phenomena, where experimentation and testing are of limited value as assurance methods. With discontinuous systems, there may be little relationship between the behavior of the system in one circumstance and its behavior in another "similar" circumstance; consequently, extrapolation from tested to untested cases is of doubtful validity. This contrasts with physical systems, where continuity justifies safe extrapolation from limited test cases. Formal methods augment testing by allowing *all* the behaviors of a system to be examined. Formal methods consider a *model* of the system, whereas testing examines the real thing, so the two approaches complement each other. An elementary description of formal methods, and their application to the certification of avionics is presented in [91], with more detail available in [90].

In addition to their role in assurance, the models constructed in formal methods can often help clarify requirements and design choices, and can lead to improved understanding of design problems. They do this by abstracting away all detail considered irrelevant to the problem at hand, and by formulating the remaining issues with mathematical precision. Formal models for partitioning could therefore help refine our understanding of this topic. Now, partitioning has much in common with certain issues in computer security, and those issues have been the target of considerable research in formal modeling extending over more than two decades. The next chapter, therefore, examines issues in computer security related to partitioning, and outlines the formal modeling techniques that have been tried.

Chapter 4

Comparison With Computer Security

Computer security is related to partitioning in that both are concerned with the ability of one software application to influence another. The concerns are that sensitive information might "leak" from one partition to another (this is called *information flow* in the security context), or that doubtful information might contaminate high-quality information (this is called information *integrity* in the security context), or that one partition might monopolize or reduce timely access to the CPU or some other resource (this is called *denial of service* in the security context). Much work over many years (see, for example, a survey published in 1981 [60]) has sought to provide a firm understanding of these security issues and their enforcement mechanisms, and we might hope to apply some of this work—or at least the underlying ideas—to partitioning. In addition, research in computer security has sought to provide rigorous, formal approaches to the specification and verification of secure systems, and there is hope that these approaches could contribute to the development of strong assurance techniques for partitioning in avionics. The following sections review these security issues and the formal modeling techniques that have been applied to them. The goal here is to explain the basic ideas and approaches, so we merely describe the formal techniques that have been used rather than present the actual formalism.

4.1 Data and Information Flow

The most studied aspect of computer security is something of a dual to one of the concerns of spatial partitioning. In spatial partitioning, a concern is that one partition might write data into a second, and thereby disrupt its operation. In security we are more concerned with the data that is written: if data in the first partition is considered highly classified, then writing it into a more lowly classified partition is tantamount to disclosing it. Reflecting this concern, computer security generally uses the more neutral term *process* for what was called a partition in the previous chapter (indeed, the computer security notion that is closest to partitioning is called "process security" [9]). *Data flow* security is concerned with controlling channels for disclosure; *information flow* security is concerned with more subtle channels in which data is not written directly, but its information content is disclosed just as effectively.

4.1.1 Access Control

A basic mechanism in enforcing both partitioning and information flow security is called *access control*: the computer system is assumed to have some means (typically, supervisor/user mode distinctions and memory management hardware) for limiting the primitive resources that a process can access, and the ways in which it can access them. Then some higher-level resources are synthesized (e.g., a file system), and rules governing access to those resources are defined and implemented in terms of the more primitive resources and protections. The rules constitute an *access control policy*. A familiar example is that of the Unix file system: each file is associated with a particular owner and group, and we can specify separately whether the owner, members of the group, or other users can read, write, or execute the file. This example raises two important topics in access control: the first concerns the choice and specification of the access control policy that is to be enforced, and the second concerns the completeness of that enforcement.

The Unix file system provides a *discretionary* access control policy: users who have read access to a file can, at their discretion, copy it and grant access to the copy in any way they choose. This may be contrary to the intent of the original owner, or to some organizational policy. To deal with these concerns, various more constrained kinds of *mandatory* access control policies have been defined. The simplest example is the *multilevel* security policy that is intended to reflect practices for handling classified military information. In a multilevel policy, every resource and every process (computer security uses the terms *object* and *subject* for these) is given a label from some ordered set (typically UNCLASSIFIED, CONFIDENTIAL, SECRET, and TOP SECRET), and a subject may have read access to an object only if the subject's label (its *clearance*) is equal or to greater than that of the object (its *classification*).[1] This rule (it is called the *simple security property*) does not stop a subject creating a copy of an object at a lower classification and thereby violating the intent of the policy, so it is augmented by another rule called the $*$-*property* (pronounced "star property") that says that a subject may have write access to an

[1] Matters are complicated in practice by the use of compartments (e.g., NATO, NOFORN) in combination with the basic classifications to create a partial ordering.

object only if the object's label is equal to or greater than that of the subject. The combination of the simple and the $*$ properties (i.e., a subject can only read "down" and write "up" in security level) constitute the historically significant Bell and La Padula security policy [10]. Under further examination, this policy raises important questions that will be considered shortly. First, though, we return to the related question of completeness of an access control policy.

The access control policy of the Unix file system can be bypassed if users can directly read or write the contents of the disk on which those files are stored. Thus, although our interest is in protecting files, we also need to be concerned about the disk, and possibly other elements of the system as well. So the issue of completeness in access control concerns how much of the system needs to be placed under access control, and in what way, for us to be sure that the resource we actually want to protect is, indeed, protected against all possible attacks. This issue is complicated by the fact that security is really about protecting *information*, not mere data, so that any channel (a metaphorical example would be by tapping on the walls in Morse code) that allows the information content of a file to be conveyed to an unauthorized user is as dangerous as the ability to copy a file directly.

The possible channels for information flow can be quite subtle and hard to detect (there were at least two in Bell and La Padula's "Multics Interpretation" [10]). For example, suppose we had a special Unix system that imposed the Bell and La Padula policy on file access, but with the additional property that file names are required to be unique across all users: an attempt to create a file with an existing name returns an error code. Then, a SECRET process can convey information to an UNCLASSIFIED one by creating files with prearranged names: the UNCLASSIFIED process retrieves the information by checking whether or not it is able to create files with those names. This is an example of a "covert" channel; more particularly, it is a covert *storage* channel (because it exploits information stored in the directory structure of the file system; the other kind of channel uses *timing* information—see Section 4.3) [59, 64]. The channel is noisy (some other, innocent, process might have created files with those names), but coding techniques allow information to be transmitted reliably over noisy channels. Covert channels are of concern for two reasons: first, they can be used to transmit information at surprisingly high bandwidth (one early demonstration drove a teletype at full speed using a channel that depended on sensing where a disk head was positioned [95]) and second, they are no different in concept from more blatant channels (e.g., the unprotected disk) that leave a resource open to direct access (both are symptoms of incompleteness)—so that unless we have methods of specification and verification that are able to eliminate subtle covert channels, we have little guarantee that we can eliminate any channels at all.

It might seem that information flow and covert channels are esoteric security concerns and that only basic access control is relevant to partitioning. However, while it is true that covert information flow may be of little concern for partitioning (because it depends on collusion between sender and receiver and is therefore implausible as a fault manifestation), the *mechanisms* used for such flow definitely are of concern. Consider, for example, the unique-file-name channel described above. This serves as a channel for information flow because one subject can affect the behavior perceived by another (i.e., whether or not the attempt to create a file returns an error), and this is surely contrary to the expectations of partitioning—for one interpretation of those expectations is that the behavior perceived by software in any given partition should be independent of the actions by software in other partitions. We might try to arrange for this expectation to be satisfied in the presence of the unique-file-name restriction by allocating disjoint name spaces to each partition. But then a fault in the software of one partition could cause it to create a file from another's name space—and thereby cause a subsequent file creation in that other partition to fail. This example shows that covert channels for information flow raise issues that are relevant to partitioning, and that examination of how security has dealt with these channels may be of use in partitioning.

Another potential problem with access control formulations of security is that they depend on informal understandings of what "read" and "write" accesses really mean. We can construct perverse systems in which these terms are given incorrect (e.g., reversed) interpretations and that satisfy the letter of an access control policy while violating its spirit [71].

Covert channels and perverse interpretations are both symptoms of the real problem with access control as we have used it: it is a mechanism for *implementing*, not an instrument for *specifying*, security policies. An adequate specification should get at the "intent" that underlies a security policy in a convincing manner. It should then be possible to prove that an implementation in terms of access control correctly enforces the policy. Problems of completeness, covert channels, and perverse interpretations should all be eliminated by a sound approach of this kind. The next section examines such approaches.

4.1.2 Noninterference

To repair the problems with access control, we need to be more explicit about our system model: we need to specify how a system computes and interacts with its environment, how inputs and outputs are observed, and how subjects and objects are identified. Then we can specify security in terms of constraints on the observable behavior of the system, without needing to describe mechanisms to enforce those constraints (although we would hope to be able to describe such mechanisms at a later stage of development, and to verify that they enforce the desired policy).

The most successful treatments of this kind are all variations on a formulation called *noninterference* that was introduced by Goguen and Meseguer in 1992 [33], although the key idea was adumbrated five years earlier [30]. That key idea is that if there is no flow of information from one security classification to another, then the behavior perceived by subjects of the second ("lower") classification should be independent of any activity by subjects of the first ("higher") classification. In particular, the behavior perceived by the second classification should be unchanged if all activity by the first is removed. The precise details depend on the formal model of computation employed, but the traditional treatment uses a finite automaton as the system model: the automaton changes state and produces an output in response to inputs, which are labeled with their security level. A relation $p \rightsquigarrow q$ indicates whether level p is allowed to convey information to or *interfere* with level q; its negation is the *noninterference* relation $\not\rightsquigarrow$, which is considered a specification of the desired *security policy.* A sequence of inputs α is *purged* for level p by removing all inputs from levels that may not interfere with p; this purged input sequence is denoted α/p. Starting from some initial state s_0, the state of the automaton after consuming the input sequence α is $run(s_0, \alpha)$, while that after consuming the purged sequence is $run(s_0, \alpha/p)$. The noninterference formulation of security then requires that any level p input must produce the same output in both these states. The intuition is that this ensures that no experiment conducted at level p can reveal anything about the presence or absence of earlier inputs from levels that should not interfere with p.

The noninterference formulation of security is stated in terms of a system's behavior in response to a *sequence* of inputs. An *unwinding theorem* reduces this to three conditions on its behavior with respect to *individual* inputs. These conditions are stated in terms of each level's "view" of the system state (intuitively, if the system state is thought of as consisting of different components "belonging" to each level, then level p's view of the state comprises its own component and the components of all the levels that are allowed to interfere with p). If the level p views of two states are the same, we say these states "look the same to p" (technically, this is an equivalence relation on states).

Output Consistency: if two states look the same to p, then a level p input must produce the same output in both states.

Step Consistency: if two states look the same to p, then the states that result from applying the same input (of any level) to both states must also look the same to p.

Local Respect (for $\rightsquigarrow$): the system state must look the same to p before and after an input of a level that is noninterfering with p.

It is straightforward to prove that these conditions are sufficient to imply noninterference. The proof is formalized and mechanically verified in one of the tutorials for the PVS verification system [94].

A connection between the noninterference and access control notions of security can be established by interpreting the unwinding conditions in access control terms. We suppose that the system state is a function from *objects* to *values* and that each object has a level. Inputs of level p are reinterpreted as *actions* performed by a *subject* of level p. Then we suppose that access control enforces the following *Reference Monitor Assumptions.*

- The output produced by an action depends only on the values of objects to which the subject performing the action has read access.
- If an action changes the value of any object, then its new value depends only on the values of objects to which the performing subject has read access.
- An action may change the values only of objects to which the performing subject has write access.

With these assumptions, access control can enforce the unwinding conditions by setting up the controls as follows (these are essentially the Bell and La Padula conditions).

1. If $p \leadsto q$, then the objects to which subjects of level p have read access must be a subset of those to which subjects of q have read access, and
2. A subject of level p may have write access to an object for which a subject of level q has read access only if $p \leadsto q$.

The connection between the two formulations is established by interpreting a subject's "view" as the values of all the objects to which it has read access. A proof is given in [89, Section 2.1]. The proof requires formalizing the reference monitor assumptions, which is surprisingly difficult to do correctly (Popek and Farber [82], who first recognized the importance of these conditions, made errors in formalizing them).

Contrary to early expectations (e.g., [34]), standard noninterference requires the interferes relation $\leadsto$ to be transitive [89]. All such transitive relations are equivalent to multilevel security policies, and the two conditions on access control enumerated in the previous paragraph are likewise equivalent to the Bell and La Padula properties in these cases [89, Section 3.1].

Because they imply a partial ordering on security levels, multilevel security policies do not seem to capture the concerns of partitioning all that closely, but *intransitive* policies (that is, those where $\leadsto$ is not required to be transitive) seem more promising. Intransitive policies capture the additional security restrictions known as *channel control* [87] or *type enforcement* [13], which are concerned not only with whether information may flow from one place to another, but with the paths through which it may flow. Channel control security policies can be represented by directed graphs, where nodes represent security domains and edges indicate the direct information flows that are allowed. The paradigmatic example of a channel-control problem is a controller for end-to-end encryption, as portrayed in Figure 4.1.

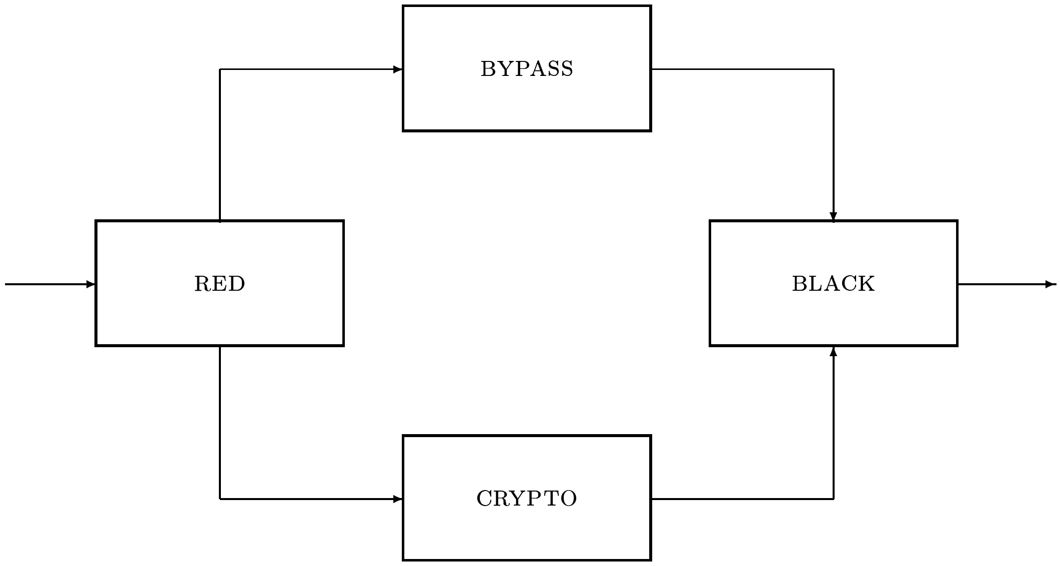

Figure 4.1: Allowed Information Flows for an Encryption Controller

Plaintext messages arrive at the RED side of the controller; their bodies are sent through the encryption device (CRYPTO); their headers, which must remain in plaintext so that network switches can interpret them, are sent through the BYPASS. Headers and encrypted bodies are reassembled in the BLACK side and sent out onto the network. The security policy we would like to specify here is the requirement that the *only* channels for information flow from RED to BLACK must be those through the CRYPTO and the BYPASS (it is a separate problem to specify what those components must do). Notice that the edges indicating allowed information flows in this example are not transitive: information is allowed to flow from RED to BLACK via the CRYPTO and BYPASS, but must not do so directly.

Noninterference can be extended to intransitive policies by substituting a more complicated *purge* function for the standard one. When $p \not\rightsquigarrow q$, the usual requirement is that deleting all actions performed by p should produce no change in the behavior of the system as perceived by q. This is too strong if we also have the assertions $p \rightsquigarrow r$ and $r \rightsquigarrow q$. Surely we should only delete those actions of p that are not followed by actions of r (in the CRYPTO example, RED, BLACK, and BYPASS or CRYPTO take the roles of p, q, r, respectively). This insight, and a definition of the generalized *purge* function, were given by Haigh and Young [37], together with corresponding unwinding conditions. Unfortunately, one of their unwinding conditions is incorrect; correct conditions were given, and formally verified, by Rushby [89]. These unwinding conditions simply replace the step consistency condition by a weak form.

Weak Step Consistency: if two states look the same to p, and also look the same to q, then the states that result from applying the same input of level q to both states must also look the same to p.

The corresponding conditions for access control enforcement consist simply of the second of the two conditions given on page 44.[2]

The formal statements of standard and intransitive noninterference use an automaton as their formal system model and therefore apply straightforwardly only to a single monolithic system. To examine the interactions of multiple, distributed systems, more general models are required—for example, transition relations or process algebras—and it is necessary to admit nondeterminism. Nondeterminism arises naturally in concurrent systems, because there is generally no system-wide coordination of the rate at which different components proceed; hence, interactions can occur in different orders in otherwise identical runs, and the behaviors perceived in those runs can diverge markedly (this is why it is so hard to debug concurrent systems). To accommodate this system-level nondeterminism, noninterference for concurrent systems is formulated to require that the *set* of behaviors possible in a given scenario is unchanged at a given level when interactions are purged in some suitable way. One problem with this formulation is that it does not define a *property* in the technical sense.

A system can be identified with the set of runs that it can produce (a run is a sequence or "trace" of inputs, outputs, and other significant interactions). A specification is likewise a set of runs, and a system satisfies a specification if its runs are a subset of those of the specification. A set of runs is called a *property*, so that specifications and systems can both be considered properties. Special classes of properties called *safety* and *liveness* play an important role in formal methods of analysis, and it can be shown that every property can be expressed as the conjunction of a safety and a liveness property [5]. Security, however, is not a property in this sense: it is not a set of runs, but a set of sets of runs [72]. This means that standard methods for deriving or verifying an implementation that satisfies a given specification do not work for security—because these methods apply only to properties.

Another problem when noninterference is extended to concurrent systems in the manner just described is that it is not *compositional*: that is, two systems individually satisfying some noninterference policy can be combined to yield a composite

[2]This might seem to suggest that the first condition on 44 is implied by the second when the policy is transitive. In fact, this is not necessarily so for a given set of access controls, but it will be possible to construct another set (i.e., a different assignment of read and write permissions) that will satisfy both conditions. This is a consequence of the "nesting property" for transitive policies [89, Theorem 5].

system that does not satisfy that policy [70]. Many alternative formulations of noninterference were proposed for concurrent systems in the attempt to overcome this unattractive result. Unfortunately, those that were compositional were either very unintuitive (having no plausible interpretation as a natural security concern), or were excessively restrictive (and unlikely to be satisfied by practical systems). A partial resolution was provided by Roscoe, who suggested that the difficulty was due to a failure to appreciate the significance of nondeterminism when contemplating security [85].

The problem with nondeterminism is that it can sometimes be resolved in a way that depends on unsecure information flow. A typical example would be a system with two levels, LOW and HIGH where HIGH is required to be noninterfering with LOW. Inputs to LOW cause the outputs *odd* or *even* to be generated nondeterministically *unless* there have been any high inputs, in which case the LOW output is *odd* or *even* according to the oddness or evenness of the last HIGH input (the HIGH inputs are assumed to be positive integers). This example satisfies most definitions of noninterference for concurrent systems because the set of *possible* behaviors observable at the LOW level is unchanged by the presence or absence of HIGH-level activity—yet it plainly violates any reasonable interpretation of "secure system." The violation is exposed when the system is composed with one that generates only even numbers on the HIGH input. Roscoe excluded such paradoxical constructions by requiring their component systems to have behavior that is deterministic at each security level. Roscoe's insistence on determinism also suggests a resolution to another difficulty that had plagued most earlier treatments: noninterference is not preserved under refinement. Refinement in this (process algebra) context means a reduction in nondeterminism, and it poses the same challenge to noninterference as composition. Roscoe's treatment is couched in the formalism of CSP [39], where a process is deterministic if it is free of "divergence" and never has a choice between "accepting" and "refusing" an event [86]. The relationship between this treatment and traditional interpretations of determinism and security in state machines is one that requires clarification.

There is another sense of refinement for which security in general (not only its noninterference formulations) is not preserved. This is the notion of refinement in the sense of elaboration, where more mechanisms and details are added to a specification in order to obtain an implementation that is effectively executable. Under the standard notion of correctness for such refinements, it is necessary only to show that the properties of the specification are implied by those of the implementation: the implementation is required to do *at least* as much as the specification, but it is not prohibited from doing *more*. An implementation of the specification suggested by Figure 4.1, for example, must provide at least the four communications channels shown, but the standard notion of correct refinement would not prevent it adding a direct communications channel between RED and BLACK—despite the fact that the absence of such a channel is the whole point of the design. For security, it is necessary to constrain the notion of correct refinement so that the implementation does not add capabilities that are absent in the specification. Clearly the implementation must contain more details and mechanisms than the specification (else it is surely not an implementation), but for secure refinement these mechanisms and details must have no consequences on the behavior that can be perceived at the originally specified interfaces. The formal characterization of this requirement is given in terms of *faithful interpretations* and is due to Moriconi, Qian, Riemenschneider, and Gong [74].

4.1.3 Separability

Using Roscoe's perspective, an adequate treatment for distributed channel-control security might be achieved by taking the nondeterministic composition of deterministic systems, each characterized by intransitive noninterference. Some architectural refinement to a more detailed implementation level could be obtained using faithful interpretations, and the restrictions within each system could then be enforced by access control, using the derivation from the unwinding conditions described earlier. (As far as I know, nobody has determined whether the formal details of the various models support this combination, nor whether satisfactory properties can be derived for the combination, but it seems plausible.) However, the resulting model would still be rather abstract for the purpose of deriving, for example, conditions on how a single processor should implement the RED, BYPASS, and BLACK components of Figure 4.1 (the CRYPTO is usually an external device).

An approach called *separability* was proposed for this problem by Rushby [87]. The idea is easiest to understand when no communications are allowed between the separate components. Then the idea is that the implementation should provide the appearance of a separate, dedicated processor to each component. The real processor is time shared, so that it sometimes performs instructions on behalf of one component and sometimes on behalf of another. The requirements for separability can be expressed in terms of *abstraction functions* that give the "view" of the processor perceived by each component. For example, if we have just two components, RED and BLACK, and Φ_R and Φ_B denote their respective abstraction functions, then the requirement when the processor is executing instructions on behalf of RED is that the following diagram should commute.

$$\begin{tikzcd}
{} \arrow[r, "op_R"] & {} \\
{} \arrow[u, "\Phi_R"] \arrow[r, "op"'] & {} \arrow[u, "\Phi_R"']
\end{tikzcd}$$

That is to say, the state change in the physical processor caused by executing the instruction *op* should be consistent with execution of the "abstract" operation op_R on RED's view of the processor. At the same time, BLACK's view of the processor should be unchanged, as expressed by the following diagram.

$$\begin{tikzcd} & {} & \\ {} \arrow[ur, "\Phi_B"] \arrow[rr, "op"'] & & {} \arrow[ul, "\Phi_B"'] \end{tikzcd}$$

Because I/O devices can directly observe and change aspects of the real processor's internal state (by reading and writing its device registers, for example), and can also influence its instruction sequencing mechanism (by raising interrupts), the activity of these devices is relevant to security. Consequently, we must impose conditions on their behavior. Expressed informally (and only from the RED component's point of view), these conditions are the following.

1. If $\Phi_R(\sigma) = \Phi_R(\tau)$ and activity by a RED I/O device changes the state of the real processor from σ to σ', and the same activity would also change it from τ to τ', then $\Phi_R(\sigma') = \Phi_R(\tau')$ (i.e., state changes in the RED view caused by RED I/O activity must depend only on the activity itself and the previous state of the RED view).

2. If activity by a non-RED I/O device changes the state of the real processor from σ to τ, then $\Phi_R(\sigma) = \Phi_R(\tau)$ (i.e., non-RED I/O devices cannot change the state of the RED view).

3. If $\Phi_R(\sigma) = \Phi_R(\tau)$, then any outputs produced by RED I/O devices must be the same in both cases.

4. If $\Phi_R(\sigma) = \Phi_R(\tau)$, then the next operation executed on behalf of the RED component must also be the same in both cases.

Separability was proposed before formal treatments of concurrent systems had been fully developed, so the justification of the above conditions presented in [88] is not fully satisfactory. Furthermore, neither the informal nor the formal presentation deals with allowed communications channels between components. The proposal in [87] is to remove the mechanisms intended to provide the desired communications channels and then verify, using the conditions above, that the components of the resulting system are isolated. Jacob [47] noted that this does not exclude a particular kind of covert channel (called a "legitimate" channel) that piggybacks undesired clandestine communication on the desired channel.

A more modern treatment [92] derives the conditions for separability with communications from those for intransitive noninterference. This treatment weakens the "triangular" commutative diagram of strict separability so that it applies only if RED $\not\rightsquigarrow$ BLACK (this derives from the "local respect for $\not\rightsquigarrow$" unwinding condition) and, when RED $\rightsquigarrow$ BLACK, it replaces the "rectangular" diagram by the following condition (which is based on the "weak step consistency" unwinding condition).

$$\Phi_R(\sigma) = \Phi_R(\tau) \wedge \Phi_B(\sigma) = \Phi_B(\tau) \supset \Phi_B(op(\sigma)) = \Phi_B(op(\tau))$$

Notice that this last condition does not use the abstract operation op_R that appears in the "rectangular" commutative diagram. This is because we do not really care what this operation is, only that $\Phi_B(op(\sigma))$ should be functional in $\Phi_B(\sigma)$, and the formula above expresses this directly.

4.2 Integrity Policies

The previous sections have considered computer security notions related to the undesired *disclosure* of information. There are similar notions related to the *modification* of information, where the main concern is to ensure *integrity* of the protected information. Integrity is related to the "reliability" or "quality" of information: information of high integrity should not be allowed to become contaminated by information of low integrity. This requirement can be treated as a strict dual to the Bell and La Padula security policy (that is, a subject can only read "up" and write "down" in integrity level), and is known as the Biba integrity policy [12].

Clark and Wilson [15] argued that the integrity of information is also a function of the operations that are performed on it, and the identity of those who invoke those operations. A user should not be able to invoke arbitrary operations on high-integrity information, but only certain *well-formed transactions*, and the admissible transactions might be determined by the state of the data, the identity of the user, and other factors. In commercial environments, the transactions available to a user are often governed by requirements for *separation of duties*: a user who authorizes a purchase should not be the same as the one who selects the vendor.

Other similar models for integrity have been proposed, and there has been considerable investigation of whether these or the Clark-Wilson model can be enforced by adaptations of security mechanisms developed to control disclosure [78].

4.3 Timing Channels and Denial of Service

Most work on formalizing security has focused on the data and information flow issues described in the previous sections. In partitioning terms, these all concern issues in spatial partitioning. There are, however, two topics in computer security that correspond to issues in temporal partitioning: timing channels and denial of service.

Timing channels (they were called "covert channels" when first identified [59]) are mechanisms for clandestine information flow that work by modulating the time when some events occur, or the rate at which they occur. For example, a process can choose whether or not to give up its time slice early. If only two processes are running, the other process can use the time at which it receives control to infer the choice made by the other process [59]. More generally, the decisions of a real-time scheduler can be manipulated to provide a channel for information flow [16]. Other timing channels modulate the load or contention on some system resource (e.g., the system bus [42]) or parameters affecting performance (e.g., the time to seek a disk track is affected by whether the previous seek was to a nearby or distant track [49]; the time to access a memory page will be affected by whether or not it was previously swapped out to disk [95]).

Where they cannot be removed, timing channels are typically rendered harmless either by reducing contention, or by introducing randomness into the behavior of the resource being manipulated [36, 67, 104], or by reducing the precision of the various "clocks" (e.g., time-of-day clocks, timers, instruction loops, asynchronous I/O performance) by which a process can measure the passage of time [42]. These measures do not block a timing channel, but they introduce sufficient noise that its bandwidth is reduced to acceptable levels (typically less than 10 bits per second).

Whereas the concerns of partitioning and security are quite close in the case of storage channels, they diverge for timing channels. The very existence of a timing channel is unacceptable in a partitioned system, since it indicates that one partition can change the temporal behavior observed by another. Similarly, the remedies used in security to reduce the bandwidth of timing channels are worse than the original problem from the perspective of partitioning, because they introduce further unpredictability into system behavior.

Formal analysis of pure timing channels is generally based on information theory (e.g., [75, 76]), but there is dispute over whether some channels (e.g., the disk arm channel) really are timing channels, storage channels, or a combination of the two [112]. Consequently, formal description and analysis of such channels is difficult, and informal methods are generally employed. As described in Section 3.1.2, static partition scheduling requires implementation choices (strict determinism, no concurrent I/O) that eliminate the mechanisms that could serve as timing channels. In systems that do not require such strict temporal partitioning, the techniques used in computer security to identify timing channels [112] might help reveal unexpected sources of temporal interference.

Denial of service can be seen as an extreme type of timing channel: the perceived performance of some resource is reduced to an unacceptable level, rather than merely modulated. In the limit, the resource may become unavailable to some processes.

The possibility of this limiting case is usually equivalent to the existence of a storage channel. For example, if file space is shared between two processes, then one can deny service to the other by consuming all available space—but this is also a channel by which one process can convey information to another (the receiving channel attempts to create a file: success is taken as a 1 bit, failure as a 0; the transmitting process determines the outcome by consuming and releasing space). Because denial of service is related to timing and storage channels, it can be prevented by enforcing strict spatial and temporal partitioning. In general-purpose systems, the strictness of these mechanisms may be considered undesirable: they would require, for example, fixed per-process allocations of file space. Attempts to provide flexible resource allocation without incurring the risk of denial of service require "user agreements" that place limits on the demands that each process may place on each resource and that are enforced by a "resource allocation monitor" or "denial of service protection base" [63, 73] (these are somewhat similar to the quality of service ideas used in multimedia systems [102]). Formalizations of these approaches are stated in terms of fair or maximum waiting times [32, 113].

These more elaborate treatments of denial of service are probably unacceptable in strictly partitioned systems because they still allow the response perceived by one application to be influenced, even if not denied, by another. They may also be unnecessary in partitioned systems because the requirements for temporal partitioning seem stronger than those for denial of service: thus, denial of service should automatically be excluded in any system that provides strict temporal partitioning. Formal justification for this claim would be an interesting and worthwhile exercise.

4.4 Application to Partitioning

The formal models for computer security reviewed in the previous sections provide several ideas that seem applicable to partitioning. In particular, the central idea of noninterference—that the behavior perceived at one security level should be independent of actions at higher levels—can be reinterpreted in the context of partitioning and fault tolerance by supposing that ordinary behavior should be independent of faults: that is, faults are actions invoked by the environment, which is at a level that should be noninterfering with the level of ordinary users. This approach has been explored by Weber and by Simpson [99, 100, 109, 110]. It works well as a specification for partitioning when the partitions are completely isolated (in which case it is equivalent to the strict form of separability): if we have two partitions A and B that do not communicate in any way, then saying that the behavior of B must be independent of that of A is a good way to say that faults in A must not affect B. It works less well when A has to communicate with B: noninterference says only that A interferes with B and does not discriminate between legitimate interference (the known communication stream) and illegitimate (e.g., changes to B's private data).

This example shows that the concerns of security are, in a certain sense, too coarse to capture those of partitioning: security is concerned only with *whether* information can flow from A to B, not with *how* the flow can affect B. Channel control and its formalization by intransitive noninterference does allow the desired discrimination, but only at the cost of introducing a third component C to represent the buffer used for the intended A to B communication stream. Using intransitive flows, we would specify $A \leadsto C \leadsto B$ and $A \not\leadsto B$. This approach seems to capture some of the concerns of partitioning, but the introduction of the third component is artificial and unattractive.

A more fundamental objection to the idea that noninterference can serve as a model for partitioning is that partitioning is a safety property (because violations of partitioning occur at specific points in specific runs) whereas noninterference is not even a "property" (recall page 46). This suggests that noninterference is an unnecessarily subtle notion for partitioning, and that something simpler should suffice.

There is another sense in which the concerns of security diverge from those of partitioning: security assumes that all components are untrustworthy and that the mechanisms of security must be set up so that only allowed information flows occur, no matter how the components behave. In partitioning, however, we are concerned only with misbehavior by faulty partitions and are willing to trust nonfaulty components to safeguard their own interests. For example, suppose that two components A and B are statically scheduled and that each begins execution at a known entry point each time it is scheduled (this is the restart model of partition swapping). Suppose further that each has an area of "scratchpad" memory that is assumed to be "dirty" at the start of each execution: that is, the software in each of A and B is verified to perform its functions with no assumptions on the initial contents of the scratchpad memory. Finally, suppose that A and B are required to be isolated from one another. Then the scratchpad can be shared between A and B under the partitioning interpretation of isolation, but not under the corresponding security interpretation. The reason is that when B receives control, the scratchpad may contain data written by A; under the security interpretation we may assume nothing about even a nonfaulty B (in particular, that it will not "peek" at the data left by A), and so the scratchpad is a channel for information flow from A to B in violation of the isolation security policy. In the partitioned system, we accept (or specify) that a nonfaulty B does not do this, and our concern is to be sure that A (even if faulty) does not write outside its own memory or the scratchpad. Notice that this arrangement would not be safe in the restoration model of partition swapping, because A could preempt B, change its scratchpad, and then allow B to resume.

These examples demonstrate that the concerns of partitioning and security, although related, do not coincide. Thus, although formal treatments of partitioning may possibly be developed using ideas from computer security, they cannot be based directly on existing security models. Research to develop formal models of partitioning, and to refine the distinctions between partitioning and security, would be illuminating for both fields.

Chapter 5

Conclusion

We have reviewed some of the motivation for integrated modular avionics and the requirement for partitioning in such architectures. We then considered mechanisms for achieving partitioning, the interactions between these mechanisms and those for system structuring, scheduling, and fault tolerance, and issues in providing assurance for partitioning. Finally, we reviewed work in computer security that has similar motivation to partitioning.

Although partitioning is a very strong requirement and imposes many restrictions, there is a surprisingly wide range of architectural choices that can achieve adequate partitioning. The space of these design choices is seen most clearly in scheduling, where both static and dynamic schedules seem able to combine flexibility with highly assured partitioning.

The strongest need for future work is to develop the narrative description given here into a mathematical framework that will permit rigorous analysis of architectural choices for partitioned systems and provide a strong basis for the assurance of individual designs. There is already some significant work in this direction [24, 26, 106, 111], but great opportunities remain, particularly with respect to distributed systems and temporal partitioning. We are examining these topics in current work and will describe our results in a successor to this report.

References

[1] *ARINC Specification 651: Design Guidance for Integrated Modular Avionics.* Aeronautical Radio, Inc, Annapolis, MD, November 1991. Prepared by the Airlines Electronic Engineering Committee.

[2] *ARINC Specification 659: Backplane Data Bus.* Aeronautical Radio, Inc, Annapolis, MD, December 1993. Prepared by the Airlines Electronic Engineering Committee.

[3] *ARINC Specification 629: Multi-Transmitter Data Bus; Part 1, Technical Description (with five supplements); Part 2, Application Guide (with one supplement).* Aeronautical Radio, Inc, Annapolis, MD, December 1995/6. Prepared by the Airlines Electronic Engineering Committee.

[4] *ARINC Specification 653: Avionics Application Software Standard Interface.* Aeronautical Radio, Inc, Annapolis, MD, January 1997. Prepared by the Airlines Electronic Engineering Committee.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[6] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, January 1980.

[7] C. R. Attanasio, P. W. Markstein, and R. J. Phillips. Penetrating an operating system: A study of VM/370 integrity. *IBM Systems Journal*, 15(1):102–116, 1976.

[8] Algirdas Avižienis and Yutao He. The taxonomy of design faults in COTS microprocessors. In Rushby [93], pages 1–17.

[9] Henry M. Ballard, David M. Bicksler, Thomas Taylor, and H. O. Lubbes. Ensuring process security in the ALS/N environment. In *COMPASS '86 (Proceedings of the First Annual Conference on Computer Assurance)*, pages 60–68, Washington, DC, July 1986. IEEE Washington Section.

[10] D. E. Bell and L. J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, Bedford, MA, March 1976.

[11] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Fifteenth ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain, CO, December 1995. (ACM Operating Systems Review, Vol. 29, No. 5).

[12] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR 3153, Mitre Corporation, Bedford, MA, June 1975.

[13] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings 8th DoD/NBS Computer Security Initiative Conference*, pages 18–27, Gaithersburg, MD, September 1985.

[14] Mikhail Chernyshov. Post-mortem on failure. *Nature*, 339:9, May 4, 1989.

[15] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987. IEEE Computer Society.

[16] Raymond K. Clark, Douglas M. Wells, Ira B. Greenberg, Peter K. Boucher, Teresa F. Lunt, Peter G. Neumann, and E. Douglas Jensen. Effects of multilevel security on real-time applications. In *Proceedings of the Ninth Annual Computer Security Applications Conference*, pages 120–129, Orlando, FL, December 1993. IEEE Computer Society.

[17] Henry S. F. Cooper Jr. Annals of space (the planetary community)—part 1: Phobos. *New Yorker*, pages 50–84, June 11, 1990.

[18] Henry S. F. Cooper Jr. *The Evening Star: Venus Observed.* Farrar Straus Giroux, New York, NY, 1993.

[19] Robert D. Culp and George Bickley, editors. *Proceedings of the Annual Rocky Mountain Guidance and Control Conference*, volume 81 of *Advances in the Astronautical Sciences*, Keystone, CO, February 1993. American Astronautical Society.

[20] Sadegh Davari and Lui Sha. Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *ACM Operating Systems Review*, 26(2):110–120, April 1992.

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

[22] David L. Detlefs. An overview of the Extended Static Checking system. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 1–9, San Diego, CA, January 1996. Association for Computing Machinery.

[23] *Profibus Standard: DIN 19245*. Deutsche Industrie Norm, Berlin, Germany. Two volumes.

[24] Ben L. Di Vito. A formal model of partitioning for integrated modular avionics. NASA Contractor Report CR-1998-208703, NASA Langley Research Center, August 1998.

[25] Eileen M. Dukes. Magellan attitude control mission operations. In Culp and Bickley [19], pages 375–388.

[26] Bruno Dutertre and Victoria Stavridou. A model of non-interference for integrating mixed-criticality software components. In Rushby [93], pages 283–318.

[27] The interfaces between flightcrews and modern flight deck systems. Report of the FAA human factors team, Federal Aviation Administration, 1995. Available at `http://www.faa.gov/avr/afs/interfac.pdf`.

[28] *System Design and Analysis*. Federal Aviation Administration, June 21, 1988. Advisory Circular 25.1309-1A.

[29] *RTCA Inc., Document RTCA/DO-178B*. Federal Aviation Administration, January 11, 1993. Advisory Circular 20-115B.

[30] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Sixth ACM Symposium on Operating System Principles*, pages 57–65, November 1977.

[31] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In SOSP-16 [101], pages 38–51.

[32] Virgil D. Gligor. A note on denial-of-service in operating systems. *IEEE Transactions on Software Engineering*, SE-10(3):320–324, May 1984.

[33] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982. IEEE Computer Society.

[34] J. A. Goguen and J. Meseguer. Inference control and unwinding. In *Proceedings of the Symposium on Security and Privacy*, pages 75–86, Oakland, CA, April 1984. IEEE Computer Society.

[35] B. D. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in retrospect. In *Proceedings of the Symposium on Security and Privacy*, pages 13–23, Oakland, CA, April 1984. IEEE Computer Society.

[36] James W. Gray, III. On introducing noise into the bus-contention channel. In SSP'93 [45], pages 90–98.

[37] J. Thomas Haigh and William D. Young. Extending the noninterference version of MLS for SAT. *IEEE Transactions on Software Engineering*, SE-13(2):141–150, February 1987.

[38] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of μ-kernel-based systems. In SOSP-16 [101], pages 66–77.

[39] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1985.

[40] Harry Hopkins. Fit and forget fly-by-wire. *Flight International*, pages 89–92, December 3, 1988.

[41] Kenneth Hoyme and Kevin Driscoll. SAFEbus™. *IEEE Aerospace and Electronic Systems Magazine*, 8(3):34–39, March 1993.

[42] Wei-Ming Hu. Reducing timing channels with fuzzy time. In SSP'91 [44], pages 8–20.

[43] M. Huguet. The protection of the processor status word of the PDP-11/60. *ACM Computer Architecture News*, 10(4):27–30, June 1982.

[44] *Proceedings of the Symposium on Security and Privacy*, Oakland, CA, May 1991. IEEE Computer Society.

[45] *Proceedings of the Symposium on Security and Privacy*, Oakland, CA, May 1993. IEEE Computer Society.

[46] *ISO Standard 11898: Road Vehicles—Interchange of Digital Information—Controller Area Network (CAN) for High-Speed Communication*. International Standards Organization, Switzerland, November 1993.

[47] Jeremy Jacob. A note on the use of separability for the detection of covert channels. *Cipher (Newsletter of the IEEE Technical Committee on Security and Privacy)*, pages 25–33, Summer 1989.

[48] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In SOSP-16 [101], pages 52–65.

[49] Paul A. Karger and John C. Wray. Storage channels in disk arm optimization. In SSP'91 [44], pages 52–61.

[50] Rick Kasuda and Donna Sexton Packard. Spacecraft fault tolerance: The Magellan experience. In Culp and Bickley [19], pages 249–267.

[51] Philip J. Koopman, Jr. Perils of the PC cache. *Embedded Systems Programming*, 6(5):26–34, May 1993.

[52] Herman Kopetz and R. Nossal. Temporal firewalls in large distributed real-time systems. In *6th IEEE Workshop on Future Trends in Distributed Computing*, pages 310–315, Tunis, Tunisia, October 1997. IEEE Computer Society.

[53] Hermann Kopetz. Should responsive systems be event-triggered or time-triggered? *IEICE Transactions on Information and Systems*, E76-D(11):1325–1332, November 1993. Institute of Electronics, Information, and Communications Engineers, Japan.

[54] Hermann Kopetz. *Real-Time Systems: Design Princples for Distributed Embedded Applications.* The Kluwer International Series in Engineering and Computer Science. Kluwer, Dordrecht, The Netherlands, 1997.

[55] Hermann Kopetz. A comparison of CAN and TTP. Technical report, Technische Universität Wien, Vienna, Austria, March 1998.

[56] Hermann Kopetz. The time-triggered (TT) model of computation. Technical report, Technische Universität Wien, Vienna, Austria, March 1998.

[57] Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[58] A. A. Lambregts. Automatic flight controls: Concepts and methods. Draft paper by FAA National Resource Specialist for Advanced Controls, January 1998.

[59] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[60] C. E. Landwehr. A survey of formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[61] John P. Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm—exact characterization and average case behavior. In *Real Time Systems Symposium*, pages 166–171, Santa Monica, CA, December 1989. IEEE Computer Society.

[62] K. Rustan M. Leino and Rajeev Joshi. A semantic approach to secure information flow. Technical Report 032, Digital Systems Research Center, Palo Alto, CA, December 1997.

[63] Jussipekka Leiwo and Yuliang Zheng. A method to implement a denial of service protection base. In Vijay Varadharajan, Josef Pieprzyk, and Yi Mu, editors, *Information Security and Privacy: Second Australasian Conference (ACISP '97)*, pages 90–101, Sydney, Australia, July 1997. Volume 1270 of *Lecture Notes in Computer Science*, Springer-Verlag.

[64] S. B. Lipner. A comment on the confinement problem. In *Fifth ACM Symposium on Operating System Principles*, pages 192–196. ACM, 1975.

[65] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[66] C. Douglass Locke. Software architecture for hard real-time applications: Cyclic executives vs. priority executives. *Real-Time Systems*, 4(1):37–53, March 1992.

[67] Keith Loepere. Resolving covert channels within a B2 class secure system. *ACM Operating Systems Review*, 19(3):9–28, July 1985.

[68] Yoshifumi Manabe and Shigemi Aoyagi. A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling. *Real-Time Systems*, 14(2):171–181, March 1998.

[69] R. A. Mayer and L. H. Seawright. A virtual machine time sharing system. *IBM Systems Journal*, 9(3):199–218, 1970.

[70] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the Symposium on Security and Privacy*, pages 161–166, Oakland, CA, April 1987. IEEE Computer Society.

[71] John McLean. A comment on the "basic security theorem" of Bell and La Padula. *Information Processing Letters*, 20:67–70, 1985.

[72] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 79–93, Oakland, CA, May 1994. IEEE Computer Society.

[73] Jonathan K. Millen. A resource allocation model for denial of service. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 137–147, Oakland, CA, May 1992. IEEE Computer Society.

[74] Mark Moriconi, Xiaolei Qian, R. A. Riemenschneider, and Li Gong. Secure software architectures. In *Proceedings of the Symposium on Security and Privacy*, pages 84–93, Oakland, CA, May 1997. IEEE Computer Society.

[75] Ira S. Moskowitz, Steven J. Greenwald, and Myong H. Kang. An analysis of the timed Z-channel. In *Proceedings of the Symposium on Security and Privacy*, pages 2–31, Oakland, CA, May 1996. IEEE Computer Society.

[76] Ira S. Moskowitz and Alan R. Miller. Simple timing channels. In *Proceedings of the Symposium on Security and Privacy*, pages 56–64, Oakland, CA, May 1994. IEEE Computer Society.

[77] A Nadesakumar, R. M. Crowder, and C. J. Harris. Advanced system concepts or future civil aircraft—an overview of avionic architectures. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, 209:265–272, 1995.

[78] *Integrity in Automated Information Systems.* National Computer Security Center, September 1991. C Technical Report 79-91.

[79] George C. Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997. Association for Computing Machinery.

[80] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, Seattle, WA, October 1996. Association for Computing Machinery.

[81] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus—an architecture for extensible distributed systems. In *Fourteenth ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, December 1993. (ACM Operating Systems Review, Vol. 27, No. 5).

[82] Gerald J. Popek and David R. Farber. A model for verification of data security in operating systems. *Communications of the ACM*, 21(9):737–749, September 1978.

[83] *DO-178B: Software Considerations in Airborne Systems and Equipment Certification.* Requirements and Technical Concepts for Aviation, Washington, DC, December 1992. This document is known as EUROCAE ED-12B in Europe.

[84] P. Richards. Timing properties of multiprocessor systems. Technical Report TDB60-27, Tech. Operations Inc., Burlington, MA, August 1960.

[85] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the Symposium on Security and Privacy*, pages 114–127, Oakland, CA, May 1995. IEEE Computer Society.

[86] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1):27–53, 1996.

[87] John Rushby. The design and verification of secure systems. In *Eighth ACM Symposium on Operating System Principles*, pages 12–21, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[88] John Rushby. Proof of Separability—A verification technique for a class of security kernels. In *Proc. 5th International Symposium on Programming*, pages 352–367, Turin, Italy, April 1982. Volume 137 of *Lecture Notes in Computer Science*, Springer-Verlag.

[89] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report SRI-CSL-92-2, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1992.

[90] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor Report 4551, December 1993.

[91] John Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1995. Also available as NASA Contractor Report 4673, August 1995, and to be issued as part of the *FAA Digital Systems Validation Handbook* (the guide for aircraft certification). Reprinted in [97, pp. 1–42].

[92] John Rushby. A foundation for security kernel verification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, October 1995. Informal report.

[93] John Rushby, editor. *Dependable Computing for Critical Applications—7*, Dependable Computing and Fault Tolerant Systems, San Jose, CA, January 1999. IEEE Computer Society. To appear (page numbers refer to preliminary proceedings).

[94] John Rushby and David W. J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report SRI-CSL-95-10, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1995. Revised, July 1996. Available, with specification files, at `http://www.csl.sri.com/csl-95-10.html`.

[95] Marvin Schaefer, Barry Gold, Richard Linde, and John Scheid. Program confinement in KVM/370. In *ACM National Conference*, pages 404–410, Seattle, WA, October 1977.

[96] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[97] Roger Shaw, editor. *Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop)*, Bruges, Belgium, September 1995. Springer.

[98] O. Sibert, P. Porras, and R. Lindell. The Intel 80x86 processor architecture: Pitfalls for secure systems. In *Proceedings of the Symposium on Security and Privacy*, pages 211–222, Oakland, CA, May 1995. IEEE Computer Society.

[99] Andrew Simpson, Jim Woodcock, and Jim Davies. Safety through security. In *Proceedings of the Ninth International Workshop on Software Specification and Design*, pages 18–24, Ise-Shima, Japan, April 1998. IEEE Computer Society.

[100] Andrew Clive Simpson. *Safety through Security*. PhD thesis, Oxford University Computing Laboratory, 1996.

[101] *Sixteenth ACM Symposium on Operating System Principles*, Saint-Malo, france, October 1997. (ACM Operating Systems Review, Vol. 31, No. 5).

[102] Oliver Spatscheck and Larry Peterson. Defending against denial of service attacks in Scout. In *Proceedings of the 3rd Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, pages 59–72, New Orleans, February 1999. Usenix and Association for Computing Machinery.

[103] Christopher Temple. Avoiding the babbling-idiot failure in a time-triggered communication system. In *Fault Tolerant Computing Symposium 28*, pages 218–227, Munich, Germany, June 1998. IEEE Computer Society.

[104] Jonathan T. Throstle. Modeling a fuzzy time system. In SSP'93 [45], pages 82–89.

[105] Paulo Veríssimo, José Rufino, and Li Ming. How hard is hard real-time communication on field-buses? In *Fault Tolerant Computing Symposium 27*, pages 112–121, Seattle, WA, June 1997. IEEE Computer Society.

[106] Ben L. Di Vito. A model of cooperative noninterference for integrated modular avionics. In Rushby [93], pages 251–268.

[107] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

[108] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, Asheville, NC, December 1993. (ACM Operating Systems Review, Vol. 27, No. 5).

[109] D. G. Weber. Formal specification of fault-tolerance and its relation to computer security. In *Proceedings of the Fifth International Workshop on Software Specification and Design*, pages 273–277, Pittsburgh, PA, May 1989. Published as ACM SIGSOFT Engineering Notes, Volume 14, Number 3.

[110] Doug G. Weber. Fault tolerance as self-similarity. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, chapter 2, pages 33–49. Kluwer, Boston, Dordecht, London, 1993.

[111] Matthew M. Wilding, David S. Hardin, and David A. Greve. Invariant performance: A statement of task isolation useful for embedded application integration. In Rushby [93], pages 269–282.

[112] John C. Wray. An analysis of timing channels. In SSP'91 [44], pages 2–7.

[113] Che-Fn Yu and Virgil D. Gligor. A specification and verification method for the prevention of denial of service. *IEEE Transactions on Software Engineering*, 16(6):581–592, June 1990.