SRI International

Technical Report SRI-CSL-08-XX • February 2008

Separation and Integration in MILS (The MILS Constitution)

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

This report was not released at the time because I considered it unfinished. However, a copy seems to have "escaped" and as of January 2021 it has 33 citations, so I thought I may as well make it belatedly available myself.

This research was performed under a subcontract to Raytheon Corporation, sponsored by the US Air Force Research Laboratory.

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

This document is formatted for two-sided printing with a binding: even-numbered pages should appear on the left and odd-numbered pages on the right when opened as a double-page spread. This page is the back of the title page.

Abstract

We describe the MILS approach to design, construction, integration, and evaluation of secure systems. The crucial feature of the MILS approach is that it separates the problems of *enforcing security policy* from those of securely *sharing resources*. MILS design proceeds in two steps: first, we develop a logical *security policy architecture* in which the system is deconstructed into interacting components in such a way that the trusted components are as simple as possible; second, we allocate components of the policy architecture to resources that are securely shared through mechanisms for logical *separation*.

MILS identifies certain standard resources such as processors, networks, consoles, and file systems and publishes *protection profiles* for their logical separation; a COTS marketplace is developing that provides components evaluated to these profiles. Standard protection profiles and a marketplace for evaluated policy components (such as guards and filters) are also anticipated. Top-down design of a MILS system pays attention to existing protection profiles and strives to target these where appropriate. MILS construction can then incorporate COTS products evaluated to these protection profiles.

MILS integration takes COTS and bespoke policy components and allocates them to physical resources that may be shared using COTS and bespoke components for separation in a way that is faithful to the original policy architecture. Security assurance and evaluation in MILS are assembled in the same way. That is so say, MILS security assurance is *compositional*: assurance for an overall system is derived from that of its components, integrated according to the specific policy architecture and resource allocation of the system concerned.

Compositional design and assurance for a system property such as security is a radical innovation; we outline the justification for the MILS approach to accomplishing this.

Contents

1 Introduction **1**

2 Separation **2**
2.1 The Policy Level 3
2.1.1 Trusted Systems Software 6
2.1.2 Communications Ports 6
2.1.3 Unidirectional Communications and Control Channels 8
2.1.4 Imperfect Communications Channels 10
2.1.5 Policy-Level Architecture Diagrams and Their Interpretation 13
2.2 The Resource-Sharing Level 14
2.2.1 Techniques for Separation 16
2.2.2 Separation Kernels 17
2.2.3 Resource-Sharing Example 19
2.2.4 MILS Protection Profiles and Products 20

3 Integration **21**
3.1 Interactions among Policy Components 23
3.2 Interactions between Policy and Resource-Sharing Components 25
3.3 Interactions among Resource-Sharing Components 25

4 Conclusion **26**

List of Figures

1 End-To-End Encryption Controller 3
2 End-To-End Encryption Controller with Systems Software . . 4
3 Deconstructed Encryption Controller 5
4 Communications Ports 7
5 Unidirectional Low-to-High Channel 8
6 Low-to-High Data Channel with Control Channel 9
7 Trusted Components for Unidirectional Low-to-High Channel 10
8 Control Channels 11
9 Policy Architecture with Imperfect Channels 12
10 Imperfect Channel and Protocol Adjunct 13
11 Idealized and Securely Shared Communication Resources . . 15
12 Encryption Controller Implementation Using Separation Kernel 20

A Note on the Subtitle

I was struck by a simile used by Kevin Driscoll of Honeywell: he likened a system architecture to the U.S. Constitution. The Constitution is not a law: it tells you what laws you can make. In the same way, MILS is not a system design: it tells you what designs may be considered MILS.

The U.S. Constitution has other merits: it is short and inspirational. In several meetings on MILS it has become clear that not all participants share the same understanding of MILS—there is a need for a definitive, consensus account of MILS in a form that is short and, ideally, inspirational. Hence, this document; it is certainly short. Determination of other merits is left to the reader.

Acknowledgments

I owe a particular debt of gratitude to Rance DeLong who reintroduced me to computer security after I had been absent for a while, and acted as my friend and guide in the world of MILS. I am also grateful for the support and patience of Carolyn Boettcher, our project manager at Raytheon, and to Wilmar Safire and Dilia Rodriguez, our managers at the Air Force Research Laboratory, and their predecessors Jahn Luke and Todd Reinhardt.

Several participants in the Real Time and Embedded Systems forum of the Open Group (where MILS and its protection profiles are reviewed) helped form my understanding of MILS and provided critical review of presentations based on drafts of the material given here. I particularly benefited by input from Joseph Bergmann of the Open Group, Ben Calloni of Lockheed Martin, Michael McEvilley of Mitre, Gordon Uchenick of OIS, and Mark Vanfleet of NSA.

1 Introduction

Security has several interpretations and encompasses many subsidiary and constituent notions. For example, *multilevel secure* systems manage information of different classification levels where the security requirement is that information must not flow "downward" in classification level; on the other hand, *guard* systems are interposed between a highly classified information source and a more lowly classified destination precisely so that information can flow downward, provided certain data fields are reviewed, removed, or modified. Other systems must ensure that certain operations are performed only with proper *authorization*, while others are concerned with *separation of duties* (so that performing one action may remove the ability to perform certain others).

The precise interpretation of security for a given system is called its *security policy*. A secure system must not only satisfy its policy, it must be provided with strong assurance that it does so. Assurance is generally more credible and less expensive to develop when systems are small and simple, and when their policies are simple. For large, complex systems and policies, it is often effective to decompose the system into components (and possibly then into subcomponents, and so on recursively) that are smaller and simpler than the overall system, and whose local security policies are simpler than the overall one.

The MILS[1] 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 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. Shared subsystems introduce the possibility of interference between logically distinct components; interference can include propagation of faults and leakage of information. MILS addresses these concerns by requiring that shared subsystems implement rigorous *separation* (similar to *partitioning* in avionics), which guarantees that sharing of resources is undetectable to logically distinct components.

[1]MILS was originally an acronym for *Multiple Independent Levels of Security*, but is now best considered simply a name for the approach to secure systems described here.

MILS is characterized by this two-level approach to secure system design: a first or "upper" or, the term we prefer, *policy* level that decomposes the system into logical components, and a second, "lower," or *resource-sharing* level that allocates these components to physical subsystems. This approach differs from other security architectures (e.g., *security kernels*), where there is generally but a single level and policy is enforced by the same mechanisms that manage resource sharing. As noted earlier, 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 subpolicies that realize a complete secure system.

A second element to MILS is cultivation of a COTS market for components; policy components are likely to be specific to the particular system concerned (though generic "guards" could be attractive) but resource sharing components have broad utility. A MILS implementation is then an integration of trusted COTS resource sharing components supporting a combination of trusted and untrusted policy-level components. The former are likely to be small bespoke implementations, while the latter may employ commodity COTS software.

Of course, a MILS system needs to provide assurance that this design and implementation strategy and, in particular, the separate subpolicies of its logical components and the resource-sharing properties of its physical subsystems, "add up" or *compose* to guarantee the security policy required of the overall system. This is the problem of *integration assurance* for MILS.

The following sections describe the MILS approach to these issues in more detail. The two levels in the MILS approach to design are each concerned with "separation": the policy level decomposes or separates the overall system and its policy into simpler components and policies, while the resource-sharing level virtualizes or separates shared resources so that they provide secure implementation platforms for the components generated at the first level. Section 2 addresses these two forms of separation; Section 3 then focuses on the issue of integration and composition, while Section 4 summarizes and outlines future work.

2 Separation

The two steps in MILS development correspond to classical policy/mechanism separation. MILS takes this further and advocates separation or decomposition of policy into simpler subpolicies, because a secure system seldom requires a single, simple security policy. Separation of func-

tions and policies can result in trusted components that are simpler, and trusted with respect to simpler policies, than a monolithic system. Assurance is a dominant cost in secure system development, and simple components and policies can drastically reduce assurance costs. The "mechanisms" of MILS correspond to its resource-sharing level and separation is also employed here—in fact, it is the basic function of a resource-sharing component to virtualize its resource into separate subresources that operate independently. We explore these facets of separation in the following subsections.

2.1 The Policy Level

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.

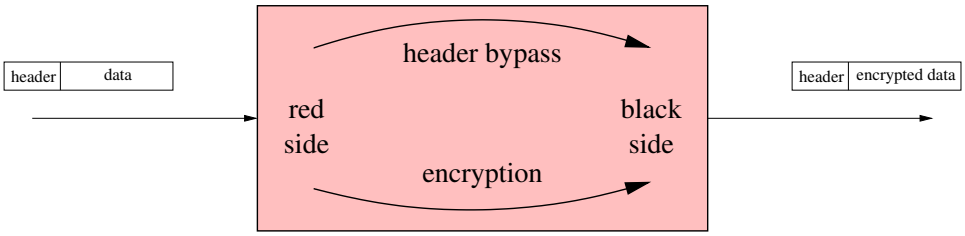

Figure 1: End-To-End Encryption Controller

The internal structure of an encryption controller is sketched in Figure 1. 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 [10]); 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 1, 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 2.

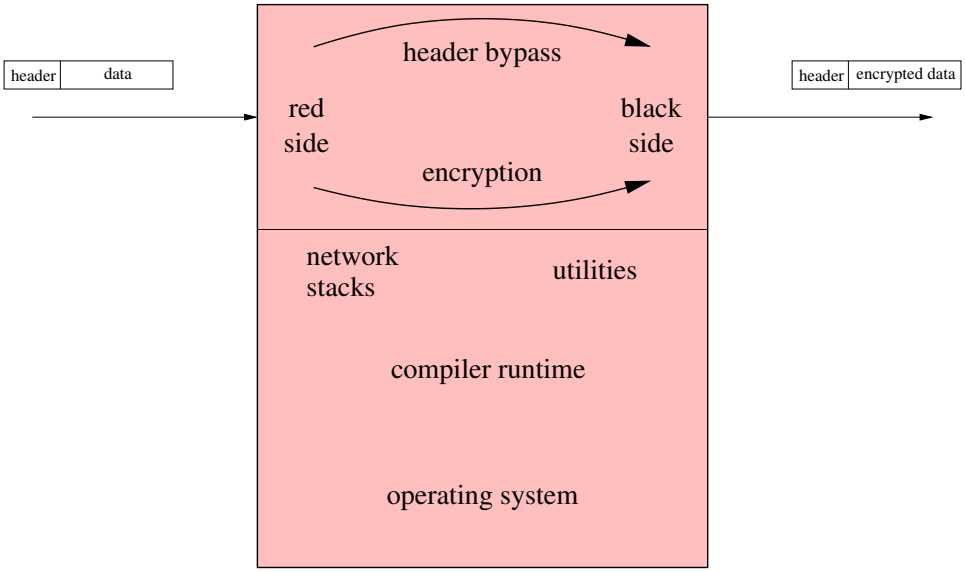

Figure 2: End-To-End Encryption Controller with Systems Software

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 in Figure 3.

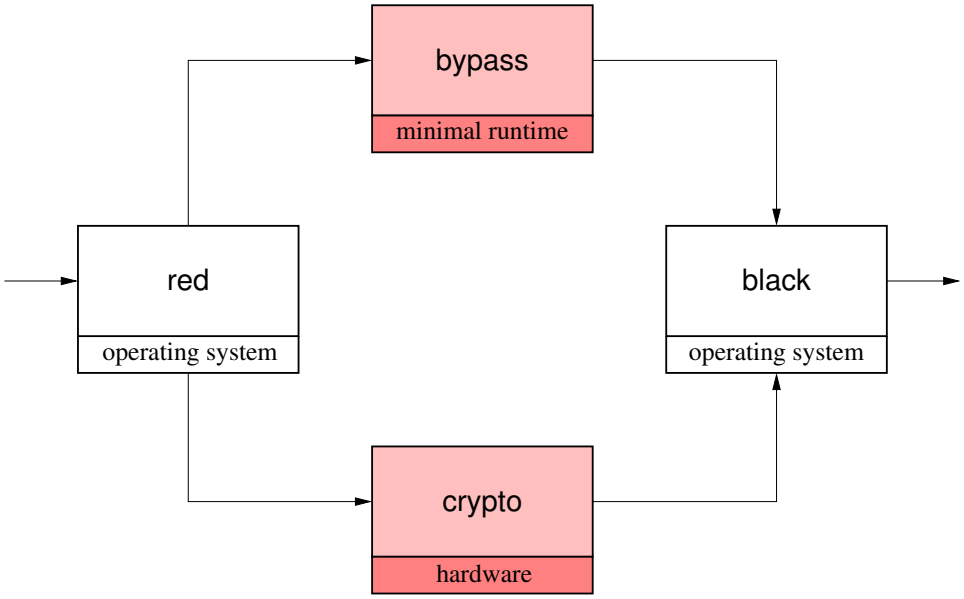

Figure 3: Deconstructed Encryption Controller

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 design indicated by Figure 3. The only paths from `red` to `black` are through the `crypto` and the `bypass` components and we can derive subsidiary 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. Hence, the `bypass` can be extremely simple, and its assurance would seem entirely feasible. 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 3 represents 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 security policies with respect to which they are trusted are also as simple as possible. In the second step of a MILS design, we will allocate some of these conceptually separate components to shared resources in a way that preserves the security assumptions of the policy architecture. The following subsections elaborate some of these assumptions, and their caveats.

2.1.1 Trusted Systems Software

The simplicity of the `bypass` and the standardized nature of the `crypto` will both be compromised if they depend on a complex operating system or other supporting software. Hence, we require that trusted components such as these depend only on very simple environments that can be provided with strong assurance. Typically, these environments will be the "bare metal" (either real, virtual, or paravirtual) of a well-known processor, or such bare metal plus a *minimal runtime* (MRT) that supplies, for example, a simple C library with `malloc` and other essentials. These elements are shown explicitly in Figure 3, and are shaded to indicate that they are trusted. Assurance for the bare metal and/or MRT will be with respect to a policy of functional correctness: this may be difficult and expensive but is a reusable artifact that exacts a one-time cost. Since the `red` and `black` components are untrusted, they may incorporate arbitrary software, including ordinary operating systems and utilities.

2.1.2 Communications Ports

The presence of a channel from one component to another needs to be interpreted carefully. We elaborate the interpretation of components and channels in Subsection 2.1.5, but first try to convey the intuition. The intuitive interpretation of a policy-level decomposition diagram such as Figure 3 is that the components (i.e., boxes) behave like separate computer systems, and the channels (i.e., arrows) are like physical point-to-point communications links, but are strictly unidirectional. Thus, a channel does not indicate arbitrary information flow to the state of a component, but the ability to read from or write into specific control registers, buffers or, the more abstract term that we prefer, *port*.

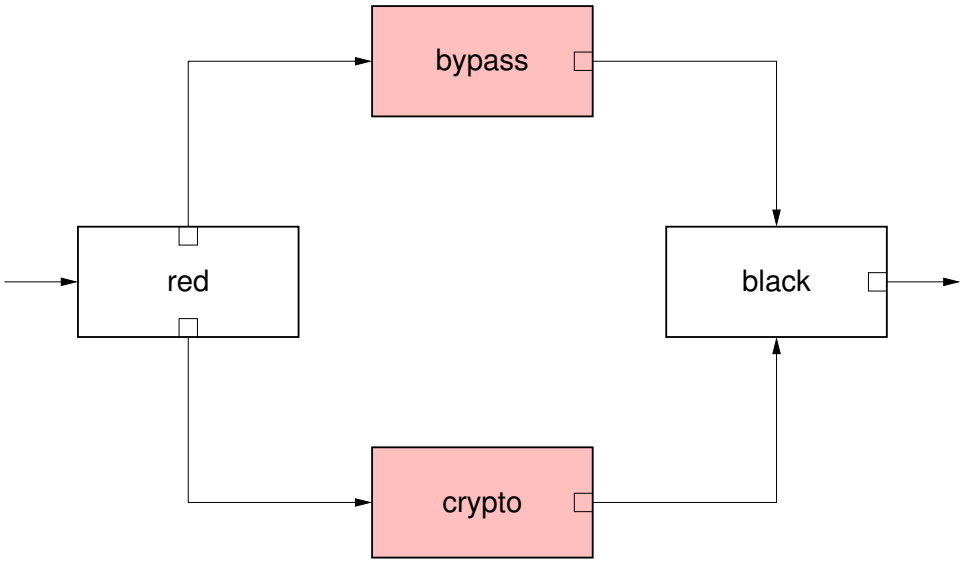

Figure 4: Communications Ports

Each channel is associated with a port, portrayed as a small square in Figure 4. A port is a read-only extension to the state of the component that is the destination of the channel; the port can be both read and written by the component that is the source of the channel. (It is obviously possible to create a dual interpretation where the port is located in the state of the destination component and functions as a write-only extension to the state of the source; it is a matter of personal preference exactly how this is done, but the arrangement chosen has the advantage that it is possible for the source to perform operations such as incrementing the value of a variable in a port.)

The requirement for ports, or some other localization of the interpretation of a channel is obvious: if the channel from `bypass` to `black` were interpreted to allow `black` to read anywhere in the memory of `bypass`, then it could read the temporary buffers holding information just received from `red`, thereby bypassing the `bypass` and rendering it unable to perform its trusted function.

2.1.3 Unidirectional Communications and Control Channels

Just as the absence of communications channels is often crucial to policy architectures, so is their strictly unidirectional character. For example, "cross-domain solutions" often require low-to-high flows, such as that shown in isolated form in Figure 5. We need to be sure that any implementation of this channel truly is unidirectional, even when the untrusted source and destination components do their best to subvert it (subversion that allows unmediated `high` to `low` flows is obviously unsecure). As this is a concern about implementation, a case can be made that it should be addressed at the resource-sharing level of MILS, and that the policy level should focus on achieving security with logically separated components and idealized unidirectional communications.

Figure 5: Unidirectional Low-to-High Channel

The problem is that certain implementation strategies require introduction of a trusted policy-level component and this should be recognized in the policy architecture. If `low` and `high` are partitions in a shared processor and the data channel is implemented in its separation kernel, then assurance that it is unidirectional will be provided with the kernel and does not obtrude to the policy level. But if the channel uses external wires, then we must either provide assurance for the corresponding software and hardware drivers, thereby elevating the source and destination components to trusted status and inviting their further decomposition, or else we must interpose a trusted "data diode" in the wire. Suitable data diode devices have been developed (e.g., using optical transducers [14]). These implementation strategies do obtrude to the policy level and suggest that, as with many system design problems, there may be some iteration back and forth between the policy and resource-sharing levels during evolution of a MILS design.

Unidirectional channels raise another issue that is definitely the province of the policy level. This is the possible need to introduce control channels in the opposite direction to data channels. In Figure 5, for example, the `low` component may supply data to `high` at a rate faster than the latter can process it. Thus, `high` must have some means to signal to `low` when it is ready and when it is not ready to receive data, or else the channel must

be allowed to become lossy, or worse (e.g., prone to data corruption). In a weak implementation, the `low` component might simply overwrite data in the port of the channel to `high`; if the timing is unlucky, `high` could then read corrupted data (e.g., part of one transmission, incompletely overwritten by a later one that is still in progress). A more sophisticated channel would implement the port as a wait-free, lock-free atomic register (e.g., Simpson's four-slot construction [24]): this guarantees that `high` always receives the most recent complete data sent by `low`, but data will be lost if `low` sends it at a rate faster than `high` can handle. This can be acceptable when the data is, for example, a stream of constantly updated sensor samples (provided these are absolute values rather than deltas on previous values—lost samples introduce permanent value errors in the latter case). To guarantee no loss of data we must either synchronize execution of the `low` and `high` components (e.g., a time-triggered implementation), so the former sends and the latter receives and processes some fixed amount of data in each "frame,"[2] or we must provide a control channel in the reverse direction as shown in Figure 6, where we use a dashed arrow to indicate the control channel.

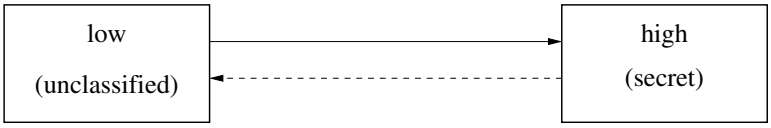

Figure 6: Low-to-High Data Channel with Control Channel

An unmediated control channel can become a path for unsecure information flow. To limit this, we need a trusted filter in the control channel, and this probably needs to correlate acknowledgments against transmissions, so it will also need access to the low-to-high channel as shown in Figure 7. As that figure suggests, the monitor for the control channel and a diode for the data channel may be available in a single component, such as the NRL "Network Pump" [11]. Note that in some policy architectures, ensuring that control channels also are unidirectional may be a significant concern.

The need for data diodes to ensure unidirectional channels is an implementation-level issue that rises to the policy level; the need for control channels and their possible mediation is, however, a policy-level concern. Thus, for the encryption controller of Figure 8, it is likely that each requires a corresponding control channel and these are shown in the more complete

[2]The synchronous approach is often preferred in embedded systems, because control channels can provide new paths for failure propagation [12].

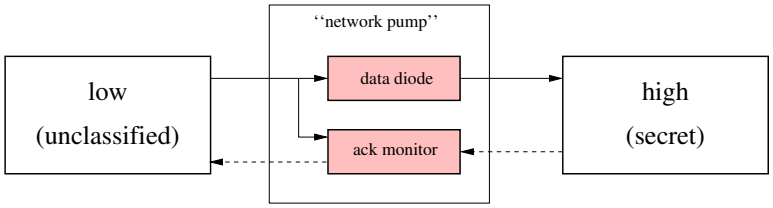

Figure 7: Trusted Components for Unidirectional Low-to-High Channel

policy architecture of Figure 8 (for simplicity, we have omitted explicit depiction of the ports). We anticipate that the implementation will ensure all the channels are unidirectional; hence, no new policy-level components are needed to ensure this.

The `bypass` and `crypto` are trusted mediators in the information flows between `red` and `black`; thus, the presence of control channels does not require new mediators, but these existing ones must be cognizant of the new channels. Note the importance of the assumption that these channels are unidirectional. If the control channels were not unidirectional, they could subvert the architecture: for example, the `bypass` might not monitor its control channels for "reverse flow" and might allow unmediated `red` to `black` information flow via this unanticipated route. Suitably enhanced `bypass` and `crypto` components could cope with perverse control channels, but this would have to be considered in their local security policies. Hence, security of the policy architecture of Figure 8, and the precise local policies of its trusted components, is contingent on assumptions about the unidirectional character of the implementations of its control channels.

2.1.4 Imperfect Communications Channels

Figures 3 and 8 indicate only the presence or absence of channels between components, and their directions; they do not indicate their properties, such as whether they are susceptible to loss, error, tampering, or eavesdropping. By default, channels are assumed to be free of these defects and it is the responsibility of the implementation level to discharge this assumption. However, some policy architectures may not require such strong assumptions, and others may prefer to deal with imperfections at the architecture level (rather as the assumption of unidirectional channels can be discharged at either the policy or implementation level). Hence, each channel in a policy architecture should be annotated with its assumed properties if these are other than "perfection."

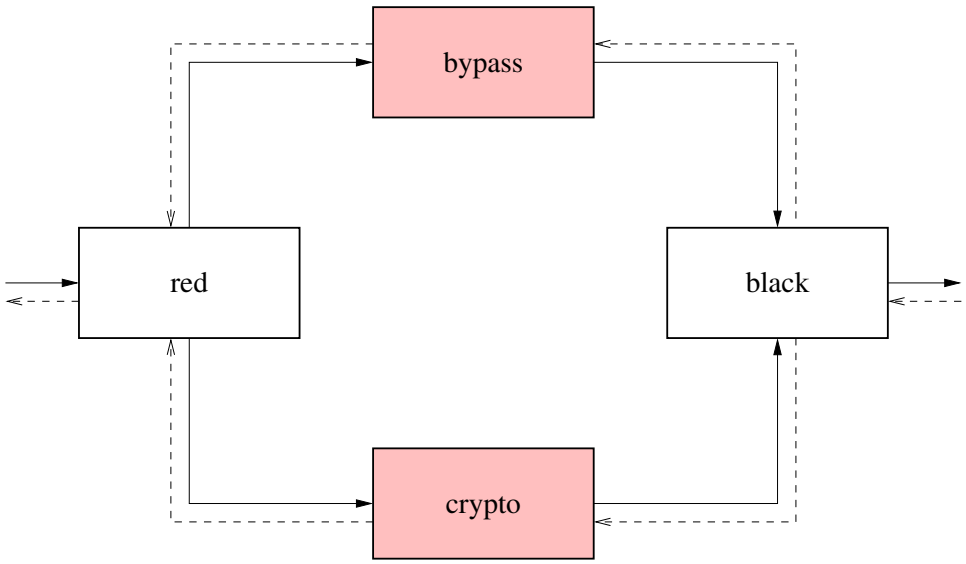

Figure 8: Control Channels

These annotations are necessary because the properties of imperfect channels can affect the policies and implementations of components—and even the feasibility of the whole design. For example, the entire rationale for Figure 3 is vitiated if the channel from `red` to `crypto` is vulnerable to eavesdropping. It might seem that eavesdropping on the `red`-to-`bypass` channel is less threatening since the headers will be revealed anyway in the output from `black`—but this neglects the fact that `red` is untrusted and cannot therefore be relied on to send only headers and not data over the channel to the `bypass`. We conclude that these channels must not be vulnerable to eavesdropping (an assumption that must be discharged at the implementation level, or through the context of the system's deployment); on the other hand, it does no harm if the channels from `bypass` and `crypto` to `black` are subject to eavesdropping.

A different issue arises if the channels between `red` and `bypass` are unreliable ones that can lose, change, duplicate, or reorder headers or their control information. These attributes do not directly compromise the security policy, but they would reduce the functional effectiveness of the device (since headers would no longer be reliably associated with their bodies). Hence, we might introduce a protocol on the `red`-to-`bypass` channel to ensure reliable communication, as portrayed in Figure 9 (there would probably need to be corresponding protocols on other channels, too).

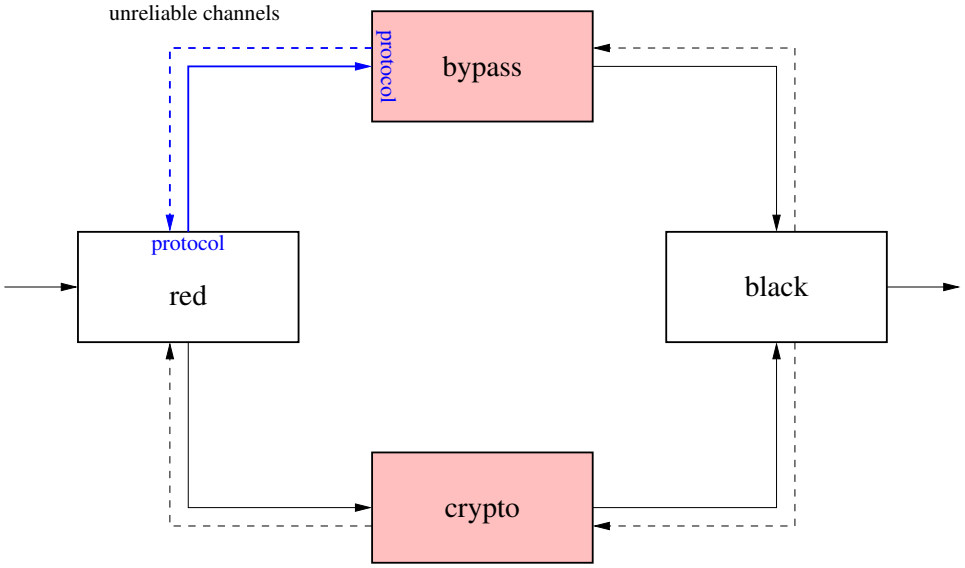

Figure 9: Policy Architecture with Imperfect Channels

A new problem is that the `bypass` now contains software for managing this protocol and is no longer the minimally simple component desired for credible assurance. The available design choices are either to push this issue down to the implementation level, or to offload the `bypass` protocol handler to a separate `adjunct` as shown in Figure 10.[3] Note that the "local" channels between the `adjunct` and the `bypass` are assumed to be reliable: such error-free channels are feasible when the components concerned are, for example, implemented in partitions of the same processor, and the channels are memory-to-memory transfers performed by the separation kernel.

[3]Note that Figure 10 shows the `red` protocol handler remaining in that component rather than being offloaded to a separate adjunct. This is because both `red` and its protocol handler are untrusted (with respect to security). Note, however, that although untrusted functionalities may be combined in a single component, trusted ones should not be. Thus, if the `bypass adjunct` were trusted (for some policy), it would still be sensible to separate it from the trusted `bypass` as these two components would be trusted for different purposes and the assurance of two simple components with respect to their individual policies will be more credible than that for a more complicated single component with multiple policies.

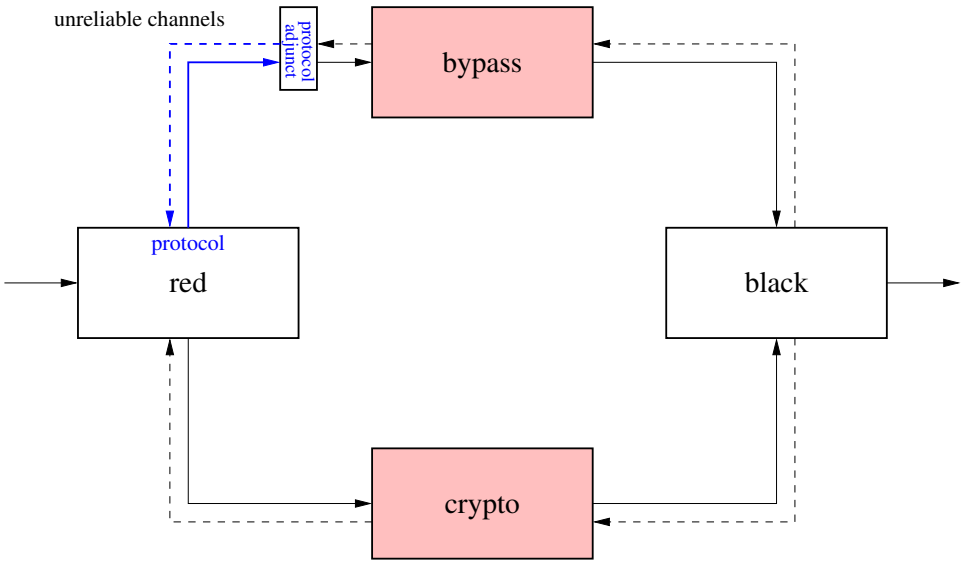

Figure 10: Imperfect Channel and Protocol Adjunct

2.1.5 Policy-Level Architecture Diagrams and Their Interpretation

The properties of a specific MILS system derive from the policies enforced by its trusted components and the context established by its architecture. A MILS policy architecture diagram such as Figure 4 has a precise interpretation. The formal definition of this interpretation is deferred to another document [5], 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.

   Read access and write access are defined in [5]; read access is particularly tricky to specify correctly (see, for example, [22, Section 2.1]).

   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 architecture is the asynchronous composition (i.e., the disjunction) of the transition relations of its components.

The formal security policy model sketched above can be shown to be consistent with other similar models, such as intransitive noninterference [22] and GWVr2 [7]. This model provides the *assumptions* for the policy architecture level of MILS and the *requirements* for the resource-sharing level, which is described next.

2.2 The Resource-Sharing Level

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 are free and generally unidirectional, secure, and reliable (caveats were discussed in Sections 2.1.3 and 2.1.4).

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 Section 2.1.5 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* [19, 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.

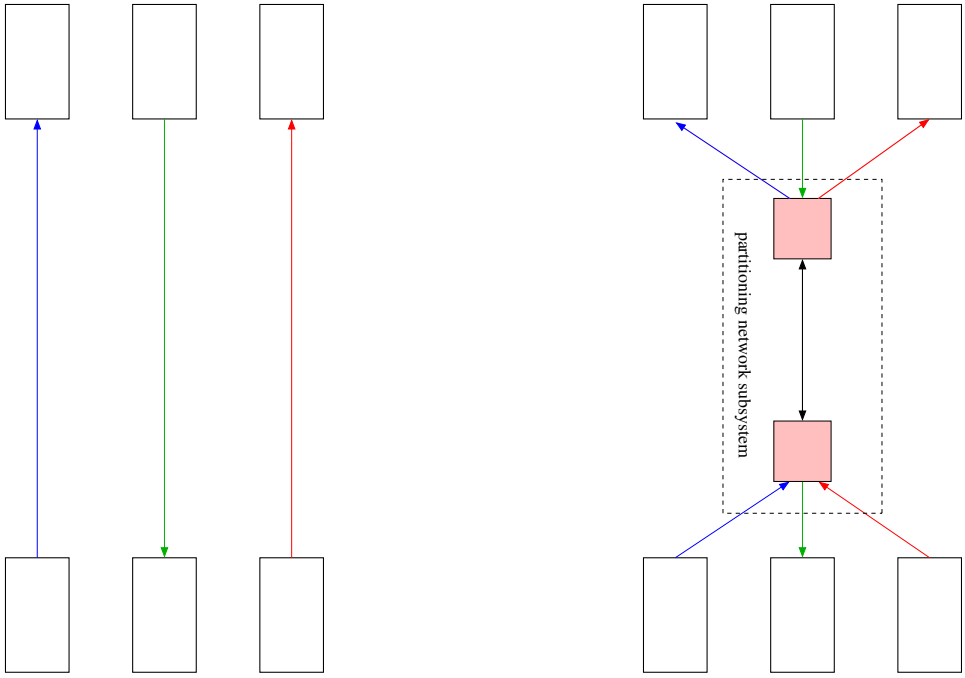

Figure 11: Idealized and Securely Shared Communication Resources

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." It is not only components and their interconnections that can share resources: a collection of channels could be routed

through a shared wire or network using VPN-like capabilities provided by a "partitioning communication or network subsystem" as suggested in Figure 11. 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.

2.2.1 Techniques for Separation

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 [15]).

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, 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.

2.2.2 Separation Kernels

A separation kernel is generally used in cases where it is impossible to analyze the software that will reside in some of its partitions. Such untrusted software 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 this kind of security policy 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 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 (an avionics kernel will typically be upwards of ten thousand lines of code).

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 [2]). 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 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.[4] 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 region 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.

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

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.

2.2.3 Resource-Sharing Example

Returning to the policy architecture of the encryption controller shown in Figure 4, 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 12 (control channels are omitted in the interests of readability).

The untrusted `red` and `black` software resides 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 12 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 12 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 4, and interpret the ports correctly.

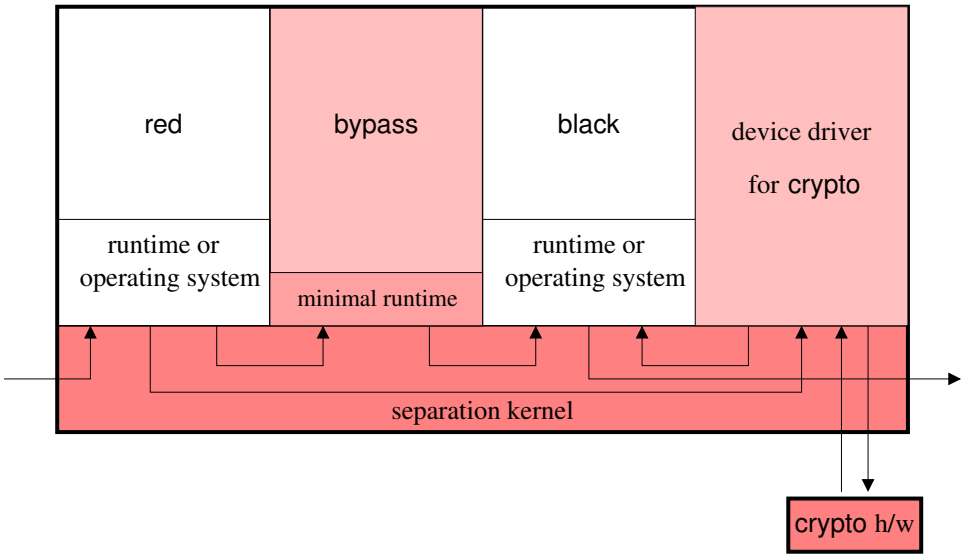

Figure 12: Encryption Controller Implementation Using Separation Kernel

2.2.4 MILS Protection Profiles and Products

Pure top-down functional and policy decomposition might yield few components sufficiently similar to share any resource but a partitioned processor; however, a repertoire of partitioned resources available off the shelf would provide an incentive to adjust top-down decomposition to target available products, where these are suitable.

Several DoD programs are working in concert to encourage development of broadly useful off the shelf components and the growth of a commercial marketplace for these. The means of encouragement is through development and approval of *Protection Profiles*[5] for a useful variety of shared resources such as the partitioned filesystems, console subsystems, communication and network subsystems, and separation kernels alluded to earlier. A protection profile for high robustness separation kernels has already been approved [9].

Figure 11 sketched a partitioning network subsystem in which secure multiplexing/demultiplexing of many communication channels onto a single network is performed by the trusted components of a partitioning network subsystem (probably using cryptographic separation). In practice, these

[5]The *Common Criteria for Information Technology Security Evaluation* [4] specialize their general requirements to specific classes of systems and components through *Protection Profiles* (PP) to *Security Targets* (ST), and finally to *Targets of Evaluation* (TOE).

components are most likely to be implemented as trusted software that runs in one partition of a separation kernel.

A partitioning network subsystem allows a network resource to be shared just as a separation kernel allows a processor resource to be shared; in this sense the network subsystem is a first-class MILS resource-sharing component, just like a separation kernel. On the other hand, the network subsystem is trusted software that runs in a partition provided by a separation kernel, and in this sense it is subordinate to the kernel. Some discussions of MILS [26] refer to components such as a trusted network subsystem as MILS *middleware*, as distinct from the separation kernel below it and the application software above it. In my view, the middleware sobriquet is perfectly acceptable, provided it is understood that this refers to an implementation strategy, and that the logical rôle of a network subsystem is identical to that of a separation kernel: each is a component at the resource-sharing level of MILS, responsible for sharing a particular kind of resource.

Implementation strategies for MILS resource sharing components may select from a range of options. For example, a secure filesystem could be implemented as a middleware service running in one of the partitions in a separation kernel (possibly structured internally using analytic separation), or as a standalone component (possibly using a separation kernel) dedicated to that function . It is because of this range of implementation options that we insist that the policy architecture is the interface between the upper and lower levels of a MILS design, and that the responsibility of the lower level is the sharing of *logical* resources. If a resource such as a filesystem is actually synthesized from still lower resources (e.g., a separation kernel, and a disk shared with cryptographic separation), then the MILS approach can be applied recursively to this subsystem [18].

3 Integration

The MILS approach to design proceeds in two stages: policy-level decomposition followed by implementation of the resulting abstract policy architecture on concrete, generally shared, resources. In principle, each system development could proceed in a purely top-down fashion, developing its own policy components and its own shared resources. The costs of such system developments will be considerable, particularly when high levels of assurance are required, so reuse becomes very attractive—and this is cultivated in MILS through development and approval of protection profiles and products evaluated to these. Currently, these are all at the resource-sharing level,

but protection profiles for policy-level components such as a generic guard, network pump, and authentication services seem entirely feasible.

Given protection profiles for a variety of MILS components, and a COTS marketplace for evaluated products compliant to these, most MILS system developments will deviate from a pure top-down approach to one that attempts to target existing components where these are suitable. The implementation of a MILS policy architecture is then constructed by integrating some bespoke components and some existing components. The developers will hope to assemble the assurance and evaluation argument for the full system similarly—by integrating the separate arguments for its bespoke components and those of its pre-evaluated components.

This is an instance of *compositional assurance*, and it is a radical step for any certification or evaluation regime. Most regimes evaluate only complete systems (the FAA, for example, certifies only airplanes, engines, and propellers) and provide no way to evaluate a component apart from its deployment in a specific system, and no way to assemble component or subsystem evaluations to provide (much of) the evaluation of a larger system.

The reason for this is that failures of complex systems, and often the most dangerous failures, are seldom due to faults in individual components but to flaws in the *interactions* of several (individually "correct") components. Typically, an unanticipated event provokes unanticipated and undesired interactions (see, for example, almost any accident report, such as that for Ariane Flight 501 [16]). By insisting on a complete system, the evaluators can examine possible interactions in the specific context of the given system. In contrast, component-level evaluations, and compositions of these, have to consider all possible interactions in configurations as yet unanticipated.

Now, by a happy coincidence, security is all about eliminating undesired interactions, and the genius of MILS is that it focuses on ensuring this precisely at the level of *logical* components. Thus, if we have a MILS policy architecture in which, for example, a guard is interposed in a channel from a component at a high security level to one at a lower level, we can be sure that any MILS-compliant system will ensure that this guard is in place and there are no other channels nor means of interaction between the high and low components—and this will remain true no matter what (MILS-compliant) resource sharing is employed and no matter what the larger (MILS-compliant) context is for this three-component subsystem.

A MILS system has two kinds of components, corresponding to the two levels of a MILS system development. These are *policy* components such as the `bypass` and `crypto` in the encryption controller example, and *resource-sharing* components such as separation kernels and partitioning communi-

cations and filesystems. When we compose a system from these two kinds of components, we must consider the three kinds of pairwise interactions among them: policy/policy, policy/resource sharing, and resource sharing/resource sharing. The following subsections consider these in turn.

3.1 Interactions among Policy Components

A MILS policy architecture diagram such as Figure 4 describes only the geometry of interactions among its components; a full policy architecture requires specifications for the local security policies of its trusted components, and specification of the overall security policy that the system is required to satisfy.

A subsequent document will provide a worked example, but the informal local security policy of the `crypto` is that everything that leaves its output port is encrypted, and the informal local policy of the `bypass` is that it passes plaintext at very low bandwidth and only when it has the syntactic and semantic form of plausible message headers. The required security policy for the overall system is that "very little" information derived from the data parts of incoming `red` packets should be present in the clear in outgoing `black` packets.

I think it is reasonable to believe that suitably elaborated specifications of these local policies, combined with the interpretation of the policy architecture diagram sketched in Section 2.1.5, can deliver a credible argument that this architecture delivers the required security policy—at least to the level of "Medium Assurance" as this is understood in the U.S.A., that is, approximately Evaluation Assurance Level (EAL) 4 or 5 of the Common Criteria.

Higher assurance levels require formal methods of reasoning. Formal specifications for components often take the form

- Component A guarantees property P if its environment ensures property Q.

When components interact, they provide the environment for each other, so we may have

- Component A guarantees P if its environment ensures Q, and
- Component B guarantees Q if its environment ensures P.

Under suitable assumptions it is then sound to conclude that the composition of A and B (often written $A||B$) guarantees the conjunction $P \wedge Q$.

In a MILS architecture, the environment of a component consists of the other components that are sources of channels whose destination is the component under consideration. The "suitable assumptions" under which this *assume-guarantee* rule for compositional reasoning is sound are primarily that components interact only through the explicitly given computational mechanisms (i.e., no hidden or unexpected channels for interaction), and this is exactly what MILS guarantees.

In the case of the encryption controller, we know nothing about `black` except that its environment is `bypass` and `crypto`; we will have suitable specifications for `bypass` and `crypto`, and both of these have `red` (about which we know nothing) as their environment.[6] Formal compositional reasoning about this example would be challenging (especially if we sought measures for leakage bandwidth) but I believe that it is feasible.

Fortunately, although many MILS architectures will be more complex than this, the arguments for their security will be simpler because they depend primarily on assigning security levels to information flows [20]. Information flow is generally formalized in terms of *noninterference* [6] and there are several known difficulties that might seem to pose complications here—namely, that noninterference does not compose and is not preserved under refinement [13]. These difficulties are consequences of the fact that noninterference is not what computer scientists call a "property" (that is, a predicate on—i.e., a subset of—the traces of a state machine): it is instead a higher-order concept (that is, a predicate on predicates of traces). Furthermore, it is crucial to the MILS policy architectures that flows are intransitive, whereas standard noninterference is transitive [22]. If flows were transitive, then the channels from `red` to `bypass` and from `bypass` to `black` in the encryption controller would (by transitivity) allow a direct channel from `red` to `black`, which vitiates the whole design.

Now, the only information flow behavior that can be enforced by a policy component is a property (more particularly, a safety property [21]) and it turns out that there is a compositional characterization of these flows in terms of a reformulation of intransitive noninterference [25]. It seems likely that compositional analysis for many MILS policy architectures can be constructed on this foundation. Indeed, tools that perform information flow analysis on AADL specifications that closely resemble MILS policy architec-

[6] The full example would need to consider the control channels in the reverse direction; this would make the argument appear circular (as it is in the $A||B$ definition), but it remains sound.

tures are already under construction [8, 28], and formal underpinnings for these can probably be provided along these lines.

3.2 Interactions between Policy and Resource-Sharing Components

I noted earlier that Section 2.1.5 specifies a model for the *assumptions* of the MILS policy architecture level, and the *requirements* for the resource-sharing level. Thus, the desired interaction between the policy and resource-sharing levels is that the resource-sharing level should precisely *implement* the policy architecture. That is, the combined behavior of the components of the policy level, implemented on the resource-sharing level, should be identical to that of the policy architecture executing in an idealized manner where each component is its own separate resource. This is a property called *composability*: the policy architecture behaves the same on its own as when it is composed with (i.e., runs on) the resource-sharing components.

If all the components of the policy architecture were well behaved, then composability simply means that resource sharing does not "get in the way." However, untrusted policy components might attempt to subvert the architecture by creating covert channels, or might attempt to crash the resource-sharing component that hosts them. Hence, the resource-sharing level must definitely "get in the way" of misbehaving policy-level components: its task is to enforce the policy-level architecture independently of the cooperation of the policy-level components.

Although composability is described as a relation between given policy and resource sharing components, in practice resource sharing components will implement a stronger property that ensures they are composable with *any* policy components, including faulty or malicious ones. This is similar to the standard verification requirements on separation kernels (nonbypassable, tamperproof, and correct, sometimes extended and abbreviated NEAT [26]).

3.3 Interactions among Resource-Sharing Components

If we implement a fragment of a policy architecture on a separation kernel and another fragment on a partitioning filesystem, each of these resource-sharing components is required to be composable with the fragment that it implements. It is natural to expect that the combination of the two components will be jointly composable with the combined fragments. Symbolically, this is

- composable(A) and composable(B) yields composable(A+B)

where + denotes composition of resource-sharing components. We call this requirement *additive composability*.

Compositional assurance and evaluation for MILS is based on the properties sketched above: compositionality for policy components, and additive composability for resource sharing components. Subsequent reports will develop the mathematical foundations for these and establish their soundness, but they are no more than formalizations of a reasonable intuition, which is summarized in the slogan (due to Rance DeLong) "composability makes the system safe for compositional reasoning."

4 Conclusion

I have described the MILS approach to secure systems design and assurance. The essential feature of MILS is separation of the issues of policy enforcement and resource sharing: the former is tackled in the first step of MILS design and results in an abstract policy architecture, while the latter is tackled in a second step that implements the architecture on suitable resource-sharing components.

The goal at the policy architecture step is to decompose functions so that the components that need to be trusted are as simple as possible and trusted with respect to simple local policies. The policy architecture assumes that components are isolated state machines that communicate only over known, unidirectional channels, and that subdividing functions to simplify the trusted components exacts little cost in performance or acquisition. The components at the resource-sharing level must discharge these assumptions: that is, they must allow many policy-level components to share physical resources in a manner that is both secure, efficient, and cost-effective.

Assurance in MILS is developed compositionally: that is, the assurance for the full system is derived from that of its components and its architecture. As with design, assurance in MILS takes place in two main steps. In one step, the local security policies of the trusted policy-level components are shown, under the constraints of the architecture, to deliver the security policy of the full system; this is a property called *compositionality*. Subsidiary to this step, the implementations of the trusted components must be shown to enforce their local security policies. In the other major step, resource-sharing components must be shown, individually and collectively, to deliver and enforce the architectural assumptions of the policy level; this is a property called *additive composability*. I have sketched the basis for the mathematical

model in which these processes can be formalized; the formalization will be developed in a subsequent report.

MILS assurance and evaluation is performed within the basic framework of the Common Criteria [4], although it should be noted that the Common Criteria do not sanction compositional evaluation of the kind advocated here. System developments are encouraged and aided to use the MILS approach through sponsored development and approval of common criteria protection profiles for necessary and useful components, and stimulation of a COTS marketplace for products compliant to these. The protection profile for high-assurance separation kernels has already been approved [9]and several others are under construction.

These protection profiles began their development prior to the approach to compositional assurance presented here. An issue that requires attention is compatibility between these protection profiles and our proposed approach to compositional assurance. Compositional assurance requires that assurance for components delivers certain claims: compositionality for policy components and additive composability for resource-sharing components. We need to check that current protection profiles deliver these claims, to suggest modifications if not, and to provide guidelines for future profiles.

It is unlikely that extensive revision to existing profiles will be needed because the approach and requirements described here really do little more than codify the intuition that underlies MILS. For the same reason, there is no conflict between the account of MILS given here and other accounts such as [3, 26]; the difference is one of emphasis. The account here is focused on the framework for assurance in MILS, whereas earlier accounts were more concerned with the mechanisms of MILS implementation. Similarly, there is little conflict between MILS and some other recent approaches to secure system development, such as the High Assurance Platform (HAP) program: MILS cultivates a marketplace of resource-sharing components, whereas HAP provides a specific platform, but both adhere (implicitly at least) to the two-stage approach to secure system design.

Compositional assurance and evaluation, as pioneered by MILS, is an exciting and radical advance in certification practice. Success here could have widespread impact, beginning with integrated modular avionics (IMA), which have a very similar architectural basis but a very weakly compositional approach to certification [17], and proceeding to other fields such as medical devices, where the demand for "plug and play" interoperability [27] requires exactly this capability.

References

[1] Marshall D. Abrams and Harold J. Podell, editors. *Tutorial: Computer and Network Security*. IEEE Computer Society Press, 1986.

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

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

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

[5] Rance DeLong, John Rushby, and N. Shankar. A MILS formal security model, 2008. Forthcoming.

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

[7] David Greve, Matthew Wilding, Raymond Richards, and W. Mark Vanfleet. Formalizing security policies for dynamic and distributed systems. Unpublished, September 2004.

[8] Jörgen Hansson. OSATE plugin for security analysis. Available at `http://la.sei.cmu.edu/aadlinfosite/OSATE1.2.4Download.html`, January 2008. See also `http://www.sei.cmu.edu/news-at-sei/features/2008/01/01-feature-2008-01.html`.

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

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

[11] Myong H. Kang, Ira S. Moskowitz, and Stanley Chincheck. The Pump: A decade of covert fun. In *Proceedings of the Twenty-First Annual Com-*

puter Security Applications Conference, pages 352–360, IEEE Computer Society, Tucson, AZ, December 2005. Invited "Classic Paper" presentation.

[12] Hermann Kopetz. Elementary versus composite interfaces in distributed real-time systems. In *The Fourth International Symposium on Autonomous Decentralized Systems*, IEEE Computer Society, Tokyo, Japan, March 1999.

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

[14] *Validated Product—Owl Computing Technologies Data Diode Network Interface Card Version 4*. NIAP CCEVS, February 2007. Available at `http://www.niap-ccevs.org/cc-scheme/st/index.cfm/vid/10208`.

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

[16] *ARIANE 5: Flight 501 Failure*. Report by the Inquiry Board, July 1996. Available via `http://en.wikipedia.org/wiki/Ariane_5_Flight_501`.

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

[18] Jeffrey Choi Robinson and Jim Alves-Foss. A high assurance MLS file server. *ACM Operating Systems Review*, 41(1):45–53, January 2007.

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

[20] John Rushby. Networks are systems. In *Proceedings of the Department of Defense Computer Security Center Invitational Workshop on Network Security*, pages 7–24 to 7–37, Publ. by Department of Defense

Computer Security Center, New Orleans, LA, March 1985. (Reprinted in [1, pp. 300–316]).

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

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

[23] John Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Available at `http://www.csl.sri.com/~rushby/abstracts/partitioning`, and `http://techreports.larc.nasa.gov/ltrs/PDF/1999/cr/NASA-99-cr209347.pdf`; also issued by the FAA.

[24] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings, Part E: Computers and Digital Techniques*, 137(1):17–30, January 1990.

[25] Ron van der Meyden. What, indeed, is intransitive noninterference? (extended abstract). In *Proc. 12th European Symposium on Research in Computer Security (ESORICS)*, Volume 4734 of Springer-Verlag *Lecture Notes in Computer Science*, pages 235–250, Dresden, Germany, September 2007.

[26] W. Mark Vanfleet, Jahn A. Luke, R. William Beckwith, Carol Taylor, Ben Calloni, and Gordon Uchenick. MILS: Architecture for high-assurance embedded computing. *Crosstalk*, August 2005. Available at `http://www.stsc.hill.af.mil/crosstalk/2005/08/0508Vanfleet_etal.html`.

[27] Susan F. Whitehead and Julian M. Goldman. Getting connected for patient safety: How medical device "plug-and-play" interoperability can make a difference. *Patient Safety and Quality Healthcare*, January/February 2008. Available at `http://www.psqh.com/janfeb08/connected.html`.

[28] WW Technology Group. EDICT Tool Suite. Available at `http://wwtechnology.com/products/EdictCore.htm`, March 2008.