Reformatted from Proceedings 7th DoD/NBS Computer Security Conference, Gaithersburg, Maryland, September 24-26 1984 (pp. 294-311).

# A Trusted Computing Base for Embedded Systems

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

**Abstract**

The structure of many secure systems has been based on the idea of a security kernel—an operating system nucleus that performs all trusted functions. The difficulty with this approach is that the security kernel tends to be rather large, complex, and unstructured.

This paper proposes an alternative structure for secure embedded systems. The structure comprises three layers. At the bottom is a *Domain Separation Mechanism* which is responsible for maintaining isolated "domains" (also known as "processes" or "virtual machines") and for providing controlled channels for their intercommunication. The other resources of the system (for example, devices and the more abstract entities, such as file systems, built upon them) are each controlled by independent *resource managers* which comprise the second layer of the system. The *applications* code provides the third layer. Components in both the resource management and applications layers are protected from each other by the domain separation mechanism. The Trusted Computing Base is composed of the domain separation mechanism and a reference validation mechanism associated with each resource.

The benefit of this approach is that it leads to a separation of concerns: each component of the embedded system performs a single, well-defined activity and can be understood (and verified) in relative isolation from all other components. Implementation and language issues are also discussed.

# 1 Introduction

This paper is concerned with the design of secure computer systems. The DoD Computer Security Center has established criteria which such systems should satisfy and has structured these criteria into several divisions according to the degree of assurance of security that they confer [6]. Although some of the criteria are quite specific, the document in which they are described (the "Orange Book" [6]) falls (intentionally) short of being a "how to" manual on secure systems design. Furthermore, although the document includes a section on the rationale underlying the criteria, many readers continue to find their motivation obscure and their interpretation difficult. In this paper, I will present my understanding of the motivation behind some of the criteria (specifically, those applying to evaluation classes B3 and A1) and I will provide my interpretation of how they should influence the design of secure embedded systems—by which I mean systems dedicated to the support of a single application (I wish to exclude the more complex case of general purpose systems, although many of my observations will apply to that class of systems as well).

It is important to stress that these are my personal opinions and interpretations concerning the DoD Evaluation Criteria; they have not yet been presented to, much less sanctioned by, the DoD Computer Security Center.

# 2 What's in a TCB?

There is considerable experimental evidence that conventional computer systems are not secure. Furthermore, repairing security flaws as they are discovered has proved an inadequate approach to the provision of truly secure systems. Firstly, many of the flaws in conventional systems reflect fundamental inadequacies in their design and their complete repair is infeasible. Secondly, there is no technique for demonstrating that all the security flaws have been eliminated from a system not designed with that aim in mind; penetration testing only reveals the presence of flaws, not their absence. The only sound approach to the provision of secure computer systems is to design security into those systems right from the start. Furthermore, the primary evidence that a system is secure must be based on the analysis of its design and implementation. The Evaluation Criteria express this as follows:

> "Systems representative of higher classes in division B and division A derive their security attributes from their design and

implementation structure. Assurance that the required features are operative, correct, and tamper proof under all circumstances is gained through progressively more rigorous analysis during the design process" [6, p5].

Thus, a secure computer system must contain mechanisms that are sufficient to guarantee its security in all circumstances, and it must be possible to provide compelling evidence that those mechanisms are entirely adequate to their task; it is not enough for the mechanisms to be be correct, they must be seen to be so. Generally speaking, small, simple, and localized mechanisms are easier to get correct, and more easily shown to be correct, than large, complex, or diffuse ones. The first task in the design of a secure system, therefore, is to find a way of structuring it so that its security mechanisms are localized as much as possible, and are as small and as simple as possible. In addition, since the evidence for the security of a system is to be provided by analysis of its security mechanisms, those mechanisms must be based on some overall concept of what constitutes security and the evaluation of those mechanisms must be performed with reference to that concept.

The Evaluation Criteria refer to the totality of security mechanisms within a secure system as its *Trusted Computing Base* (TCB). For Evaluation Class B3 and above, it is required that

> "... The TCB shall be internally structured into well-defined largely independent modules.
>
> "... The TCB modules shall be designed such that the principle of least privilege is enforced.
>
> "... The TCB shall be designed and structured to use a complete, conceptually simple protection mechanism with precisely defined semantics.[1] This mechanism shall play a central role in enforcing[2] the internal structuring of the TCB and the system. The TCB shall incorporate significant use of layering, abstraction, and data hiding. Significant system engineering shall be directed toward minimizing the complexity of the TCB and exclud-

[1] This is an incorrect use of the word "semantics" (which refers to the association of meaning with language). The word "behavior" seems to be intended.

[2] This also seems an inappropriate choice of words. Structure is a property of a design and cannot be "enforced" by a mechanism which is an artifact of that same design. A better choice of words would be "influencing" or "determining".

> ing from the TCB modules that are not protection-critical" [6, p37].

The "complete, conceptually simple protection mechanism" endorsed by the Evaluation Criteria is that of a *Reference Validation Mechanism* (RVM), which is the name given to a mechanism that implements the concept of a *reference monitor*. Unfortunately, the definitions given for these notions in the Evaluation Criteria are not particularly helpful. We are told that a reference monitor is

> "an access control concept that refers to an abstract machine that mediates all accesses by subjects to objects" [6, p112],

and that a reference validation mechanism

> "validates each reference to data or programs by any user (program) against a list of authorized types of reference for that user" [6, p64].

In order to understand what is intended by these terms, an analogy with the human world will be helpful. Imagine a bureaucracy that uses untrusted clerks to process classified information. Each clerk is assigned a *level* which is the most highly classified information that he may process: for example, a Secret level clerk may process information classified as Secret, Confidential, or Unclassified. The clerks are assigned to offices according to their levels: all the Unclassified clerks occupy one office, the Confidential ones another, and so on. The information processed by the clerks is recorded in folders stored in vaults. There is a separate vault for each security classification: one contains only Unclassified folders, another contains the Confidential folders etc. Periodically, a clerk will need to retrieve a folder from a vault, or to return one. The exits from the clerks' offices all lead into a central hallway containing the vaults and patrolled by a guard. A clerk who leaves his office in order to retrieve information from a vault will be intercepted by the guard and allowed only to enter a vault with a classification less than or equal to his own level. Once he has obtained the information required, the clerk will leave the vault and the guard will escort him back to his office. A clerk who wishes to deposit information in a vault will be treated similarly, except that the guard will allow him to enter only a vault with classification *greater than* or equal to his own level. The clerks have no memory of their own, everything they process must be written down on paper; the guard ensures that clerks are empty-handed when they enter a vault classified below their own level, and also when they leave a vault classified above their own level.

It seems clear that this arrangement provides security in the sense that no information derived from a folder in a highly classified vault can ever wind up in a folder stored in a vault of lower classification. The interesting question here is to enquire what it is about the arrangement that makes it secure.

Clearly, the guard plays an important role in the security of this system—he is, in fact, its reference validation mechanism in that he "validates each reference to data (i.e. folders) by any program (i.e. clerk) against the type of reference authorized for that program" [6, p64 (paraphrased]. But is the guard the only characteristic of this system that is necessary to its security? Clearly not—certain properties of the *environment* in which he operates are crucial to his ability to perform his task correctly. Suppose, for example, that the different offices were not isolated from one another, but had interconnecting doors. The system would then provide no security at all since Unclassified clerks could enter the Secret office and could there observe and record information classified as Secret. Similar problems would arise if the vaults had interconnecting doors, or if clerks could slip by the guard and evade his supervision. The Evaluation Criteria address these problems by citing the following three design requirements which must be met by a RVM [6, p64].

- The RVM must be tamper-proof,
- The RVM must *always* be invoked, and
- The RVM must be small enough to be subject to analysis and tests, the completeness of which can be assured.

These requirements are often referred to as the *isolation*, *completeness* and *correctness* of an RVM, respectively.

Of these three requirements, only the last (correctness) is really a property of the RVM itself; the other two are more properly regarded as properties of the environment in which the RVM operates. Thus, although the concept of a reference monitor may serve to guide and motivate the design of a TCB, it is clear that a TCB must be more than just an RVM. The *first* requirement of a TCB is that it should create an environment in which an RVM can operate securely.

The type of environment required for this purpose is one of cleanly separated "domains" in which untrusted programs can operate with no opportunity to interfere with each other. Domains cannot be completely isolated, of course, or there would be no possibility for information flow between the different security levels at all—the purpose of a secure system is not to *prohibit* information flow between different security levels, but to *control* it. The Domain Separation Mechanism (DSM) of a TCB must therefore provide channels for inter-domain communication, but these communication channels must be under strict control and must be "wired up" correctly. (For example, there must be no channels directly connecting untrusted domains of different security levels.) Abstract domains and communication channels are represented by the isolated "offices" and "vaults" and by the "doors" and hallways of the human-world example described earlier.

It is interesting to observe that for certain simple systems, domain separation and controlled "wiring" of the inter-domain communication channels may be all that is necessary to provide security: an explicit RVM is not always necessary. In the example we have been considering, it is conceivable that a cunning layout of corridors may make it possible to dispense with the guard altogether—a clerk wishing to obtain information from a vault would leave his office and find himself in a corridor giving access only to the vaults that he is allowed to enter. There is an apparent difficulty here in that in that each clerk is restricted to a *different* set of vaults depending on whether he wishes to read, or to write information. This problem can be overcome by providing different channels for different operations: a Secret clerk wishing to read information from a vault would exit his office by a door which gave him access to the Unclassified, Confidential, and Secret vaults, while one wishing to deposit information would leave by a door leading to the Secret and Top Secret vaults.

We should now ask whether this scheme of carefully routed corridors is really any different to the original one involving the guard. I think not: both schemes are implementations of the reference monitor concept, but whereas the guard is a *run-time* mechanism—checking each access as it is about to occur—the corridor routing approach is applied at *system configuration* time.

So, to summarize so far: domain separation is a necessary prerequisite for the implementation of an RVM. In some simple cases, the RVM can be "hard-wired" into the routing of the inter-domain communication channels. (Certain communications processing applications lend themselves to this approach—see [3, 10] for an example of this type.) In more complex cases, it may be better to use a run-time RVM to check each inter-domain communication as it occurs.

The question we should now consider is whether a run-time RVM should be part of the DSM, or a distinct mechanism in its own right. In the former case, the DSM could monitor each inter-domain communication channel and consult its in-built RVM before allowing the communication to proceed. In the latter case, untrusted domains would not be allowed to directly communicate with each other at all; instead, they would have to relay their communication through a special domain containing an RVM. This reference validation domain would check each communication and pass on only those that were authorized.

On the surface, the second scheme appears to correspond to a better separation of concerns, but would also seem inefficient (in that it doubles the number of communication steps and introduces additional domain swaps). Further consideration, however, reveals that these are not the only two alternatives: there is a third which has certain advantages over both the others. In order to see this, it is necessary to consider the nature of inter-domain communication more carefully.

There are basically two reasons why one domain should need to communicate with another. The first reason is simply for the purpose of passing information: one domain passes information to another that has need of it. This style of communication is common in communications processing applications where messages go through several stages of processing. The second reason is quite different: it is performed in order to obtain a *service* from a domain which encapsulates a *resource*. Examples of such services include storing and retrieving files, and providing access to a communications line. Different kinds of resource naturally provide different kinds of services and the access control restrictions necessary to enforce security will naturally differ with the different services. For example, it is necessary to prevent domains from reading files classified *above* themselves, and to prevent them from writing files classified *below* their own clearance: the reading and writing of information have different security implications and this is reflected in the different access control restrictions that apply to the two operations. For this reason, it seems reasonable to have a separate RVM associated with each different type of resource; in this way, each RVM can be tailored to the particular characteristics of the service provided by the resource with which it is associated.

A counter-argument maintains that all operations on all resources can be characterized as either read-like or write-like. If this is the case, then only a single RVM is needed: it could operate by first consulting a record of whether the requested operation is read-like or write-like and then applying whichever of the two fundamental access control disciplines is appropriate. The benefit that can be claimed for this approach is that the crucial reference validation function is localized in a single place. This argument seems plausible, but it loses much of its force if resource managers (i.e. the domains which encapsulate resources) have to be *trusted*. For certain simple kinds of resource, it is possible to construct resource managers that do not need to be trusted, but this is possible only if a separate instance of the resource can be synthesized a for each security level, and if the resource manager does not retain private state information. (The vaults in our example have these characteristics.)

In order to understand why it is necessary to synthesize separate instances of a resource at different security levels it will be helpful to return to the analogy with secure office procedures. Suppose that the only communication between our bureaucracy and the outside world is via a single telephone line. When the phone rings, an operator answers it and interrogates the caller in order to determine his identity and authorized security level—this could require that the caller gives a secret password, for example. The telephone operator will then switch the call over the internal telephone system to an office containing clerks of an appropriate level. It is clear that the telephone operator must be trusted to perform these tasks correctly. Of course, we could require that the guard who controls access to the vaults doubles up as the telephone operator, but this evades the issue. The procedures necessary to operate the telephone securely are quite different to those concerned with access to the vaults and the principle of separation of concerns indicates that they are best treated separately: the best structured mechanism is that which uses a separate and trusted telephone operator (who personifies a resource manager for the telephone line).

To understand the significance of retaining state information in a resource manager, suppose that the vaults do not merely contain a haphazard collection of folders which ordinary clerks can sort through at will, but are instead maintained as orderly filing systems. This could be achieved by having a filing clerk inside each vault who obtains files on behalf of the ordinary clerks and who stores returned files back in their proper places. In order to maintain the filing system, each filing clerk is allowed to maintain a directory recording which file is stored where inside his vault. Now suppose the Unclassified vault contains, among others, 26 folders of different sizes and suppose further that one of the ordinary Secret level clerks makes 6 trips to the Unclassified vault and retrieves the 5th, 14th, 9th, 7th, 13th and 1st smallest folders, in that order. It is not hard to see that the Secret string "ENIGMA" has been communicated to the Unclassified filing clerk—who can now manufacture a folder containing this information and hand it to the next ordinary Unclassified clerk who visits his vault.

These examples should convince the reader that resource managers, such as the telephone operator and the filing clerks, generally need to be trusted to perform their functions *securely* as well as correctly. This being the case, it is surely most appropriate for each resource to have its own RVM associated with it—that RVM can then be designed to integrate cleanly with the other trusted functions of the resource manager.

Summarizing this discussion, I propose that trusted embedded computer systems should be structured as follows. At the bottom, closest to the hardware, there should be a *domain separation mechanism* whose purpose is to divide the system into a number of separate execution domains—virtual machines, in effect—which are interconnected by carefully "wired" communications channels. It is an engineering decision whether the reference monitor function at this level is accomplished by the fixed routing of the the inter-domain communications channels, or by a run-time reference validation mechanism within the domain separation mechanism.

Above the domain separation layer should come a *resource management layer*. Some resource managers will consist of simply the software needed to encapsulate and control some hardware device (i.e. a device driver). Others will synthesize more sophisticated resources out of primitive ones—for example, a file system may be built on top of a primitive disk resource. The code that manages each resource should be isolated in one or more domains; any trusted code must reside in a separate domain from that which is untrusted. Care and skill are needed to minimize the amount and complexity of the trusted code in each resource manager. This can often be achieved by careful layering. For example, it is a complex task to synthesize a secure file system directly on top of a raw disk driver. It may be better to first synthesize securely partitioned "mini-disks" on top of the single physical disk and to then build the secure file system on top of the secure mini-disks.

Access to all "multilevel" resources must be controlled by a reference validation mechanism that provides the only outside interface to the services of the resource. Most often, the reference validation mechanism will be part of a domain that performs other trusted functions concerned with the management of the resource.

At the top, above the resource management layer, should come the *application* layer comprising the code necessary to tailor the system to the intended application. The domain separation provided by the domain separation mechanism must be exploited to separate trusted from untrusted applications code, and to partition untrusted applications code operating with different security attributes (e.g. different "levels").

Within this structure, the Trusted Computing Base consists of the domain separation mechanism, together with all domains that perform trusted functions. These will include the reference validation and other trusted domains from the resource management layer, together with the few (if any) application-specific trusted domains from the applications layer.

This approach to system structuring is very similar to that employed in modern operating systems (e.g. Thoth [5] and Tunis [9]) and is in contrast to the older approach to secure system design in which nearly all trusted functions were combined with the domain separation mechanism to yield a rather large "security kernel" with little internal structure [1, 10]. The approach advocated here extends naturally to distributed systems (where domain isolation is achieved using separate processors and inter-domain communication uses external communications lines) and seems well able to satisfy the system architecture requirements of the Evaluation Criteria for B3 systems and beyond.

# 3 Implementation and Language Issues

In order to implement a TCB with the structure described in the previous section, it is sensible to begin with the DSM. One approach is to use *physical* domain separation—that is, to use separate processors for each domain. (See [4, 12] for systems based on this approach.) If, however, a single processor is to support multiple domains, then we have a choice of compile-time or run-time separation mechanisms.

The compile-time approach relies on a programing language implementation to provide separation. In order to be suitable for this purpose, the chosen programming language must be oriented towards the construction of programs from inter-communicating, but otherwise isolated, units (such as "tasks" or "processes"). Modern message-based programming languages for distributed systems (such as CSP and Gypsy) have this character; languages based on shared-variable parallelism are not suitable. The problem with the language-based, compile-time approach to domain separation is that it depends on the correctness of a (generally large and complex) compiler. Although the semantics of the language may indicate that domains with no explicit communication between them are unable to influence each other, there is a danger that a compiler bug, or possibly a deliberately planted Trojan Horse, may permit communication anyway. There is evidence that some real compilers do contain serious security flaws: one notorious example concerns a Fortran compiler that allowed the creation and execution of arbitrary machine code, while a particularly insidious Trojan Horse is described in [13]. Since there is no practical technology for verifying compiler correctness, the Evaluation Criteria appear not to sanction this approach.

Instead, the Evaluation Criteria for Divisions B and A require use of a run-time domain separation mechanism. (The full requirements are introduced at the B2 level, but some of the key requirements are present at the B1 level also.) A run-time domain separation mechanism relies on hardware protection mechanisms to provide domain isolation. The protection mechanisms provided by currently available hardware are usually based on multiple CPU states (at least a superviser/user mode distinction is necessary) together with memory management functions. These may range from the fairly crude (e.g. the fixed set of PAR/PDR registers provided by a PDP-11/34) to the relatively sophisticated (e.g. the descriptor-based addressing scheme of the Intel iAPX286). The Evaluation Criteria for Class B2 and above require hardware protection mechanisms such as these to be present and to be exploited as follows.

> "The TCB shall maintain a domain for its own execution that protects it from external interference or tampering (e.g. by modification of its code or data structures). The TCB shall maintain process isolation[3] through the provision of distinct address spaces under its control. It shall make effective use of available hardware to separate those elements that are protection-critical from those that are not." [6, p30]

The TCB structure that I am advocating allocates exactly these tasks to its domain separation mechanism. I will use the term *separation kernel* to refer to a run-time DSM which operates in the way described above. In contrast to a conventional "security kernel", a separation kernel does nothing but provide domain separation and this gives it a conceptual simplicity that

[3] I interpret "process isolation", in the sense used here, as synonymous with domain separation.

is lacking in a security kernel. The next step is make the implementation correspondingly simple.

The task of a separation kernel is to manipulate the protection features of the hardware in order to manufacture separate domains. The kernel must also provide inter-domain communication and synchronization mechanisms and should probably hide some of the less attractive hardware features (such as interrupts—these should be mapped onto the standard inter-domain communication mechanism). By removing all complex tasks from the separation kernel, each of its remaining functions can be accomplished in a small fixed number of machine instructions. This makes it possible for the kernel to run with interrupts masked off—which enormously simplifies the kernel and contributes greatly to its comprehensibility, since it can now be understood as a single sequential program.

The functionality of a separation kernel as described here is almost identical to that of the "nucleus" of any modern operating system [5, 9] and is therefore based on a well understood technology. The primary difference between a conventional nucleus and a separation kernel is that the latter must provide absolutely rigorous separation between its client domains. Considerable simplification is possible in the case of embedded systems since their process structure is generally static. The separation kernel for an embedded system need not, therefore, support dynamic domain creation, and can use very simple domain scheduling strategies. Such a separation kernel should require no more that a *couple of hundred* machine instructions in total.

An important question concerns the choice of language in which a separation kernel should be programmed. Most high-level programming languages are ruled out immediately since they depend upon a run-time support system that is quite often *larger* than the kernel we wish to construct. Since the separation kernel is to be the fundamental security mechanism in the system, its behavior cannot be allowed to depend on any code but its own. Thus, the kernel implementation language must not require a run-time support system, it must permit special hardware registers to be named directly, and it must allow the use of special machine instructions. These requirement rule out all but assembler and a few system implementation languages, such as the sequential subset of Concurrent Euclid [9]. Personally, I can see no reason for using anything other than assembler since the code of a separation kernel consists almost exclusively of assignments to special hardware registers (e.g. loading the protection status or the memory management registers) and special instructions (e.g. those to set the interrupt mask or the processor status word). The behavior of a separation kernel written in a high level language cannot be inferred from the standard semantics for that language; its behavior and effects are primarily determined by the characteristics of the *hardware* which it manipulates.[4] For this reason, it is absurd to suggest that a separation kernel should be written in a high-level language in order that it be verifiable. The effect ascribed to the assignment statement `x := 0` by a standard programming language semantics does not begin to address the real behavior of the program when `x` is the processor status word! The task of verifying a separation kernel is rather different than conventional program verification. A technique for accomplishing the task is described informally in [10] and more formally in [11]. An application of the technique is described in [8].

Although a separation kernel must essentially be written in assembler (or in assembler disguised as a high-level language), one of the merits of the TCB structure advocated here is that it isolates all the machine dependencies in the (very small) separation kernel so that the rest of the system *can* be written in a high-level language. So the next question concerns the choice of language to be used in the rest of the system—that is, in the resource management and application layers. Essentially, any language, or set of languages, *could* be used, since each domain may contain whatever run-time support mechanisms are necessary for languages used within that domain. Inter-domain communication can be provided by subroutine calls on the separation kernel interface.

But while it may be *possible* to use any language(s) whatever, it does not follow that all languages are equally appropriate. An embedded system presumably has some overall purpose: all its domains must cooperate towards that end and must be understood in combination with each other. The programming language used should therefore be one which is matched to the environment created by a separation kernel: that is one of isolated domains with controlled inter-domain communications channels. Essentially, this environment simulates that of a *distributed system* and so the most appropriate programming languages are those intended for distributed systems. If such a language is used, then the separation kernel becomes, in effect, part of its run-time support system and the inter-domain communication facilities provided by the kernel must correspond to those assumed by the language. Unfortunately, the design of programming languages for distributed systems

[4] A simple separation kernel written in the sequential subset of Concurrent Euclid is described in [7]. However, I found the assembler code from which it was derived considerably easier to comprehend.

is still in its early stages and for systems to be constructed in the near future, it will probably be necessary to graft inter-domain communication primitives onto an existing sequential programming language whose choice is dictated by other factors.

Design of the inter-domain communications primitives that should be supported by the separation kernel is an interesting problem (see [2] for a discussion of communications primitives). Recall that the resource management layer of the proposed TCB structure provides *services* to *callers*. The form of communication mechanism that is most appropriate to this form of interaction is that of a *Remote Procedure Call* (RPC): the domain requesting the service must call the domain that provides (the interface to) it and must wait until the requested service has been performed and its results (if any) returned. From the caller's point of view, the behavior of an RPC is identical to that of an ordinary procedure call: the fact that the service is actually provided by a remote domain is invisible. As well as a simple semantics, RPCs have a simple message-passing implementation using a "blocking send with reply" and a "blocking receive" [2, 5].

In contrast to the resource management layer, where RPCs provide the communication mechanism of choice, inter-domain communication in the application layer may be better served by straightforward message passing (i.e. "non-blocking" send and receive). However, the implementation of non-blocking send and receive is more complex than that of their blocking counterparts, since it requires the buffering of messages that have been sent but not yet received. To my mind, the implementation of these primitives is more complex than is desirable within a separation kernel (remember, we want each kernel operation to execute in a fixed, small number of machine instructions, so that it will be safe to mask interrupts during the interval). A compromise approach is possible, however, by providing non-blocking message passing as a service of the resource management layer. This can be accomplished using a "queue management" domain that responds to RPCs containing requests to enqueue a message from one domain to another, or to dequeue a message from the input queue of its caller.[5]

[5]The question of how the messages constituting an RPC call and its reply are actually moved from one domain to another is an interesting one. The conventional solution of simply passing a pointer into a global message pool is obviously unsecure. Modern hardware often provides mechanisms whereby a segment containing a message can be mapped out of one domain's address space and into another, but it is not easy to integrate this mechanism into a high-level programming language. Also, the sanitization of message buffers required by the Evaluation Criteria [6, p15] renders this solution less attractive (in

As well as requiring only a simple, relatively efficient implementation, the use of RPCs as the basic inter-domain communication mechanism has another advantage: it can support the use of Ada. It is likely that many future secure system developments will require use of Ada. Unfortunately, Ada was designed while hardware and language technology were in transition [14] and its tasking facilities contain shared-variable features that are poorly matched to the environment of a distributed system—and to the environment described here. However, the Ada rendezvous is essentially an RPC mechanism [2] and my suggestion for accommodating Ada within secure systems development is to use the rendezvous as the (only) inter-domain communication mechanism. Within each domain, full Ada will be made available (including unrestricted multi-tasking and shared-variable communication) by an Ada run-time support system contained within the domain. Between tasks located in different domains, however, communications will be allowed only via the rendezvous mechanism, which in this case will be provided by the separation kernel rather than the standard run-time support. The integration of the inter-task rendezvous provided by the separation kernel and the intra-domain communications provided by the Ada run-time support system should eventually be made as seamless as possible—though it may require considerable research and development to reach this stage.

# 4 Summary and Conclusion

I have described a system structure suitable for implementing secure embedded systems. The structure is comprised of three layers. At the bottom is a *Domain Separation Mechanism* which is responsible for maintaining isolated "domains" (also known as "processes" or "virtual machines") and for providing controlled channels for their intercommunication. The other resources of the system (for example, devices and the more abstract entities, such as file systems, built upon them) are each controlled by independent *resource managers* which comprise the second layer of the system. The *applications* code provides the third layer. Components in both the resource management and applications layers are protected from each other by the domain separation mechanism. The Trusted Computing Base is composed of the domain separation mechanism and a reference monitor associated with

the absence of "write-only" protection). Physical copying of messages from one domain to another seems the most practicable mechanism.

each resource. This approach to system structuring is very similar to that employed in modern operating systems (e.g. Thoth [5] and Tunis [9]) and is in contrast to the older approach to secure system design in which nearly all trusted functions were combined with the domain separation mechanism to yield a rather large "security kernel" with little internal structure [1,10].

The implementation of a domain separation mechanism is called a separation kernel. There is little merit in attempting the implementation of a separation kernel in anything other than assembler; the total size of a separation kernel should be no more than a couple of hundred machine instructions. A message-based implementation of remote procedure calls is the most appropriate choice for the inter-domain communication mechanism to be provided by a separation kernel. More complex communications mechanisms can be provided as services of the resource management layer.

The resource management and application layers can be written in any high level language. It is proposed that Ada can be accommodated by mapping the inter-task rendezvous onto the inter-domain remote procedure call provided by the separation kernel.

The approach advocated here extends naturally to distributed systems (where domain isolation is achieved using separate processors and inter-domain communication uses external communications lines) and seems well able to satisfy the system architecture requirements of the Evaluation Criteria for B3 systems and beyond.

# References


[1] S. R. Ames Jr. Security kernels: A solution or a problem? In *Proceedings of the Symposium on Security and Privacy*, pages 141–150, Oakland, CA, April 1981. IEEE Computer Society.

[2] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[3] D. H. Barnes. The provision of security for user data on packet switched networks. In *Proceedings of the Symposium on Security and Privacy*, pages 121–126, Oakland, CA, April 1983. IEEE Computer Society.

[4] T. A. Berson, R. J. Feiertag, and R. K. Bauer. Processor-per-domain guard architecture. In *Proceedings of the Symposium on Security and*

*Privacy*, page 120, Oakland, CA, April 1983. IEEE Computer Society. (Abstract only).

[5] D. R. Cheriton. *The Thoth System: Multi-process Structuring and Portability*. Operating and Programming Systems Series. North-Holland, 1982.

[6] *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, December 1985. DOD 5200.28-STD (supersedes CSC-STD-001-83).

[7] P. F. Fisher. An operating system security kernel. Master's thesis, Computing Laboratory, University of Newcastle upon Tyne, England, September 1982.

[8] B. A. Hartman. A Gypsy-based kernel. In *Proceedings of the Symposium on Security and Privacy*, pages 219–225, Oakland, CA, April 1984. IEEE Computer Society.

[9] R. C. Holt. *Concurrent Euclid, the UNIX System, and TUNIS*. Addison-Wesley, 1983.

[10] 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).

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

[12] John Rushby and Brian Randell. A distributed secure system. *IEEE Computer*, 16(7):55–67, July 1983.

[13] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[14] P. Wegner. Capital-intensive software technology. *IEEE Software*, 1(3):7–45, July 1984.