Security, Safety, and Partitioning*
Extended Abstract

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
`rushby@csl.sri.com`

Abstract. Important aspects of both security and safety are related to process encapsulation and controlled flow of information through known interfaces. Partitioning refers to architectural mechanisms that enforce these attributes. In this paper, we examine formal characterizations of partitioning.

1 Introduction

Many security objectives concern restrictions on the flow of information and can be portrayed informally by diagrams such as those in Figures 1–4. The intuitive reading of the diagrams is that information may flow only along the explicitly shown arrows. Figure 1 represents the well-known "multilevel" security policy, where information is allowed to flow only "upward" through the levels `unclassified`, `confidential`, `secret`, and `top secret`. Whereas multilevel security is concerned with *confidentiality*, Figure 3 represents a type of *authorization* requirement: messages prepared by the `user` may be sent to the `network`, but only via the `release agent`, which censors them according to some policy. Figure 2 combines both these ideas by adding a `downgrader` to the multilevel policy: information may flow "downward" in level, but only if approved by the downgrader. A final example is provide by Figure 4, which depicts a controller for end-to-end encryption: cleartext messages arrive at the `red` side and their bodies are sent through the `crypto` device, message headers must not be encrypted, however (because the network switches would then be unable to read them), so they are sent through the `bypass` (which performs various checks and limits the bandwidth provided) and recombined with their bodies in the `black` side and set out to the network. The critical requirement here is that there must be no direct path between the `red` and `black` sides.

On the other hand, many safety objectives concern restrictions on the propagation of *faults* but also can be portrayed by diagrams similar to those in Figures 1–4. For example, Figure 3 is reinterpreted in Figure 5 as part of a fly-by-wire system. Here, a faulty `air data` system may certainly pass bad data to the `autopilot` along the arrow, but is otherwise confined to its "box" and cannot interfere with the `autopilot`

* This research was supported by SRI internal investment funds, by NASA under contract NAS1-00079, and by NSA under contract MDA904-02-C-1196

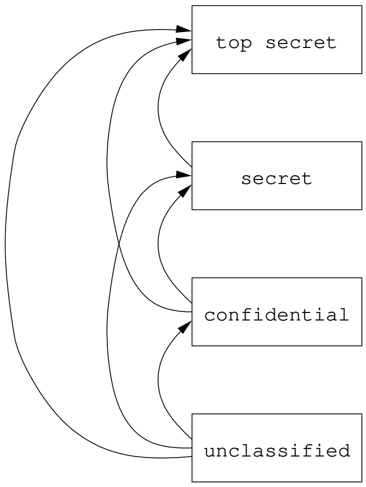

Fig. 1. Multilevel Security

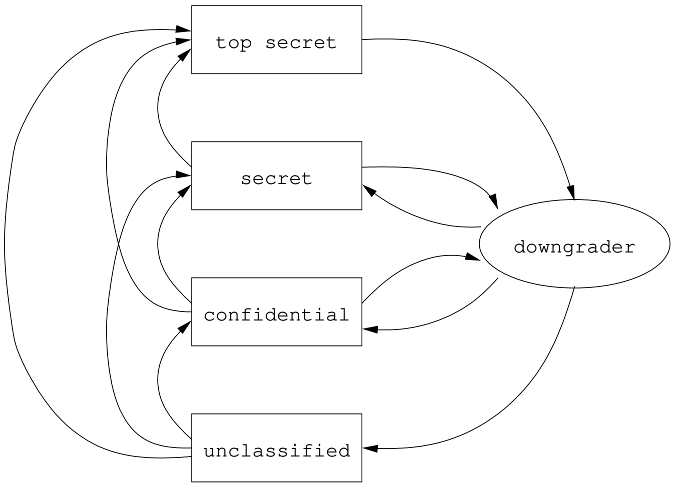

Fig. 2. Controlled Downgrading

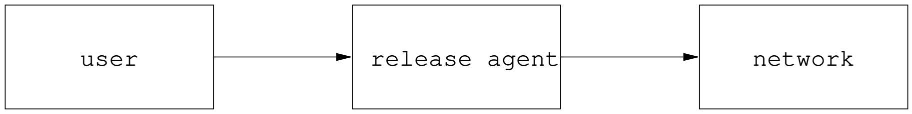

Fig. 3. Authorization by Release Agent

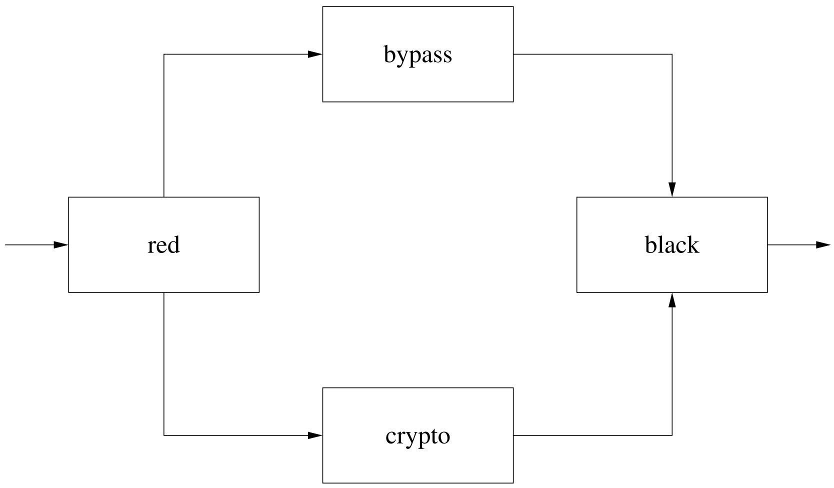

Fig. 4. End-To-End Encryption Controller

in its separate box. The similarity between some aspects of security and safety has been noted before [17, 18] and is also discussed in [14, 15].

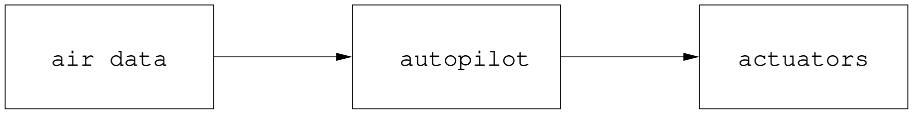

Fig. 5. Fly by Wire

Although the diagrams in Figures 1–5 have a certain intuitive appeal, precise specification of the objectives they are intended to represent is problematic. A satisfactory specification of the multilevel policy for sequential deterministic systems was provided by Feiertag, Levitt, and Robinson [2], and later generalized as *noninterference* by Goguen and Meseguer [3], but the extension of noninterference to concurrent and nondeterministic systems has been an active area of research for the past 25 years ([7] provides a good overview). All these noninterference formulations are restricted to cases where the “interference” relation (represented by the arrows) is transitive; the generalization to the intransitive case (as in Figures 3–4) has proved quite difficult and the correct formalizations are still debated [4, 8–10, 13].

Furthermore, in most applications, the goal is not to analyze whether specific software components satisfy a given security or safety policy but to define an *architecture* that *enforces* the policy. The idea of an architecture (implemented, for example by an operating system kernel, or a distributed “bus” such as TTA [6]) is that it provides an environment corresponding to the (empty) boxes and arrows of a policy diagram so that arbitrary software can be “dropped into” the boxes in full assurance that the intended policy will be satisfied. The intended policy is satisfied by virtue of mechanisms provided by architecture, independently of the actual software that runs within the boxes. This architectural approach is attractive for safety because, although faults may transform software in the boxes and render it “hostile” to the intended policy, the architecture will continue to enforce it. Of course, faults in the mechanisms of the architecture could render them ineffective, so they must be fault tolerant—but that is a separate problem. The same approach is attractive in security because it is expensive or infeasible (e.g., in the case of proprietary software) to examine the security properties of all the software components that will be employed; instead the large costs of development and assurance to high levels of security can be focused on the architectural mechanisms and amortized over many applications.

In avionics, the architectural approach suggested above is called *partitioning*; in security, there is no correspondingly abstract term but *kernelization* describes the mechanism most often used and is sometimes employed in a more general sense.

Partitioning is an attractive idea, and one that potentially can unify some aspects of security and safety. This is becoming important as systems are now being developed that must satisfy both kinds of requirements. For example, the Airbus A380 employs Integrated Modular Avionics (IMA), where partitioning is crucial for safety, and it also has connections between the flight control systems and the information systems in the cabin, so that security concerns are also important.

In this paper, we consider the notion of partitioning and the problem of how to make the idea precise.

2 Formalizing Flow Properties

Most functional attributes of computer systems can be understood in terms of *properties*: each run of a system generates a *trace*, which is a record of its behavior (the exact definition depends on the model of computation involved), and a property is a predicate over these traces. Bowen and Alpern [1] showed that all such system properties can be expressed as the conjunction of a *safety* property and a *liveness* property. Unfortunately, information flow, unlike data flow, is not a property: it may be impossible to tell whether information has flowed in one trace without looking at other possible traces. For example, if I observe that a certain variable has *not* changed, it may tell me that you have not received a certain message—because it would have changed if you had (cf. "the dog that did not bark" in the Sherlock Holmes mystery *Silver Blaze*). Information flow is therefore formalized as a predicate over *sets* of traces: the basic idea is to say that information has not flowed to an observer if they cannot tell the difference between one trace in the set and another. The primary challenge then is to determine how traces should be gathered into sets that should look equivalent to particular observers. (Secondary challenges are to define "look equivalent" and other technicalities that are largely dependent on the model of computation employed.)

The simplest example is classical noninterference where we say that the low level observer should not be able to tell the difference between one trace of the system and another in which just the high level components have been changed. But we have to be careful here: arbitrarily changing some of the high level components of a trace may result in something that is not a valid trace of the system. In this regard, noninterference resembles *counterfactual* reasoning. Counterfactual statements concern what might have been: e.g., "if you hadn't been driving so fast, you would not have crashed." Counterfactuals have long been studied by philosophers, logicians, psychologists and linguists and it is fair to say that their analysis still poses problems. One issue is that when we consider what might have been we cannot simply subtract out the event we are interested in because that might produce an inconsistent or impossible world (e.g., a world in which you were not born but your children are present) so we have to make some other adjustments—and how are we then to be sure that it is not those adjustments that are the cause of the consequences we wish to investigate.

Problems of this kind are responsible for some of the difficulties in extending noninterference to concurrent and nondeterministic systems, and to intransitive policies. When we consider partitioning, however, some of these subtleties become moot. The only leverage that an architecture can apply is to permit or deny specific data flows or accesses. As explained in [12], and elaborated by Schneider [16], this means that the only policies that can actually be enforced by architectural means are safety properties. Thus, noninterference formulations of desired policies will be enforced by safety properties that are often much stronger, so that different nuances in noninterference formulations will collapse to the same enforcement policy. Nonetheless, noninterference or counterfactual formulations seem closer to the real requirements, so we will seek such a formulation for partitioning and later relate it to the safety properties that enforce it.

3 Formalizing Partitioning

For security, we are concerned with *whether* information can flow from one place to another: in this sense, the arrows in Figure 3 between the `user` and `release agent`, and `release agent` and `network` are less important that the *absence* of an arrow between `user` and `network`. For safety, on the other hand, the arrows in Figure 5 are additionally intended to suggest an idea of *interface*: the `air data` system may send data to the `autopilot` via the interface associated with the arrow, but must not affect it in any other way. Thus, if the `air data` system becomes faulty, it may send bad data to the `autopilot` over the known interface, but cannot otherwise interfere with its operation (e.g., by monopolizing a CPU or bus shared with the `autopilot`, or by writing into the private memory of the `autopilot`). Formalizing this idea has proven quite difficult. One approach has been to forget the arrows (i.e., treat them as if they are absent as in Figure 6) and focus on the boxes: this is the policy of *separation* or *isolation* and it can be formalized as a degenerate case of noninterference.

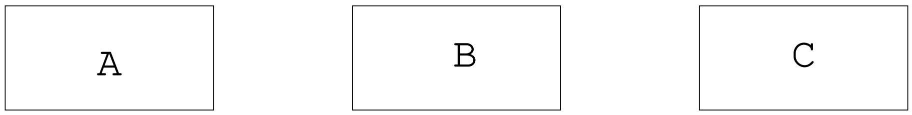

Fig. 6. Isolated Systems

The difficulty in trying to extend from the isolation of Figure 6 to the generic flow policy of Figure 7 is that most attempts to formalize addition of the flows allowed by the arrows (e.g., using intransitive noninterference) lack the idea of interface (suggested here by the labels `X` and `Y`) and allow indiscriminate interference from `A` to `B`. A necessary first step, therefore, is to formalize the notion of interface. We want to do this as generically as possible, without assuming a model of computation in which channels, or labeled events are primitive.

One idea that seems promising attacks the problem of interfaces directly using a counterfactual approach: simply remove the mechanisms that are intended to provide the allowed interfaces (i.e., "cut the wires") and establish that the resulting system is separated [11]. If there are any unintended interfaces, then these should be manifested as the inability to establish isolation. Unfortunately, as shown by Jacob [5], this approach is flawed because the unintended interface could be contingent on the intended one (e.g., a hostile component could check if the intended interface is present; if so, it uses an *un*intended interface to accomplish undesired flow, and otherwise does nothing).

Despite the difficulty identified by Jacob, we will approach formalization of partitioning through a counterfactual approach. Our first task is to identify what it means

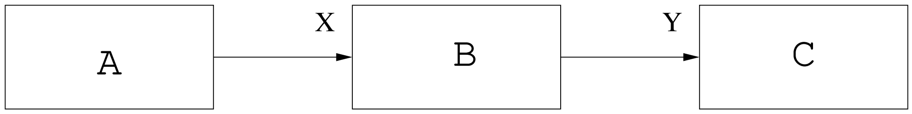

Fig. 7. Generic Policy

for an architecture to enforce an interface X from process A to process B. Our intuitive requirement is that the presence of A may certainly affect the behavior of B, but it may do so only through the interface X. One way to say this is that if we enclose A in a "wrapper" that limits the external interactions of A to X, then B should see no difference between interacting with A and the wrapped version of A. The problem with this approach is that A may legitimately use some other interfaces to interact with other processes and the wrapped process will prevent this, thereby changing A's behavior and that perceived by B. Rather than wrapping A, a more satisfactory approach is to replace A by a completely nondeterministic ("chaotic") process that changes only X, and to require that the behavior of B when interacting with A is a subset of its behavior when interacting with the chaotic process. If we denote the chaotic process on X by $Chaos_X$, process composition by $||$, projection (on to B) by $\downarrow_B$, and inclusion on behaviors by $\subset$, then we can say that the requirement for an architecture to enforce an interface X from A to B (written $A||_X B$) is given by the following definition

$$A||_X B \stackrel{\text{def}}{=} A||B \downarrow_B \subset \ Chaos_X||B \downarrow_B .$$

Of course, to make this formal, we need to provide a precise interpretation to the symbols employed (e.g., processes are relations on states, parallel composition is conjunction or disjunction, states are mappings from addresses to values, an interface is a set of addresses etc.) but I prefer to focus initially on the general ideas and to take care of the details later.

To specify that A cannot directly interfere with C (as in the policy indicated by Figure 7), we observe that in the absence of B, C should be unable to distinguish A from any other process, and the chaotic process in particular. Thus, the requirement that A should not interfere with C (written $A \not\rightsquigarrow C$) can be given by the following definition

$$A \not\rightsquigarrow C \Rightarrow A||C \downarrow_C = \ Chaos||C \downarrow_C .$$

It seems to me that this approach evades Jacob's objection to the "cutting the wires" approach because if there is an unintended channel that A is careful not to exploit in the absence of B, then $Chaos$ will expose it.

It could be that A does not interfere with C in the absence of B, but only in its presence (e.g., it might tamper with the interface Y). A way to test for this is to replace A by the process that is chaotic everywhere *except* its interface X to B (which we denote $Chaos_{\neg X}$) and then require that this leaves the behavior perceived by C unchanged:

$$A \not\rightsquigarrow C \Rightarrow Chaos_{\neg X}||B||C \downarrow_C = A\ ||B||C \downarrow_C .$$

In the full version of the paper, we will formalize these definitions and examine them in more detail.

References

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

[2] 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.

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

[4] 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.

[5] Jeremy Jacob. Separability and the detection of hidden channels. *Information Processing Letters*, 34(1):27–29, February 1990.

[6] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003.

[7] Heiko Mantel. Possibilistic Definitions of Security—An Assembly Kit. In *13th Computer Security Foundations Workshop*, pages 185–199, IEEE Computer Society, Cambridge, UK, July 2000.

[8] Heiko Mantel. Information flow control and applications—bridging a gap. In Jose Nuno Olivera and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe*, Volume 2021 of Springer-Verlag *Lecture Notes in Computer Science*, pages 153–172, Berlin, Germany, March 2001.

[9] Sylvan Pinsky. Absorbing covers and intransitive non-interference. In *Proceedings of the Symposium on Security and Privacy*, pages 102–113, IEEE Computer Society, Oakland, CA, May 1995.

[10] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *12th Computer Security Foundations Workshop*, pages 228–238, IEEE Computer Society, Mordano, Italy, June 1999.

[11] 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).

[12] John Rushby. Kernels for safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989. (Proceedings of a Symposium held in Glasgow, October 1986).

[13] 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.

[14] John Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[15] John Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Available at `http://www.csl.sri.com/˜rushby/abstracts/partitioning`,

and http://techreports.larc.nasa.gov/ltrs/PDF/1999/cr/NASA-99-cr209347.pdf; also issued by the FAA.

[16] Fred Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[17] 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, IEEE Computer Society, Ise-Shima, Japan, April 1998.

[18] 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.