Formal Construction of the Mathematically Analyzed Separation Kernel

W. Martin P. White F. S. Taylor A. Goldberg

Abstract

This paper describes the formal specification and development of a separation kernel. The Mathematically Analyzed Separation Kernel (MASK), has been used by Motorola on a smartcard project, and as part of a hardware cryptographic platform called the Advanced INFOSEC Machine (AIM). Both MASK and AIM were jointly developed by Motorola and the National Security Agency (NSA).

The paper will begin by describing the separation kernel concept and its importance to information security. Next it will illustrate the SPECWARE *formal development methodology used in the development of MASK. Experiences and lessons learned from this formal development process will be discussed. Finally, the results of the MASK development process are described, project successes are discussed, and related MASK research is highlighted.*

1. Introduction

In 1993 an informal separation kernel working group was established to study a new model called the *separation kernel paradigm*. The separation kernel paradigm was originally proposed in the 1980's by John Rushby [5]. It was investigated as an approach to replace the traditional security kernel [1] concept of the Orange Book [3]. While the separation kernel concerns itself with only two functions, namely, *information flow* and *data isolation*, traditional security kernels implement all of the functions specified for a given level of certification in the Orange Book.

One problem with past implementations of the security kernel paradigm was that application specific procedures and processes always found their way into the kernel and, consequently, into the trusted computing base. In the security kernel paradigm, the trusted computing base grows and eventually consumes a large portion of the system. The reason this happens in the security kernel paradigm is that the kernel and the trusted computing base are both unwilling to share the security burden with the application.

In contrast, the separation kernel empowers the application with the ability to define complex security policies and then provides the application the ability to enforce those policies itself. This is done by strictly enforcing an information flow policy between application components, which, in turn, ensures that critical components are not bypassed.

MASK is an implementation of the separation kernel paradigm. Developed using the SPECWARE formal methodology, MASK achieves a high assurance of separation. Moreover, MASK is currently being considered for submission for common criteria EAL7 [2] certification.

2. Separation kernel concept

Operating system research has concentrated on organizing basic operating system functions into a collection called a kernel. The kernel presents abstractions of the fundamental resource management mechanisms to other, less primitive, service providers. In operating system implementations that attempt to provide a basis for secure information processing, the kernel software is carefully constructed and evaluated. To aid the evaluation process, the kernel functions are implemented as relatively small programs that are independent of one another.

A separation kernel is charged with the critical task of providing separation among process spaces by manipulating the protection features of the system. A separation kernel creates "an environment which is indistinguishable from that provided by a physically distributed system." [5] The environment within the parallelogram of figure 1 depicts separation in a physically distributed system.

This view becomes somewhat more clouded when the functionality of these boxes and arrows is implemented inside an environment in the context of an operating system and hardware. Figure 1 is its entirety provides a view of this context and the possibilities of unintentional information flow due to the platform substrate. For example, while Box 4 should not be able to communicate to Box 1, the substrate allows information to pass from Box 4 to Box 1, as noted by the arcing arrow. This type of information flow is often referred to as a covert channel. That is, a mechanism which is used to establish communication between two entities, where the mechanism was not intended to be used for communication.

Naturally, separation kernels are desirable because they enforce the policy of restrictions on communications. These

0-7695-0710-7/00 $10.00 © 2000 IEEE

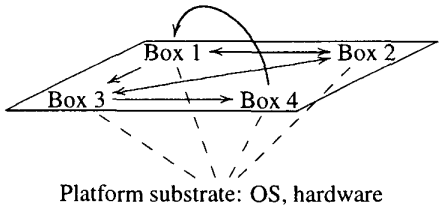


Figure 1. Potential covert channel

concerns are becoming more prevalent as networking of resources intensifies. For instance, there are many programs within the government which combine communications functionality implemented on physically separate boxes into a single communications device, exactly the situation introduced above. The resulting communications device requires a high degree of separation between the communications channels, since these channels may have different security levels and inherent separation requirements. Thus miniaturization of communications devices is an important stimulus for the separation kernel concept.

In addition, the following are benefits of the separation kernel:

- *Precise translation of the logical architecture into a physical implementation.* Because the separation kernel completely separates processes, it provides a faithful physical implementation of a logical architecture.
- *Fail safety.* When two processes have unintentional, and often unknown, interactions, it is not possible to consider the error properties of one process in isolation. Errors in one process may propagate to others. Indeed, this interdependency makes it difficult or impossible to determine the fail safety of the system.
- *Availability.* When processes are truly independent, it is possible to provide independent backups and check processes for a critical function.
- *Security.* A classical security property is red/black separation on a cryptographic device. When processes have unintentional communications, it is not possible to conclude that red processes may communicate to black processes only via a cryptographic process. However, if the system is implemented on a separation kernel, by a simple inspection of the system architecture diagram, it is possible to conclude that red processes communicate with black processes only via a cryptographic function.

3. Realization of a separation kernel

The principle abstraction for MASK is the cell. A cell is a domain of execution, or a context, and a collection of strands. Cells have globally unique CellIDs, and strands have globally unique StrandIDs. Each strand is a stream of instructions to be executed when a message is input to a strand of a cell. These abstractions, and their interrelationships, are illustrated in figure 2.

The context of the cell consists of memory segments, permanent and transient, defined by a starting address and a length. Additionally, each cell can execute only one strand at a time. In figure 2, Cell_B is currently not executing any strand. It will begin execution some time after it is granted access to a message from Cell_A bound for Strand_A. When the strand of the cell is launched, the cell receives access to the message segment(s) that make up the input message. The cell may also require scratch segments to process the message. The cell can dispose of transient segments either by sending them in a message to another cell, or by suspending. The substrate under the cell layer will ensure that when the cell suspends, access to all transient segments is removed, leaving the cell in its base state of having access only to its permanent segments.

Each strand is "run to completion." This means that the strands on the same processor do not preempt each other. The only exception is that an interrupt strand is run immediately upon receipt of the interrupt. Thus a strand's execution may be interleaved with the execution of interrupt strands.

Finally, a cell interfaces to its external world (hardware and other cells) only through the substrate. An example of cell interfaces, is illustrated in figure 2. In this figure, Strand_3 of Cell_A is running. First, the strand is launched by the kernel, at which time the strand is given access to its message segment(s) and scratch segment(s), by the kernel. After more of the thread of execution is consumed, the strand uses the Send interface to send a portion of its transient segments to Strand_A of Cell_B. Cell_A loses access to the segments transmitted at the time it uses the Send interface. The next interface used by Strand_3 of Cell_A is the Throw interface, causing an exception condition. This terminates execution of Strand_3 of Cell_A, and causes the Handler function to be run. When Strand_3 of Cell_A terminates, the kernel removes access to all of the transient segments remaining in Cell_A. If Strand_3 of Cell_A had not terminated by the exception condition, it would have reached the end of its strand, where it calls the TerminateStrand interface. This interface terminates the execution of the strand, at which time the kernel removes access to any transient segments still accessible to the cell.

3.1. Separation concept

This section describes two of the key axioms of the separation concept, so that the reader gets some feeling for the notion of an abstract specification, having only properties of the desired system, but no details of the system. The commuting diagram in (1) depicts the fundamental sorts and operations that define the system to be considered, the evolu-

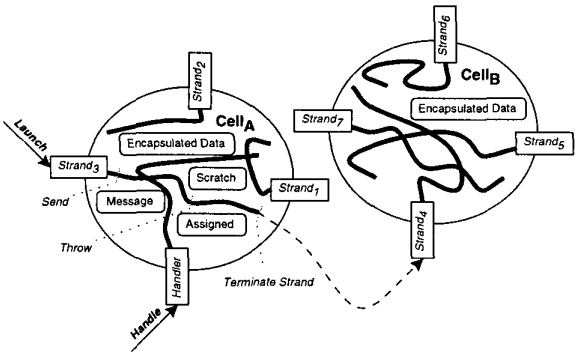


Figure 2. MASK cells

tion of the system over time, the subjects of the separation policy in the system, and the identification of the subjects of the system. These concepts are:

- *System:* Multiple Cell Abstraction (MCA)
- *System Evolution:* The Init and Next operators, producing all possible multiple cell abstractions.
- *Subject:* Cell or Single Cell Abstraction (SCA)
- *Identification:* CellID

$$\begin{array}{ccc} \mathsf{MCA}_1 & \xrightarrow{\mathsf{Next}_c} & \mathsf{MCA}_2 \\ \downarrow{\scriptstyle \mathsf{Fiber}_c} & & \downarrow{\scriptstyle \mathsf{Fiber}_c} \\ \mathsf{SCA}_{c_1} & \xrightarrow[\mathsf{FiberNext}]{} & \mathsf{SCA}_{c_2} \end{array} \qquad (1)$$

$$\mathsf{Fiber}_c \circ \mathsf{Next}_c = \mathsf{FiberNext} \circ \mathsf{Fiber}_c$$

The Fiber function is displayed in its curried form, i.e. as a subscripted function from MCA to SCA. The curry operation takes a function of two arguments, such as Fiber, and produces a family of functions of one argument, such as Fiber_c. There is one Fiber_c for each CellID c. Each function Fiber_c determines the SCA corresponding to the CellID c in the subscript.

The operators Init and Next are specified to be the constructors of the MCA. This means that all possible MCA states are the result of system initialization, and the advancement of the system state one step at a time by the Next operation. This specifies that the system cannot land in any "unspecified" states that might not satisfy the security constraints. The only possible states are arrived at using Init and Next, which obey all of the axioms of the specification.

There is a similar constraint upon the SCA, stating that any SCA must be the result of a Fiber operation on an MCA. Since the MCA is constrained to be in valid states, as constructed by Init and Next, this constrains the SCAs to be in valid states resulting from taking a Fiber of a valid MCA.

The axiom associated with the commuting diagram is shown in (1). This axiom equates the result of applying the functions along the two paths from MCA_1 to SCA_{c_2}.

This equation also introduces the Next and FiberNext operators. The Next operation advances the state of the system (the MCA), while the FiberNext operation advances the state of a single cell within the system. Similar to the Fiber function, Next requires two arguments, the system state (MCA), and the CellID of a cell. Next_c is the curried form of Next, it advances the system state by advancing the cell indicated by the subscript c.

In simple terms, what this equation is saying is that when the system state is advanced by one step, the change in the system state corresponds to a change in the state of one of its cells, as identified by the Fiber function. If you look at the system state before advancing it one step, and then look at it after advancing it one step, then there will be a unique cell c accounting for this advance in the state of the system. This cell c accounts for the advance in the state of the system by advancing its own single cell state by one step.

This equation ties in the Fiber function with the Next function. Without this tie, there is no guarantee that (for example) Next might not interchange two fibers, resulting in confusion. In summary:

- The Fiber function must choose a portion of the system state (the multiple cell abstraction) corresponding to the CellID.
- The portion of the system state chosen must be transformed in a way that makes sense with the Next and FiberNext operations.

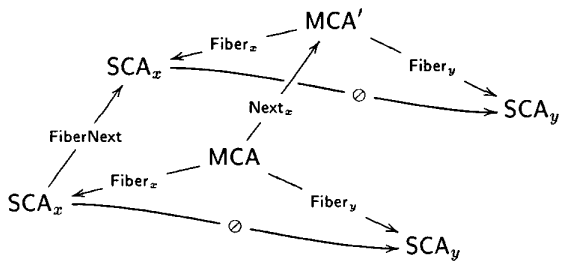
3.2. Separation axioms

The first separation axiom, or *communication policy*, describes which cells are permitted to communicate with which other cells. Since the communication policy references the concept of a cell, it is logical that the statement of the communication policy should reference the definitions of the Next and Fiber functions. Indeed, one purpose for separating the system into cells is to restrict communications. Figure 3 depicts the axiom for the communication policy in the MASK system.

(2) shows the communication policy axiom in its contrapositive form. It states that if the fiber of cell y changes as the result of advancing the state of cell x, it must be the case that x is permitted to communicate with y.

$$\begin{aligned} &\mathsf{Fiber}_y(\mathsf{MCA}) \neq \mathsf{Fiber}_y(\mathsf{Next}_x(\mathsf{MCA})) \\ &\Rightarrow \mathsf{Communicates}(x, y) \end{aligned} \qquad (2)$$

When the MASK kernel is elaborated in the cell level specifications, a particular consequence of the communica-



⊘ indicates that communication cannot occur.

Figure 3. Fibration and communication

tion policy axiom is that cell x can send messages to y only if cell x has permission to communicate with y.

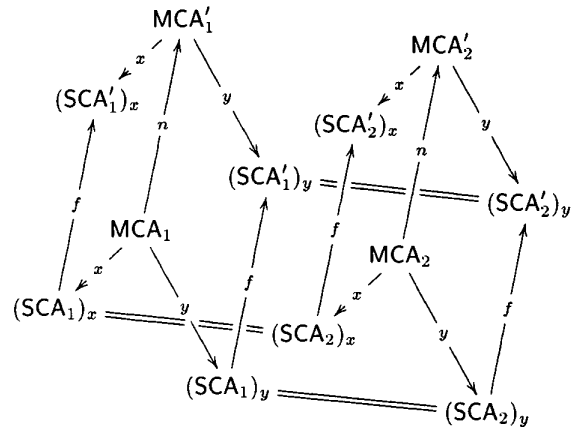
The second separation constraint upon cells is depicted in the (non-commuting) diagram in figure 4 and as the consequence in (3). These represent that if an action by cell x (advancing the state of the SCA for cell x) is going to change the state of cell y, then the change in the state of y depends only on the states of x and y. In other words, the new state of y is a function of the previous states of x and y. Without this axiom, if x changes y, it could do so by copying everything from a third cell z into the SCA of y. This would constitute an unspecified connection between z and y. The purpose of this axiom is to rule out such a connection.

$$\begin{aligned} \mathsf{Fiber}_x(\mathsf{MCA}_1) = \mathsf{Fiber}_x(\mathsf{MCA}_2) \Rightarrow \\ \mathsf{Fiber}_y(\mathsf{MCA}_1) = \mathsf{Fiber}_y(\mathsf{MCA}_2) \Rightarrow \\ \mathsf{Fiber}_y(\mathsf{Next}_x(\mathsf{MCA}_1)) = \mathsf{Fiber}_y(\mathsf{Next}_x(\mathsf{MCA}_2)) \end{aligned} \tag{3}$$

3.3. Kernel design choices

This section describes design choices that were made to preserve the separation property, while permitting a relatively simple table-driven kernel implementation. The basic concept for memory allocation is that a strand receives all of its memory resources when it is started (launched), and yields all of its memory resources when it terminates. Figure 2 shows a strand in operation, from Launch through TerminateStrand.

At strand launch time, the strand is allocated the memory it requires, according to the strand memory requirements. These requirements are constant for that strand, in that they are the same each time the strand is launched. The strand receives two kinds of transient segments upon launching, message and non-message segments.



f: FiberNext x: Fiber$_x$
n: Next$_x$ y: Fiber$_y$

Figure 4. The second separation axiom

- *Message segments* are segments which have been sent by some other strand to the strand being launched for further processing. It is the existence of this message that caused the strand launch.
- *Non-message segments* are segments which are described by the strand memory requirements.

The strand requirements describe the number and size of non-message transient segments required by the strand. As the strand executes, it undergoes interactions with the kernel, in this case by sending a message. If the strand elects to send some of its transient segments as part of the message, then access to these segments is lost at this point. Thus the strand can lose access to transient segments as it executes, but it cannot gain access to any more transient segments as it executes.

The allocation of transient segments to strands obeys the following properties:

- Before strand launch, the strand has access only to the permanent segments of the cell.
- Transient memory requirements of the strand are a function of the strand, known at compile time.
- The kernel does not launch the strand until there is sufficient memory to satisfy the strand's requirements.
- Transient segments that are not message segments are zeroized before allocation.
- As the strand executes, it can lose memory segments (by send message), but it cannot allocate more segments.
- Strand termination causes all transient segments to be released by the kernel.

From these properties, the following conclusions can be drawn:

- After strand termination, the strand has access only to the permanent segments of the cell.
- Each time the strand runs, the amount of memory available is the same, as specified by the strand memory requirements.

Because of these properties and their implications, the strand sees exactly the same amount of allocated memory each time it runs, and consequently, there is no flow of information caused by the amount of allocated memory while it is running. Thus, there is no covert channel stemming from the amount of allocated memory. This eliminates the typical memory allocation covert channel that results from the ability of one process causing resource exhaustion, while another detects it.

In general, kernel design choices were made to avoid situations where cells and strands can deplete resources internal to the kernel. There is still one case where the cells and strands can deplete a kernel resource. A send message could deplete the kernel schedule queue, so this is considered a fatal error. The kernel schedule queue must be engineered to be sufficient to handle the maximum number of strands scheduled for the maximum load of the application running on the kernel.

4. MASK program overview

The primary objective of the MASK project was to formally construct a separation kernel that could be used in the development of secure embedded systems. In an effort to formally construct such a separation kernel, the following major tasks were outlined and completed: (1) the specification of the separation property; (2) the development of an abstract description of the separation kernel; and (3) formal implementation of this abstract description.

The first step was *specification of the separation property.* This work began by considering typical operating system concerns such as how to handle registers, devices, transient memory, permanent memory and so on. These concepts were clouding our thinking about separation. Indeed, the many operating system details were threatening to lead to the formulation of a separation property that was tied to a particular operating system. Such a separation property might miss an important aspect of separation, or be so tied to a particular implementation that it could not be reused on even slightly different systems. In addition to avoiding the complicating details mentioned above, the objective of establishing with a high degree of certainty that our end system satisfied the separation property made it clear that the simplest formulation of the separation property was needed. Consequently, separation was considered in the abstract, in a way that would apply to separate channels, separate frequencies, separate accounts, etc. The result of this endeavor was a very short specification of the separation property that can be applied to any sort of system where separation is a concern, such as operating systems, radio frequencies, plumbing, etc.

The second step was an *abstract description of the separation kernel.* During the process of specification, the discovery was made that the step from an abstract specification of separation to an implementation level specification was too large of a step. What was missing was a specification of what services the kernel would offer to the user mode processes. This specification details the view of the world offered to each process. The view of the world includes direct access to resources, such as memory and registers, and indirect access to resources, such as the ability to send a message. If the kernel does something internally to its resources that makes it impossible for a process to send a message, then the kernel is in danger of implementing a covert channel, because the process could detect something about the kernel state by its ability to send a message. This specification details how each service offered to the user mode process affects the view of the world given to the user mode process. The specification was very valuable because the software designers could begin to design their software based on the interfaces detailed in this specification. The abstract description of the separation kernel was a refinement of the separation property, in that the separation property must be implied by this specification.

The final specification step was the *formal implementation of the abstract description.* Given the specification of the services to be provided, the implementation level specification details the data structures and algorithms used to build the kernel. For example, a priority queue was specified to implement the scheduling provided by the kernel. The formal implementation of the abstract description is a refinement of the abstract description of the separation kernel, in that it supports the services as specified, and has no other "features" besides those services.

There were two main sources of changes to the specifications. The first source of changes was requests for additional or extended services. One team was using the kernel to support the smartcard development. The concern for the smartcard team was to separate smartcard applications. For example, if two different credit card companies have a credit card application on the same smartcard, then neither wants to trust the other to interfere with their operations. Thus strict separation between these applications is important. As progress was being made, the team building the Advanced INFOSEC Machine (AIM) cryptographic processor began building their software. The main concern for AIM was establishment of separate cryptographic channels. Each channel might have data at a different security level, using different cryptographic algorithms. The AIM team had some

additional requirements for MASK. For example, to support AIM a processor version of the remote procedure call (RPC) mechanism between the separated processes was built.

Because of the hierarchy of specifications in three layers, when a new or extended service was considered, consideration was first given to whether the service could be implemented in accordance with the separation property. In addition, consideration could be given to how much impact the service would have on the existing implementation level specifications. By playing these two considerations off against each other, services could be provided that would support both the smartcard and AIM projects, that would embody the principle of separation, and that could be implemented economically.

A second source of changes came from the fact that the bottom layer specification was hand translated into C code, a formal implementation of the abstract description. When the coders took their hand to the design, they often found more efficient ways to implement the kernel, in terms of both memory and time. When such a design improvement was suggested, consideration was given how to modify the specifications to match the proposed change, and if the change could still establish the kernel as a refinement of the abstract description of the separation kernel. Note that the abstract separation property does not enter into the decision, because the abstract description of the separation kernel already embodies the separation property. By showing that the abstract description is still valid, it is shown that the separation property is preserved.

5. Formal methodology

SPECWARE, under development at Kestrel Institute in Palo Alto, California, is an environment for the specification and formal development of software with its primary objective being the correct development of entire systems. It is the mechanism which enables builders to organize building blocks, i.e., specifications, into a purposeful whole. It achieves this goal by incorporating ideas from mathematical logic which help to integrate several notions in software engineering. More specifically, SPECWARE provides for the formal composition of specifications and software components and the refinement of specifications to code.

The major specification layers and refinements used for the MASK development took place at three levels, the separation level, the multiple cell abstraction level, and the kernel level, as discussed in section 4.

5.1. Separation specification

The separation specification is a type of specification that may be less familiar to software designers. As highlighted in section 3.1, it specifies only mathematical properties of the desired system, so it is more akin to an axiomatic definition of a mathematical structure such as a *group* or a *field* than it is to a specification of a piece of software. It specifies only the separation properties of the system. It is extremely under-specified in the sense that there are many systems, other than MASK, which could satisfy this specification.

5.2. Multiple cell abstraction specification

The multiple cell abstraction (MCA) level specification details how the MASK kernel shall meet the separation specification, i.e. this specification instantiates the separation specification. The design choices of MASK are made in this specification, and in the refinement of separation to MCA, as described in section 3.3.

The MCA specification is built from the Single Cell Abstraction (SCA), which formally details how a cell, strand, segment, etc. behave, as informally described in section 3. It then formally describes what happens to a system composed of many cells. It serves as the basis for a description of the kernel to application programmers building cells and strands to run on the kernel. It describes precisely the semantics within a cell. The kernel calls available to the programmer are described in thorough detail. The assembly language instructions available to the programmer are described only in terms of the segments, e.g. the execution of a load operation will affect the registers of the cell, based upon the memory segments of the cell.

5.3. Kernel specification

The purpose of the kernel specification is to detail the actual data structures and algorithms used to construct the implementation of the kernel. Thus while the MCA is an application programmer view of the kernel, the kernel specification is the kernel implementor's view.

5.4. MCA to kernel specification refinement

The refinement of the MCA specification to the kernel specification is primarily a matter of data structure refinement. In the MASK implementation level specifications, the scheduling is accomplished by means of a priority queue. As shown in figure 5, the multiple cell abstraction layer version of the priority queue, called schedule map, is built from a total mapping from priority to a list of messages. Message further decomposes into a quadruple, consisting of a source strand, target strand, command word, and a list of message segments. A message segment list is a list of transient segment IDs (TSID). Thus, a priority queue specification gets built from many specifications: Lists, total maps, quadruples, TSID, strand ID, etc. In the schedule map specification, all of these component specifications are

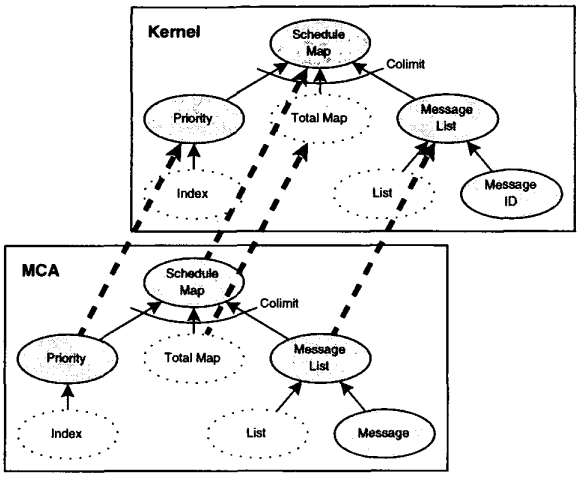


Figure 5. Priority queue refinement

combined, and this combination is then extended with the particular functions that implement the scheduling mechanism of the kernel.

At the kernel layer, the specification hierarchy for priority queue is similar, as shown in figure 5. The main difference is in how a message is represented. At the MCA level, the cell wishes to be able to see the components of the message. At the kernel level, the message is represented by the message ID. In the kernel, the contents of the message are found by using the message ID to index a message map. The message map, not shown in the figure, is a kernel implementation detail not visible at the MCA layer. If the message map were visible at the MCA layer, then it is likely that covert channels could be implemented using the message map. Keeping such implementation details invisible is important in the design of the kernel, and in the specifications and refinements used to build the kernel.

Software development is accomplished in SPECWARE by refining one specification into another. Refinement is a process in which all the elements of one specification are represented in terms of another [4]. As shown in figure 5, where dashed arrows represent refinement and solid arrows represent composition, the priority queue can be refined into a map of lists. Each element of the map corresponds to one of the system priorities. The use of a map makes it easy to search from the highest to the lowest priority in the priority queue. Each entry in the map is a list maintained in First In First Out (FIFO) order. Each message can be a structure of four elements (source strand ID, target strand ID, command word, and message segment list). At the cell level, once a message is delivered to a cell, the cell can directly reference the components of the message, such as the source strand ID, to determine which cell sent the message. When the delivery of the message is mapped down to the kernel specification, the kernel must first look up the message in the message map, and then make the components of the message directly visible to the receiving cell. When the cell is scheduled, the kernel in fact looks up the message and places the contents of the message on the stack of the scheduled cell. The cell can see that the contents are on its stack, but the cell knows nothing of the message map that is used by the kernel. The refinement specifies how the cell's view of messages is mapped down to the particular implementation chosen for the kernel.

6. MASK review

The review of the SPECWARE MASK kernel specification to C code implementation required a comparison between each sort, operation, and axiom of the MASK kernel specification and the corresponding C code implementation. The specifications are arranged in the three levels, separation, multiple cell abstraction, and MASK kernel. In addition, the C code implementation forms another level to be reviewed. The separation level is simple, describing separation in abstract terms. The multiple cell abstraction (MCA) level is complex, combining many small specifications into the MCA specification to detail the behavior of a process from the point of view of a cell. The MASK kernel specification is the most complex, detailing how the MASK kernel implements the MCA level.

In addition to these levels, there are also basic, shared, and kernel primitive specifications. The basic specifications are domain independent specifications such as map and list. These specifications can be thought of as the basic building blocks upon which any development might be built. The shared specifications are domain dependent and are used across multiple layers of the specification, usually in the MCA and MASK kernel layers. For example, much of the priority queue specification can be shared between the two layers. The MASK kernel layer incorporates the detail of how the priority queue is implemented, but the behavior of the priority queue in terms of scheduling algorithm is invisible at the MCA level. Finally the kernel primitive specifications are the domain dependent specifications upon which MASK was built. Examples of kernel primitive specifications are send and launch. The launch primitive describes how a strand is started, and the send primitive describes how messages are sent between strands.

6.1. Specification review—MCA

This review focused on the Multiple Cell Abstraction (MCA) specification. The MCA is a collection of single cell abstractions, where a single cell abstraction defines the

view of the world of a single cell. Therefore, the MCA embodies the collection of all cells in the system.

The main finding of the review at the MCA level was that the MCA should be defined in a functional manner. The first draft of the MCA specification defined the cell abstractions in terms of their properties only, rather than as functions. For example, prior to launching a strand of a cell, the properties state only that the resources required by that strand are visible (allocated) to that strand. When these properties were replaced by a functional definition, a function was specified to explicitly construct the lists of resources. Using functional definitions brought two important points to light:

- *Executability:* The functional definition of the MCA level primitives raises the possibility of having a functional model of MASK without the MASK implementation level specification. This can be useful in determining whether the kernel is robust enough to support the project objectives.
- *Totality:* Though not strictly required, all of the functional definitions were made total. While if it can be shown that a function is never called with certain arguments, then the function need not be specified on those arguments, having these partial functions makes the specification harder to review, since the question always arises "What will happen if the function is called with those arguments?".

6.2. Refinement review—MCA to MASK kernel

The MCA to MASK kernel refinement provides an avenue for representing sorts and operations at the cell level as sorts and operations at the kernel level. The actual refinement is built of three specifications and the relationships between these specifications. The specification `MCA-AS-MASK-KERNEL` is developed and acts as mediator between the `MCA` and `MASK-KERNEL` specifications. This development is done in large measure by writing cell operations in terms of kernel operations. Having a functional definition (with no partial functions) made the refinement from `MCA` to `MASK-KERNEL` easier to specify. In short, the less detailed functions at the MCA level are mapped to the more detailed functions at the MASK kernel level.

6.3. Specification review—separation

This review focused on the separation specification. The separation specification incorporates several operations and constraints on the interactions of cells. It remains strictly abstract, specifying properties and not functions. This is appropriate because it is intended to apply to many systems, not just MASK.

6.4. Refinement review—separation to MCA

The final aspect of the review was that of the refinement of Separation to Multiple Cell Abstraction. The components of this refinement, `SEPARATION-AS-MCA`, acts as the mediating specification between `SEPARATION` and `MCA`. The focus of this review was to ensure that each operation provided to cells by MASK maintains the separation property between cells. A major purpose of the MCA level is to make this refinement easier. Because the MCA level details the effect of each operation on the resources that are visible to cells, the separation property for each type of resource can be stated in the MCA level, and the maintenance of those properties can be checked.

6.5. MASK kernel specification to code—details

Several changes were identified in the `MASK-KERNEL` specification to code review. The change to using both total functions and functional specifications was found to help here, since it is much easier to review a functional definition against the code than the specification of properties. The functions in the `MASK-KERNEL` specification correspond one to one with functions in the C code. Some of the changes identified were:

- *Kernel Trace:* The specifications require a kernel trace, i.e. a log of the kernel calls made by the cells. The intent was that the trace be of assistance when the applications required debugging. The debugger could view the sequence of kernel activities, resulting in information about what was going on in the system before the error occurred. Future versions of MASK are envisioned in which the kernel trace can be retrieved by a system administrator to assist in remote debugging of systems. One project was resistant to putting in the kernel trace, because of the perceived performance hit. The kernel trace was put in, and proved useful in development of the applications hosted by the kernel. The lesson is that even real time, performance critical applications should have trace capabilities in the delivered system.
- *Kernel Tampering:* The code had several checks of the "this should never happen" variety. The specification did not have these, since in the mathematical world, such events are indeed impossible. However, MASK may be implemented on hardware that does not offer good protection against hardware faults. While tampering with MASK by the application code is not possible in the delivered system, such tampering is possible in the development environment, so these checks prove useful in the development of applications on MASK. The `MASK-KERNEL` specification was modified to reflect these error conditions.
- *Code Errors:* A few code errors were found. For example, it was discovered that one cell could pass interrupt handing responsibilities to another cell, forming a type of communication channel while the specification required that interrupt handling responsibilities for a given interrupt always remain within a single cell. Naturally, the code was updated to have this restriction.

7. Success story

The MASK kernel is now executing on board the AIM chip, which is being incorporated into many Department of Defense (DoD) projects, such as the KOV 20 key management system for the F22 fighter aircraft, the Defense Message Radio (DMR) for naval communications, and others. Based in part upon the MASK technology, the AIM chip has been certified for Type-1 cryptographic applications. MASK has also made clear the relevance of the SPECWARE technology to Information Security (INFOSEC) applications such as AIM. The use of the composition and refinement operations was crucial in our efforts to clearly state the separation property, and refine it into a usable kernel. MASK has proven efficient enough to support the applications of AIM to date. This includes being small enough to fit in the very limited on board ROM of the AIM chip.

8. Future work

Krenz [6] is an architecture that can provide higher grade security to COTS platforms such as NT, Linux, or Unix. The Krenz architecture builds on the separation concept developed under the MASK program. It extends the concept by requiring that a specified filter be applied to information flowing between any two cells.

The basic architectural concept of the Krenz is to permit COTS platforms to operate in the information processing modes specified by the organization, and to have the platform enforce the filters between the information processing modes. One conception of a COTS platform, from a security perspective, is depicted in figure 6. Within this figure, the COTS platform is pictured as a whirlwind of activity that is beyond the control of any serious security enforcement.

A COTS platform is pictured as *out of control* for the following reasons:

- It includes millions of lines of code. Such large bodies of code are beyond analysis.
- It is frequently compromised, as witnessed by the security advisories released by the Computer Emergency Response Team (CERT).
- It is frequently updated, with new versions of system features, patches to bugs, and even patches to address the CERT advisories.
- Applications on it are frequently updated.
- It is frequently exposed to the Internet, and the hackers and viruses that lurk there.

The first architectural concept is to bound the COTS platform. In figure 6, the COTS platform remains a whirlwind of activity, but the whirlwind is constrained inside a box comprised of several boundary components. These components regulate the interaction of the whirlwind with the world around it. Since they can be brought under control,

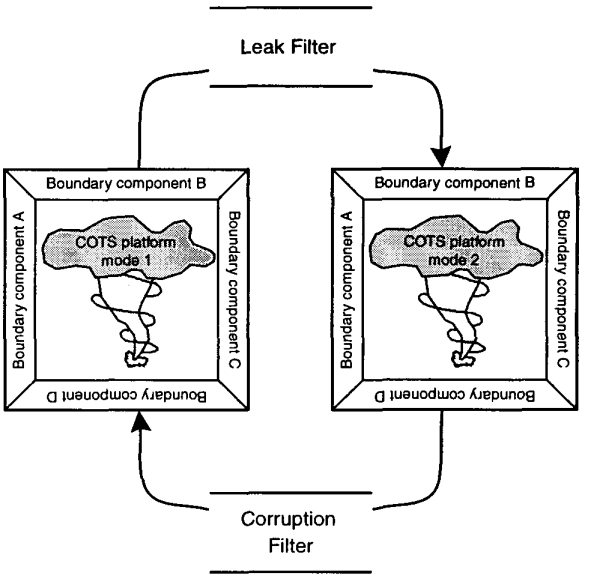


Figure 6. Information flow restrictions

assertions can now be made about the flow of information into and out of the whirlwind, but not about what happens to the information inside the whirlwind.

Given that the whirlwind can be contained, then it can be constrained to follow the organization's flow restrictions. If the whirlwinds were not contained, few assertions could be made about the flow of information between modes. The model of processing shown in figure 6 is often implemented by having two physically separate COTS platforms, on physically separate networks, with written policies concerning what can be transferred between the platforms. This is a valid implementation of the Krenz concept.

References

[1] D. Bell and L. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report MTR-2997, Mitre Corporation, March 1976.

[2] Common Criteria Project Sponsoring Organizations. *Common Criteria for Information Technology Security Evaluation, Version 2.1*, August 1999.

[3] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985.

[4] Kestrel Institute, 3260 Hillview Avenue, Palo Alto, California 94304. *Specware Language Manual*, 2.0.3 edition, March 1998.

[5] J. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, volume 15, Dec. 1981.

[6] P. White. High confidence CDSA program, final architecture report. Technical report, Motorola, 1999.