Presented at the 27th IEEE/AIAA Digital Avionics Systems Conference (DASC), St. Paul MN, October 2008.
Recipient of "Best in Session," "Best in Track" and "Best in Conference" awards.

The MILS Component Integration Approach to Secure Information Sharing

Carolyn Boettcher, Raytheon, El Segundo CA
Rance DeLong, LynuxWorks, San Jose CA
John Rushby, SRI International, Menlo Park CA
Wilmar Sifre, AFRL/RITB, Rome NY

Abstract

The US military has a vision of information superiority that requires secure and timely sharing of information between geographically separated platforms and users. Often, however, the producers and consumers of information, as well as the information itself, reside in different security domains, necessitating some form of Cross Domain Solution. A COTS marketplace of modular, high-assurance components with composable security properties would not only make this vision of cross-domain information sharing achievable, but could also help to make it much more affordable than is currently feasible. As part of the Air Force's Multiple Independent Levels of Security/Safety initiative, AFRL's multi-year High Assurance Middleware for Embedded Systems (HAMES) program is conducting research in integrating trusted components in such a way that the security properties of the system can be predicted.

MILS is characterized by a two-level approach to secure system design. At the policy level, a decomposition to a virtual architecture is performed while identifying the trusted components, the local policies and the communications channels. This is done in a way that minimizes complexity of trusted components and their policies. At the resource sharing level, implementation of components is considered, which includes the allocation of components to shared physical resources. MILS provides an implementation technology that enables virtual components of various types, and their intercommunication channels, to share physical resources without compromising the integrity of the policy level.

Security is seldom identified with a single, simple policy; the two-level approach of MILS was introduced as a rational way to organize the multiple cooperating components and sub-policies that realize a complete secure system. A MILS system needs to provide assurance that this design and implementation strategy and, in particular, the separate sub-policies of its components and the resource-sharing properties of its physical subsystems, compose to guarantee the security policy required of the overall system. This paper will describe the progress made so far in our research and some of the remaining challenges.

Introduction

MILS is a high-assurance architecture for secure information sharing that builds on and extends a long tradition of work on architectural approaches to security and safety. Its mechanisms are closely related to the "robust partitioning" employed for safety in Integrated Modular Avionics (IMA), and to the "separation kernels" employed in some secure systems. Indeed, early generations of MILS were identified as those that supported "Multiple Independent Levels of Security" (hence the acronym) on a separation kernel. More recently, the focus has evolved from individual secure systems to systems of systems, and current treatments of MILS extend its architecture to allow a layered approach to design and certification in which the assurance case for a system of MILS components can be assembled from those provided for its components.

This paper describes a modern treatment of MILS architectures that supports a component-based approach to design and assurance, and that encourages development of a COTS marketplace for evaluated MILS components. We begin, in the next section, by describing some of the historical evolution that led to the present version of MILS; we present our treatment of MILS in the section after that, and our conclusions in the final section.

Evolution of Secure System Architectures

Here we describe some of the changes in operational requirements, system architectures, and technological capabilities that led to the current MILS approach. Methods of specification and verification for security assurance evolved at the same time as security architectures and these had a strong influence on each other.

Computer security is concerned with system architectures that can allow untrusted application programs to access sensitive information yet not allow that information

to "leak" in ways that could make it available to unauthorized persons. The Anderson Panel Report of 1972 [4] was of seminal importance in framing this issue, and almost all treatments of computer security build on the foundation established by that report. The Anderson Panel stated that "the basic concept upon which multilevel secure computing systems can be based is that of *controlled sharing*. Explicit control must be established over each user's (programs) access to any system resource which is shared with any other user or (system) program" [4, volume 1, page 8]. The entity that exercises control is called a *reference monitor* and its implementation is referred to as a *reference validation mechanism* that "validates each reference to data or programs by any user (program) against a list of authorized types of reference for that user. . . Accompanying the concept of a reference monitor are other essential design requirements. They are

1. "The reference validation mechanism must be tamper proof.
2. "The reference validation mechanism must always be invoked.
3. "The reference validation mechanism must be small enough to be subject to analysis and tests, the completeness of which can be assured."

Volume 2 of the report elaborated the recommended development plan and introduced the idea of a *security kernel* [4, volume 2, page 24]. The objective for a security kernel design is "to integrate in one part of an operating system all security related functions. This is for the purpose of being able to protect all parts of the security mechanism, and to apply certification techniques to its design."

A crucial issue in the certification of a security kernel or other reference validation mechanism is a precise statement of the requirements against which it is to be "subject to analysis and tests, the completeness of which can be assured." In security, these requirements are referred to as a formal security model or policy, and the first and most influential of these was due to Bell and La Padula [6], who formulated the *simple security property* ("no read up") and the **-property* ("no write down").

It was originally thought that if these rules were enforced by the security kernel on all accesses by *subjects* to *objects*, where objects were identified with obvious data repositories such as files and memory segments, then (multilevel) security would be assured. It was soon discovered that this is not so: it is possible to leak information at high rates through covert *storage channels* that use the security kernel's own data structures as the paths for information flow. Numerous flows of this kind are present in the rules (based on those of Multics) that Bell and La Padula used to illustrate their model. An example (described in [17]) concerns the Change-Subject-Current-Security-Level rule: a subject with legitimate access to a high level object runs an untrusted program that extracts sensitive information and encodes this in the selection of low level objects to which it acquires (legitimate) read access; the program then releases access to the high level object, reduces its level to those of the low level objects, observes to which ones it has read access—and thereby recovers the high level information, which it may now write to a low level object.

Elimination of these storage channels requires analysis of information flow through the mechanisms of the reference validation mechanism itself. This led to the development of security models that account for information flow, such as that of Feiertag, Levitt, and Robinson [10] (and the more abstract model of Goguen and Meseguer [11]), and tools that could analyze formal specifications to verify the absence of such flows [9].

As real systems based on the precepts of the Anderson panel came to be developed, it was found that many functions needed to be exempted from the reference validation mechanism. In KSOS, for example, these included "support software to aid the day-to-day operation of the system (e.g., secure spoolers for line printer output, dump/restore programs, portions of the interface to a packet switched communications network etc.)" [7, page 365]. The Anderson Panel report stated that "if the reference validation is (or must be) suspended for some group of programs, then those programs must be considered part of the security apparatus, and be subject to the first and last requirement" (of the three listed earlier—i.e., they must be tamperproof and subject to strong assurance). The problem is that if these trusted processes (as they are known) are not subject to the security policy enforced by the security kernel, then what is the interpretation of security to which they are to be assured and how does it relate to the policy of the kernel?

Early treatments of computer security focused on multilevel security, anticipating that the main application would be to multiuser timesharing systems. However, most of the applications for the early security kernels were in cryptographic processing, message handling, and "guards" (i.e., filters and downgraders), which were employed in what would now be called "Cross Domain Solutions." Unfortunately, the security policy embedded in the security kernel was of little help in ensuring the true security of these applications. For example, the key issue in a cryptographic controller such as shown in Figure 1 is that information may flow from the red side to the black side via the crypto or the header bypass but it must not do so directly; however, all multilevel security policies are necessarily transitive and cannot express this requirement. The problem is even more acute

in the case of guards, where the basic flow is from high to low (i.e., in direct contradiction of the kernel's multilevel security policy), and a key part of the security assurance argument concerns the filtering function performed in the guard (e.g., removing specified fields from structured data, reducing precision or otherwise "fuzzing" other data, or checking for the absence of specified keywords).

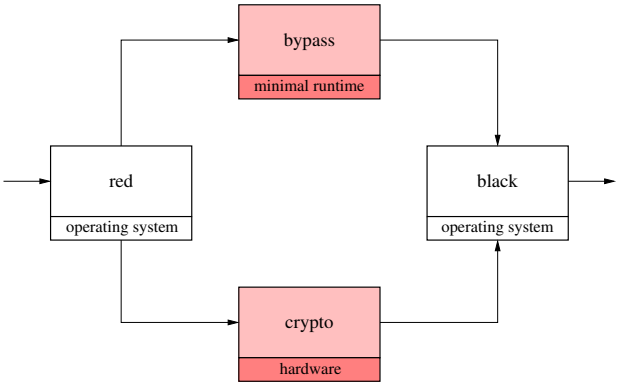

Figure 1. Information Flows in a Cryptographic Controller (shading indicates trusted, security-critical functionality)

Rushby [21] observed that security kernels have a dual responsibility: they must provide the basic protection mechanisms of an operating system (address space isolation, controlled access to privileged mode and so on) and they must enforce the system's security policy. He argued that this dual responsibility inevitably leads to complex implementations for which it is hard to provide assurance, and that it is unproductive in any case—since much of the argument for security will focus on functions such as authentication, cryptographic management, filtering, and so on, that are performed by processes outside the kernel whose local security requirements are different—but no less critical—than that of the kernel. Rushby proposed that a better security architecture would comprise a specialized operating system core, called a *separation kernel*, that focuses solely on the provision of isolated address spaces with controlled communications between them, while policy is enforced by trusted applications running in some of those address spaces. The conceptual cryptographic controller of Figure 1, for example, would be implemented by the architecture shown in Figure 2, where the separation kernel provides securely separated partitions and the intended (and no unintended) channels between them, while the bypass and the device driver for the cryptographic hardware perform security-critical functions specific to their purpose.

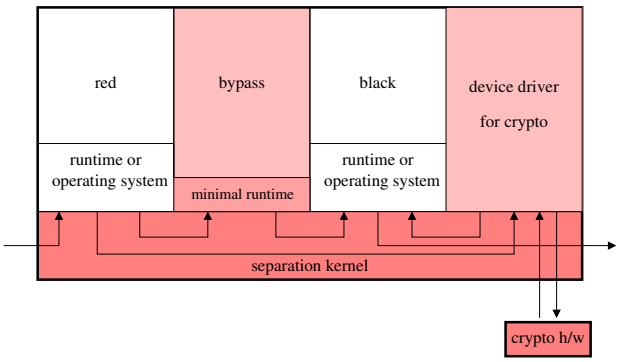

Figure 2. Cryptographic Controller Implementation Using Separation Kernel

Rushby had been influenced by the "Secure User Executive" (SUE) of a British system for end-to-end encryption [5], and these ideas in turn influenced the design of cryptographic platforms in the USA, including the "Advanced INFOSEC Machine" (AIM) and its "Mathematically Analyzed Separation Kernel" (MASK) [16]. Soon, however, separation kernels came to be considered for more complex systems.

By the late 1990s, high performance real time operating system kernels providing "robust partitioning" were becoming available for "Integrated Modular Avionics" (IMA) [1] and these had much in common with the idea of a separation kernel. With progress in hardware and kernel design, it became possible to perform tens or hundreds of thousands of partition switches per second, and this made it feasible to contemplate supporting more complex applications on a separation kernel. Many traditional operating system functions could be externalized, with untrusted separate copies running in each partition, while shared, trusted functions (such as file storage, or message routing) could run in their own dedicated partitions without excessive performance penalties. At the same time, it was becoming clear that many applications did not require true multilevel security, but rather the ability to process several levels of information simultaneously, but with little interaction between them. The security requirement for this class of applications became known as "Multiple Independent Levels of Security" (MILS) and was seen to be a good match for the newer class of system architectures based on separation kernels [25].

A conceptual design for a workstation based on this approach is shown in Figure 3. Single-level applications

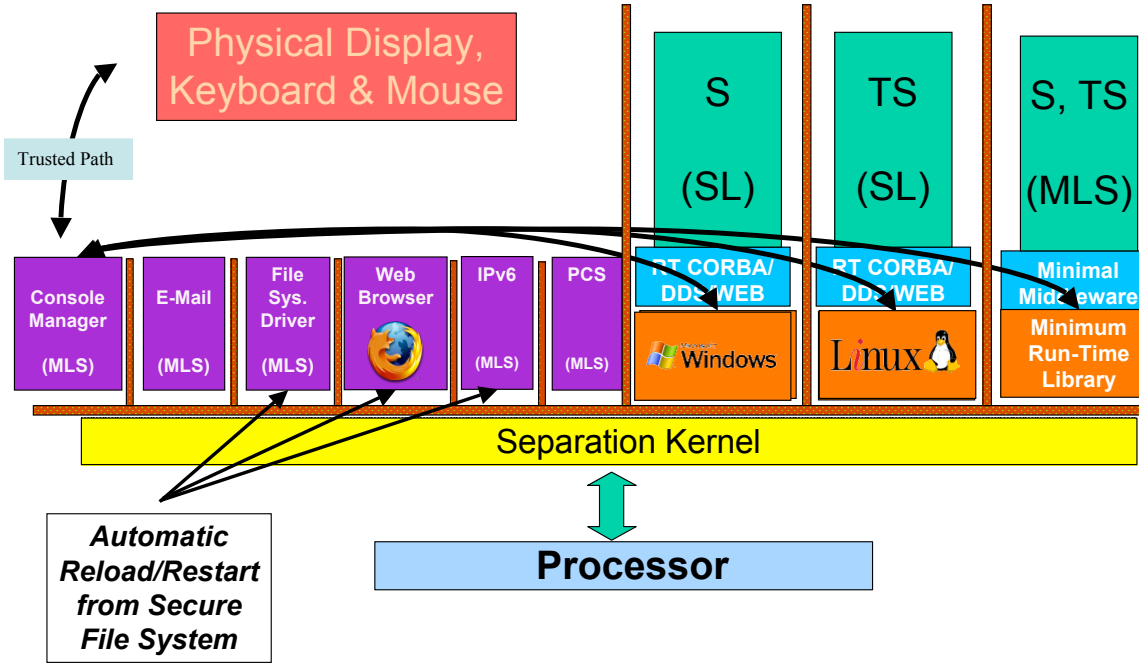

Figure 3. Conceptual Design for a MILS Workstation

hosted on Windows or Linux each run in their own partitions at different security levels, while a simple multilevel application is hosted on a trusted "minimal runtime" in a third partition. Several specialized trusted "middleware" functions such as email, and a web browser each run in their own dedicated partitions. In this formulation, MILS is viewed as a three-layered architecture: the processor hardware and separation kernel comprise the lowest layer, the trusted middleware is the next layer, and untrusted applications are the top layer. MILS-like architectures similar to this (but generally simpler, aimed at embedded applications [2], and with fewer multilevel middleware mechanisms) are proposed or under development for several large defense systems, such as the Air Force's F-35 Joint Strike Fighter, the Navy's DDG 1000 Zumwalt-class destroyer, and the Army's Future Combat Systems.

Part of the attraction of the MILS approach is that it supports a COTS-like business model: many suppliers can compete to deliver separation kernels, and the various trusted middleware components. To cultivate this market, the Air Force Research Laboratory and others are sponsoring development of *Protection Profiles* for selected components. Protection Profiles are elements in the Common Criteria approach to security assurance and evaluation [8]: they specialize the Criteria to a particular class of system or component, and are later refined to a specific "security target" and "target of evaluation." Protection Profiles are specific to a selected Evaluation Assurance Level (EAL), which can range from 1 (low) to 7 (the highest). A protection profile for separation kernels in environments requiring "high robustness" (which approximately corresponds to EAL 7) has recently been approved [14], and others are under development for (middleware) subsystems such as console, file system, network, and partitioning communications.

A crucial question is: how do we know all these protection profiles work together to ensure that a system based on MILS components is secure overall? This is obviously a matter of importance in a single MILS system such as that shown in Figure 3; it is even more pressing when we consider a distributed confederation of MILS components such as that shown in Figure 4, where multiple systems from multiple services interact to support complex joint training missions [13].

Our vision is for a modular or *compositional* approach to assurance, evaluation, and certification for MILS: that is, an approach in which the security evaluation of a system composed of previously evaluated components can largely be based on those previous evaluations, without requiring the entire system to be reconsidered from scratch.

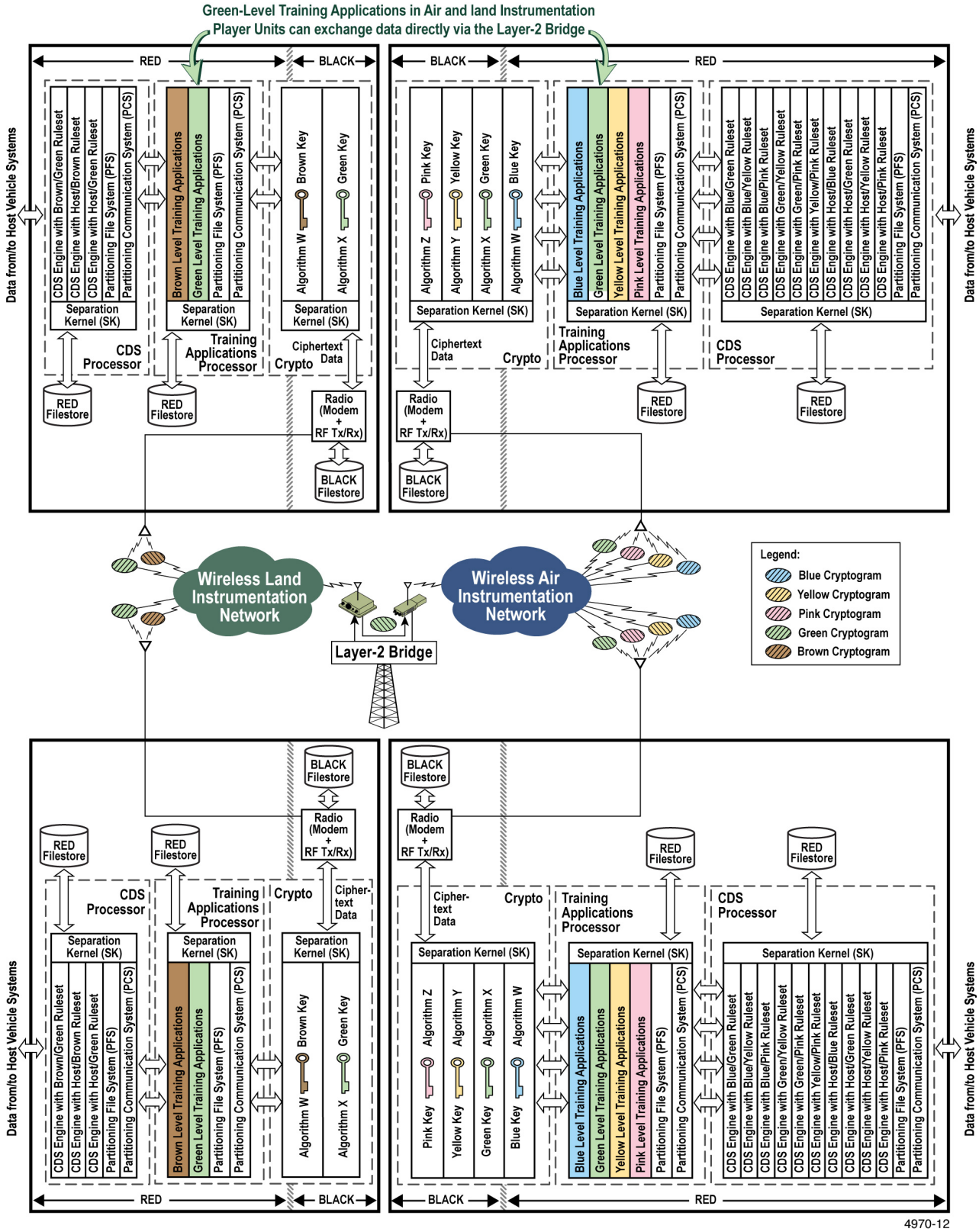

Figure 4. MILS Architecture to Support Joint Training Capability

This is a bold idea: it is not the way IMA systems are certified, for example [20]. We describe our approach in the next section.

Compositional Security Evaluation for MILS

The reason that compositional assurance and certification are challenging for any critical system property is that the assurance argument may not decompose along architectural lines. This raises the question: what is an architecture? Our response is that for systems that must ensure critical properties, a good architecture is one that simplifies the assurance argument. Hence, we propose that in secure system design, the first step should be to construct an architecture in which the security assurance argument will decompose along structural lines.

To achieve this, we must first recognize that there are two main issues in computer security.

Policy: this is the concept of security that the system is required to achieve and enforce. The security policy for a system is determined by regulation and doctrine, the context of its use, and the function it performs.

Shared resources: satisfying the security policy is often complicated by the need to share resources—since sharing introduces the possibility of interference between logically distinct components. Interference can include propagation of faults and leakage of information.

We maintain that the system architecture should be designed so that the argument that it fulfils its security policy is as simple and as strong as possible. We further maintain that shared resources are a conceptually different problem than policy and should be handled separately. The attraction of MILS is that it provides the tools to accomplish both of these goals. Reduced to essentials, separation kernels and other MILS components allow resources of various kinds (processors, files, network connections etc.) to be shared securely and inexpensively; we use this capability to synthesize distinct logical components so that the overall security argument is as simple as possible.

In more detail, the MILS approach to security advocates vigorous, *logical decomposition* as the first step in secure system design. The idea is to isolate security-critical functionality into components that are as small and simple as possible, and whose local security policies are likewise as simple as possible. The decomposition is logical, or virtual, in that it is unconcerned with the physical realization of components. Implementation of components is considered in a separate, second step, where it may be decided that some of the components identified in the first step should be implemented as physically distinct subsystems, while others may share physical subsystems. The rigorous separation enforced by MILS components guarantees that sharing of resources is undetectable to logically distinct components. We describe these two steps in the following subsections.

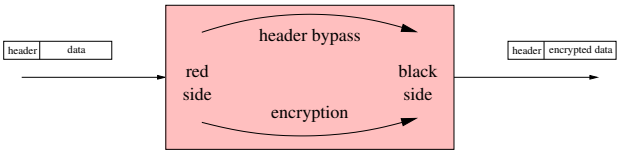

Figure 5. End-To-End Encryption Controller

MILS Policy Architecture

The top level architecture of a secure system should be designed to simplify the argument that it achieves its security policy. Hence, we call this a *Policy Architecture*.

Consider a very simple secure system: a controller for end-to-end encryption. Such a system takes in cleartext message packets from one ("red") network, encrypts their contents, and sends the encrypted packets out on another ("black") network. Packets comprise a header, which contains destination and other routing information, and data. Only the data part is encrypted because the switches in the black network have to read and process the headers so they can correctly route the packets to their destinations.

The internal structure of an encryption controller is sketched in Figure 5. There must be some software that handles the reception of packets from the red network; most likely, this will include a full handler for the communications protocol used on that network. This "red side" software will split the header information from the data and will send the header directly to the corresponding "black side" software while the data is sent via an encryption function. The black side software reassembles the header and encrypted data into a packet and sends it out on the black network. The security policy for this system is that unencrypted data must never go out over the black network. In practice the security policy would also include requirements that cryptographic keys never appear in the clear and other issues concerned with key management (see, for example [15]); we will ignore these to keep the exposition as simple and focused as possible.

If the encryption controller is implemented as traditional software running in a single processor as suggested by Figure 5, then satisfaction of the security policy depends on all of that software, and so it is shown shaded to indicate that it is trusted and must be provided with credible assurance. Furthermore, all the other software running in the processor, including the operating system and its

utilities, must be trusted and assured, as indicated in Figure 6.

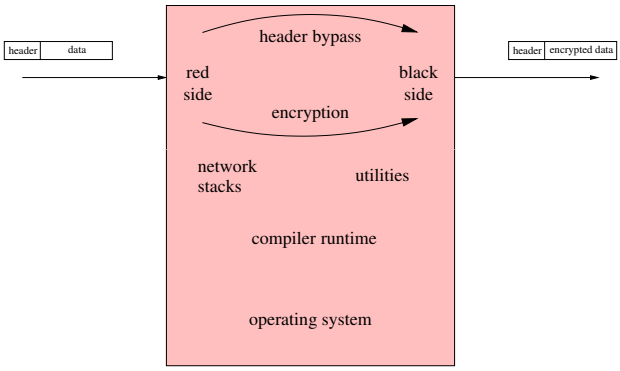

Figure 6. End-To-End Encryption Controller with Systems Software

Assurance that the system satisfies the security policy will require examination of all this software to be sure that there are no accidental or malicious mechanisms that could allow unencrypted data to reach the black side, where it could then be transmitted on the network. Malicious mechanisms could pass data through unexpected channels and could use clever encodings, so assurance would most likely have to specify exactly what each element of software is intended to perform, and to provide evidence that it does it correctly. Thus, assurance for a relatively simple property of a relatively simple system ends up requiring evidence for full correctness of an operating system, protocol stacks, and application software.

Now suppose that instead of a single monolithic implementation, we envisage the system as comprising four separate components connected by specific communications paths as shown previously in Figure 1.

Immediately, the assurance task becomes greatly simplified. There is no direct communication path between the red and black components. The *absence* of this direct path is a crucial element in the architecture represented by Figure 1. The only paths from red to black are through the crypto and the bypass components and we can derive subsidiary local security policies for these components: the crypto must encrypt everything that leaves on its outgoing channel, and the bypass must ensure that only information that "looks like" valid protocol headers is passed from red to black (and only at low bandwidth). The protocol handlers and other software in the red and black components can be completely untrusted.

The functionality of the bypass can be extremely simple (it may not even need to pass actual headers, just the destination and other essential information, since the true header will be constructed by the black protocol handler) and its assurance should be straightforward. The crypto component may be quite sophisticated, but it is likely to be a standard component whose assurance draws on long experience and well-attested capabilities.

Figure 1 is a policy architecture for the encryption controller. Construction of such an architecture is the first step in a MILS design; the goal should be to allocate functions to the components of a conceptually distributed architecture in such a way that the functionality of trusted components is as simple as possible, and the local security policies with respect to which they are trusted are also as simple as possible. Assurance for a policy architecture is accomplished by providing assurance that its trusted components satisfy their local policies, and that these local policies compose, in the context of the policy architecture, to achieve the overall security policy. For the policy architecture of Figure 1, the system assurance argument requires the following two elements.

Local policy assurance: the crypto and bypass implementations each satisfy their local security policies.

Integrating policy assurance: the local polices of the crypto and bypass in the context of the policy architecture (including its untrusted components) compose to satisfy the overall security policy.

To accomplish the second step, we need a more precise understanding of what is a policy architecture. Informally, it is a boxes-and-arrows diagram in which there are no channels for information flow other than those explicitly indicated by arrows. Arrows are unidirectional (bidirectional communication requires two arrows). Some boxes perform trusted functions for which a local security policy is specified. A more formal definition of policy architecture is presented elsewhere [24], but we sketch its basis here.

1. The *components* (boxes) in a MILS policy architecture are (possibly nondeterministic) state machines whose states are mappings from subsets of a global address space $\mathcal{A}$ to values from a global set $\mathcal{V}$.
2. The *local address space* of component P is a subset $\mathcal{A}_P$ of $\mathcal{A}$ that is disjoint from the local address spaces of all other components.
3. A *channel* (arrow) c with *source* component P and *destination* component $Q \neq P$ in a MILS policy architecture is identified with a *port* that is a subset $\mathcal{A}_c$ of the local address space of P that is disjoint from all other ports.
4. The transition relation $\mathcal{T}_p$ of component P has read access to its local address space plus all those ports

with destination P; it has write access to just its local address space.

Observe that channels are modeled by shared state; the restrictions on read and write access ensure these are unidirectional. Implicitly, channels are free of imperfections such as message loss. If imperfect channels are required, it is usual to interpose a new component that models the types of imperfections required, although these can also be modeled as transitions affecting the port of the source component.

5. The transition relation of the complete policy architecture is the asynchronous composition (i.e., the disjunction) of the transition relations of its components.

Computer science theory provides many techniques for formalizing the architectural model sketched above and tool-supported methods for reasoning about the composite properties of compositions of transition relations. It is one of the strengths of our approach that it does not impose strong constraints on how this is done. Trusted components of the architecture may be formally specified and assured by any means that is compatible with the underlying model that a component is a state machine, and the composition of trusted components may use any means that is consistent with asynchronous interleaving and the shared variable model of communication.

The formal model for policy architectures sketched above can be shown to be consistent with other similar models, such as intransitive noninterference [22] and GWVr2 [12]. This model provides the *assumptions* for the policy architecture level of MILS and the *requirements* for the resource-sharing level, which is described next.

MILS Resource Sharing

A MILS policy architecture is an abstract construction: its guiding principle is that the trusted components should have simple functionalities and simple security policies. To achieve this, we assume that splitting a larger component into several smaller subcomponents imposes no cost in acquisition or performance, and we likewise assume that communications between components impose no cost and generally are unidirectional, secure, and reliable.

The task of the MILS implementation level is to discharge not only the assumptions on which the security of the policy architecture depends, but also those about cost and performance. The latter concerns are addressed through *resource sharing*, and the former by doing this in a way that guarantees *separation*.

Separation means satisfaction of the model sketched in the previous subsection and its item 4 in particular. That item requires that the behavior of a component depends only on its local state and the ports of incoming communications channels, and that it modifies nothing but its own local state. Ports are the only interfaces to components in this model, and channels and their ports provide the only means for communication and interaction among components. If there were no channels, each component would function entirely independently (that was the original interpretation for the security policy of *separation*, while the richer security policy that includes communications was called *channel control* [21, 22]). As noted earlier, separation is similar to partitioning in avionics [23], and this provides the useful noun *partition* for the instantiation of a component within a shared resource.

To allocate the components and communications channels of a MILS policy architecture to shared resources, we first need to identify those that can be physically collocated and those that have similar functionality. We may then consider implementing those functionally similar collocated components as partitions in a shared resource that provides the functionality concerned. For example, several components that provide filesystem services could share the resources of a "partitioning filesystem," while components that present information to a human operator could share the screen area of a single "partitioning console subsystem," and a collection of channels could be routed through a shared wire or network using VPN-like capabilities provided by a "partitioning communication subsystem." Totally disparate components can share a processor partitioned by a separation kernel: each component is then implemented as a bespoke program running in its own partition.

There are several ways to ensure separation and a single system may employ more than one technique (an early example that used several is the Distributed Secure System [18]).

The most basic technique is *spatial separation*, which corresponds to the direct implementation of a policy architecture without resource sharing: each component is implemented in a physically separate resource and channels are implemented as dedicated point-to-point communications lines. Physical separation is seldom feasible for a complete architecture (and unidirectional physical communications often require special treatment, such as "data diodes"), but it can be an attractive option for certain components.

Temporal separation allows different components to share the same physical resource, but not at the same time. The resource is dedicated to one component for a period, then scrubbed clean and allocated to another component and so on. This approach is also known as "periods processing" and was used for mainframes in the 1960s and

later; in a MILS context, it is a useful option for workstations and CPU servers.

Cryptographic separation employs encryption and digital signatures or checksums to enforce read and write protection. It is difficult to perform operations on data protected in this way, so cryptographic separation is most useful when data needs merely to be stored or moved from one place to another—hence, it is particularly suitable for partitioning filesystems and communications.

Programs sharing a processor resource sometimes can be shown to satisfy the requirements for separation using *static program analysis* or other kinds of formal verification. Such analysis may be able to guarantee that no information flows from one program to another except through channels specified in the policy architecture. Analysis of this kind is feasible only when *all* programs that share the processor are available for examination beforehand and it is therefore unsuited to dynamic systems, or those that use proprietary software. However, this approach can be effective in limited environments such as smartcards.

When some of the programs sharing a processor resource are unknown or untrusted we can turn to a *separation kernel*. The *kernel* element in this name is intended to suggest that its functionality is similar to that of an operating system kernel, while the *separation* element identifies the security policy that it enforces—that is, provision of isolated partitions corresponding to each of the components in the MILS architecture concerned, and the communications channels that connect them.

Untrusted software that resides in the partitions of a separation kernel may contain malicious code that attacks other partitions or the separation kernel itself, or that conspires covertly to communicate data contrary to the policy architecture. The kernel of a commodity operating system usually cannot represent the security policy of separation and cannot provide adequate protection, still less assurance of protection, against this kind of attack. A separation kernel is therefore stripped of extraneous function and dedicated to providing just the protection and assurance needed to enforce (part of) a MILS policy architecture. Its limited function allows a separation kernel to be very small (a few thousand lines of code), to deliver high performance (hundreds of thousands of partition switches per second), and to come with strong assurance (e.g., EAL 7) that it achieves its purpose. A separation kernel is similar to the "partitioning kernels" used in integrated modular avionics (IMA), but is more aggressively minimized in order to achieve higher assurance (an avionics kernel will typically be upwards of ten thousand lines of code, while DO-178B Level A [19] approximately corresponds to EAL 5 [3]).

A separation kernel uses the protection mechanisms of its processor—i.e., its supervisor modes and memory management unit (MMU)—to create partitions whose client software is constrained to specified areas of memory. (It is interesting to observe that a kernel uses spatial separation for memory, and temporal separation for the CPU registers.) The environment perceived by the clients of a separation kernel may be a simulated copy of a full processor (a *virtual machine*), a simplified copy (a *paravirtual machine*), or an interface of the kind presented by conventional or real-time operating systems (e.g., ARINC 653 for avionics [1]). Full virtual machines allow untrusted partitions to run off-the-shelf software, including commodity operating systems such as Windows.

Until recently, true virtualization was expensive (in terms of performance and the amount of kernel code required) on some processor families; paravirtualization reduces the cost but requires modifications to client operating systems, which is generally feasible only for those whose source code is available. Innovations in processor design have made full virtualization affordable, but other infelicities (driven by the needs of the commodity marketplace) continue to pose difficulties. In particular, caches provide fairly high bandwidth channels for covert information flow (an untrusted partition at "high" security level empties the cache to signal a 1 and leaves it alone to signal a 0; a subsequent untrusted "low" partition can measure memory performance and estimate the bit value) and these are exacerbated in multicore designs where some of the caches are shared. Processor temperature (which can be driven up by intensive computation) and power states can also provide covert channels.

Memory-mapped I/O allows device registers to be allocated to specific partitions; the kernel can field external interrupts from devices and immediately route them to the relevant partition for handling.[1] A separation kernel is minimized using techniques such as this: all non-separation functions are expelled from the kernel and delegated to specific partitions. Some of these functions (e.g., device drivers, shared network stacks, sophisticated scheduling) may need to be trusted, but a separation kernel applies the MILS philosophy that it is better to create several simple functions, each responsible for a single aspect of security, than a monolith responsible for many.

In addition to enforcing the separation of partitions, the separation kernel also provides their inter-partition communication (IPC) channels as specified by the policy architecture. The IPC interface and mechanism may range from simple mailboxes to page mapping (swapping a re-

1 Devices that can initiate DMA transfers are problematic because their memory accesses bypass the protections of the MMU; forthcoming processor designs remedy this deficiency.

gion of memory from the address space of the source to that of the destination).

A separation kernel also is responsible for scheduling partitions for execution. Scheduling must be done in a way that minimizes covert channel bandwidth (an untrusted "high" security partition can indicate a 0 or 1 bit through its choice of when it relinquishes the CPU) while maximizing whatever measure of performance is important to the overall application (these measures are very different in embedded real-time vs. interactive applications, for example). Minimization of covert channels generally requires static scheduling, while performance often favors dynamic scheduling (e.g., rate monotonic or earliest deadline first); a combination is possible where groups of partitions with similar security attributes are given a static group schedule whose allocation to individual partitions may be determined dynamically.

The virtual or paravirtual machine interface presented by a separation kernel is attractive for untrusted partitions because it allows them to run commodity operating systems and software, but it is rather an austere foundation for the software of trusted partitions. Hence, these partitions will often employ a minimal runtime (MRT) library that provides functions for managing local memory (malloc etc.), scheduling, and accessing IPC. The MRT must generally be trusted and assured for full functional correctness.

Returning to the policy architecture of the encryption controller shown in Figure 1, we see that several implementation strategies are possible. We could use four separate processors connected by wires (spatial separation), or four separate partitions in a single processor shared using a separation kernel, or some combination of these. A plausible choice is for the crypto to be a self-contained hardware device, while red, black, and bypass share a single processor. The red and black components are untrusted, so we need to use a separation kernel (as opposed to program analysis) to ensure that they cannot conspire to communicate plaintext data directly from one to the other. The crypto device will need a device driver and other support software and this will be trusted software located in a partition of its own. We thus arrive at the implementation structure portrayed in Figure 2.

The untrusted red and black software reside in partitions that may contain arbitrary support software, such as a full runtime library or operating system; the trusted bypass software resides in a partition that provides a trusted minimal runtime; the trusted device drivers and support software for the crypto reside in a fourth partition, where the kernel will vector interrupts from the crypto device and also provide access to its device registers (indicated in Figure 2 by arrows between the crypto device and its device driver partition). Device drivers and network stacks for the incoming and outgoing networks are located in the red and black partitions, respectively.

The separation kernel provides the channels between red, bypass, black, and the device partition for the crypto (indicated in the Figure 2 by internal arrows). The separation kernel must ensure that these channels are truly unidirectional, provide exactly the geometry of connectivity indicated in the policy architecture of Figure 1, and interpret the ports correctly.

The assurance argument that an implementation based on Figure 2 is a secure realization of the policy architecture of Figure 1 is composed of the following two elements.

Individual resource separation assurance: the separation kernel implementation and, in general, those of any other resource sharing components that might be present, satisfy their separation policies.

Integrating resource-sharing assurance: the configuration of the separation kernel and, in general, those of any other resource sharing components that might be present, jointly enforce the policy architecture.

Assurance and evaluation for separation kernels has long been a research area and is moderately well-understood (though still a very challenging undertaking at the highest evaluation levels). Assurance and evaluation for other individual resource sharing components is less well researched; although they generally operate as middleware and can rely on the protections of the separation kernel, other resource sharing components may be larger than the kernel. Provided their protection profiles are coherent (e.g., have compatible threats, assumptions, policies), the integrating assurance argument seems fairly straightforward [24].

Conclusion

We have described an approach to secure systems design and implementation that supports compositional assurance and evaluation. Our approach is based on the MILS architecture, but whereas previous treatments of MILS have focused on the three layers comprising its realization within a single device (separation kernel, middleware, and applications), we focus on its two-level architecture for distributed systems and systems of systems.

The first or upper level is concerned with decomposing functional and security objectives for the system to yield a MILS *policy architecture* in which all security-critical functions are performed by trusted components that are as

small and as simple as possible. The trusted components enforce local security policies that work together, in the context provided by the policy architecture, to achieve the security policy of the overall system.

A MILS policy architecture is a "boxes and arrows" description in which simplicity of trusted mechanism is generally achieved by allocating data and functions of different sensitivities or of different domains to separate components. A policy architecture uses separate components and communications channels freely, without concern for their physical realization. Allocation of the idealized separate components and communications channels of a policy architecture to physical resources is undertaken as the second or lower level of a MILS system development. Components and communications that are separate at the upper level may share physical resources at the lower level. Shared resources introduce the possibility of interference between conceptually distinct components; interference can include propagation of faults and leakage of information. MILS eliminates this hazard by requiring that shared subsystems implement rigorous *separation* (similar to partitioning in avionics), which guarantees that the sharing of resources cannot be detected by conceptually distinct components. MILS provides trusted resource-sharing components, such as separation kernels, partitioned file systems, and partitioning communications systems that deliver the required guarantee of separation.

Assurance for a MILS system involves four steps:

1. Assurance that individual trusted policy components enforce their local policies,
2. Assurance that the individual trusted policy components, in the context of the policy architecture, compose to enforce the overall system policy,
3. Assurance for individual resource-sharing components,
4. Assurance that the individual resource-sharing components compose to enforce the policy architecture.

MILS Protection Profiles (PPs) encourage development of a commercial marketplace for individual resource sharing components; the major RTOS vendors are developing separation kernels to the SKPP [14], and it is anticipated that similar developments will follow for network and communications subsystems, file systems, and so on, when their PPs are published. These latter components are peers of the separation kernel in the architectural view (since they all partition some shared resource) but run as middleware on the separation kernel in the implementation view. All MILS PPs under current development are for resource sharing components; we believe that PPs for standardized policy components such as generic "guards" could encourage a lively commercial marketplace for Cross Domain Solutions.

Our vision for component-based assurance and evaluation is that commercial MILS components will be delivered with security evaluations corresponding to items 1 and 3 in the list above. Systems integrators can largely base their system assurance case in this pre-existing evidence and need develop only items 2 and 4 to support system evaluation. Realization of this vision requires that PPs for individual components are harmonized so that they compose coherently.

We believe this approach to compositional assurance and evaluation could extend beyond MILS and security to IMA and safety, and to other critical systems and properties. Realization of this ambitious vision probably requires reexamination of current standards-based approaches to certification, and a move towards approaches based on assurance cases.

References

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

[2] Alves-Foss, J., W. S. Harrison, P. Oman, and C. Taylor, 2006, The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2(3/4):239–247.

[3] Alves-Foss, J., B. Rinker, and C. Taylor, 2002, Towards common criteria certification for DO-178B compliant airborne software systems. Technical report, Center for Secure and Dependable Systems, University of Idaho.

[4] Anderson, J. P., 1972, Computer security technology planning study. Technical Report ESD-TR-73-51, US Air Force. (Two volumes, available at `http://seclab.cs.ucdavis.edu/projects/history/seminal.html`).

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

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

[7] Berson, T. A. and G. L. Barksdale Jr., 1979, KSOS—development methodology for a secure operating system. In *National Computer Conference*, volume 48, pp. 365–371. AFIPS Conference Proceedings.

[8] 2006/7, *Common Criteria for Information Technology Security Evaluation*. Version 3.1, available at `http://www.commoncriteriaportal.org/thecc.html`.

[9] Feiertag, R. J., 1980, A technique for proving specifications are multilevel secure. Technical Report CSL-109, Computer Science Laboratory, SRI International, Menlo Park, CA.

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

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

[12] Greve, D., M. Wilding, R. Richards, and W. Vanfleet. Formalizing security policies for dynamic and distributed systems. Unpublished, 2004.

[13] Hanz, D. and J. Rushby, 2008, Joint national training capability: MILS integration roadmap. Project report, SRI International, Menlo Park, CA.

[14] Information Assurance Directorate, National Security Agency, Fort George G. Meade, MD 20755-6000, 2007, *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*. Version 1.03.

[15] 2001, *Security Supplement to the Software Communications Architecture Specification*. Report MSRC-5000 SEC V1.1, available at `http://sca.jpeojtrs.mil/home.asp`.

[16] Martin, W., P. White, F. S. Taylor, and A. Goldberg, 2000, Formal construction of the mathematically analyzed separation kernel. In *Fifteenth IEEE Conference on Automated Software Engineering (ASE '00)*, pp. 133–142, Grenoble, France. IEEE Computer Society.

[17] Millen, J. K. and C. M. Cerniglia, 1983, Computer security models. Working Paper WP25068, Mitre Corporation, Bedford, MA.

[18] Randell, B. and J. Rushby, 2007, Distributed secure systems: Then and now. In *Proceedings of the Twenty-Third Annual Computer Security Applications Conference*, pp. 177–198, Miami Beach, FL. IEEE Computer Society. Invited "Classic Paper" presentation.

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

[20] Requirements and Technical Concepts for Aviation, Washington, DC, 2005, *DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*. Also issued as EUROCAE ED-124 (2007).

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

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

[23] Rushby, J., 1999, Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center. 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.

[24] Rushby, J., 2008, A formal model for MILS integration. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA.

[25] Vanfleet, W. M., J. A. Luke, R. W. Beckwith, C. Taylor, B. Calloni, and G. Uchenick, 2005, MILS: Architecture for high-assurance embedded computing. *Crosstalk*. Available at `http://www.stsc.hill.af.mil/crosstalk/2005/08/0508Vanfleet_etal.html`.