

Software Mediators for Transparent Channel Control in Unbounded Environments*

Nadine Hanebutte⁺, Paul Oman, Michael Loosbrock[‡], Austin Holland[‡],
 W. Scott Harrison and Jim Alves-Foss

Center for Secure and Dependable Systems
 University of Idaho

Moscow, ID, 84844-1008
 phone: (208)885-4114
 fax: (208)885-7099

hane,oman,harrison,jimaf@cs.uidaho.edu

‡loos0769,holl12391@uidaho.edu

Abstract— Establishing verifiably secure communications is a daunting task, especially in unbounded computing networks such as the Internet and the Global Information Grid. The Multiple Independent Levels of Security (MILS) architecture has been developed to facilitate this task. Wrappers, filters and mediators, both hardware and software, have been proposed as MILS mechanisms to enforce communication security policies such as data isolation and sanitization.

This paper describes two experimental projects showing how software mediators can be implemented using CORBA in two different environments: a standard Unix TCP/IP network with multiple workstations, and a single board computer running the Integrity operating system with a separation kernel supporting multiple isolated execution environments. The first example shows how protocol mediators can enforce communication-related security policies on standard networks, while the second shows that same functionality implemented on a MILS-based architecture. The projects show how transparent communication security policies can be implemented with existing technologies and without any modifications to the operating system kernels.

Category: Experience

I. INTRODUCTION

High-assurance computing systems often rely on resource separation to enforce data integrity and confidentiality, and to provide system reliability. Resource separation in such systems has traditionally been implemented through redundancy of both hardware and software. For instance, classified data residing on military computer systems is normally separated according to sensitivity level and stored

in separate systems. However, the overhead of system redundancy is often costly to develop, deploy, and maintain, and the interfaces between redundant systems can become complex and inefficient. To address the demand for efficient resource separation in high-assurance computing systems, the Multiple Independent Levels of Security (MILS) architecture has been developed through a collaboration between industry and the academic community [1], [2], [3]. MILS is a high-assurance, high-performance computing architecture that can enforce strict security and separation policies on data and processes residing on one or more distributed micro-processors. MILS systems use a separation kernel based on the idea of separable resource partitions first proposed by Rushby [4], [5], [6]. The MILS architecture provides a secure framework for hosting and maintaining separation between resources on a single micro-processor which would traditionally be hosted on separate micro-processors.

Operating under the assumption of safety and security through isolation, MILS provides maximum separation of data and processes within a single computing environment. MILS also represents a paradigm shift from the monolithic structure of traditional secure computing platforms to the division of security enforcement mechanisms into modular components which work in tandem to enforce separate but inter-dependent policies. Furthermore, the MILS architecture is designed to be formally verifiable, allowing the development of safety-critical and high-security systems which can be certified as high as Evaluation Assurance Level 6 or 7 in the Common Criteria [7]. One target application of MILS is the U.S. military's Joint Tactical Radio System (JTRS) which will allow soldiers in the field to transmit and receive communication of different security levels on a single piece of equipment [8], [9]. Similar use of the MILS architecture is planned for the future U.S.A.F.

*This material is based on research sponsored by AFRL and DARPA under agreement number F30602-02-1-0178. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and DARPA or the U.S. Government.

⁺Contact Author

F35 Joint Strike Fighter [10], [11].

The MILS architecture incorporates the concept of separation in two different regards. First, MILS provides *process separation* by enforcing security policies that strictly control information flow between user processes. Information channels, both overt and covert, should only exist when explicitly permitted by the system security policy. Without any means of communication, two processes residing on a single microprocessor become just as separated as if they each resided on physically disjoint microprocessors. Secondly, MILS provides *functional separation* by moving security mechanisms normally implemented at the kernel-level into external modular components [12]. The components enforce narrow, specific policies that are layered to comprise the overall system security policy. An advantage of this modular approach is that MILS components are small and simple enough for rigorous, independent evaluation using formal methods such as model-checking and theorem proving [13]. Also, MILS systems can be assembled using components built and verified by multiple independent vendors, reducing the time and cost of development.

In this paper we present two example implementations of a MILS architecture. The first implementation is a proof-of-concept using physical space separation to show the implementation of time and space separation using the Linux operating system (OS). The second example was built using Integrity OS, running on a partitioning kernel that allows for true process separation on a single host. The first example shows the use of commercial-of-the-shelf (COTS) components to control covert channels and failure propagation, while the second example does the same using a state-of-the-art MILS architecture.

II. SPACE SEPARATION TO ENSURE ACCURATE DATA ACCESS

The challenge of data distribution through an ad-hoc, uncontrolled environment such as the Internet is lack of predictability of process reliability and security. The majority of services and communication paths cannot be classified as trusted, nor is there a non-trivial way of increasing trust in unbounded environments such as the Internet. As a result, every communication path leading through an uncontrolled host or device can be potentially compromised.

The problem of path protection is not limited to an unbounded environment. Every network that connects somewhere to the “outside” must at least be able to handle malicious traffic at its boundaries. This is especially true in a net-centric, defense-in-depth environment like the Global Information Grid (GIG) that must connect all legitimate users to their needed information in a timely manner, while relying on communication channels that are not under its direct control [3], [8]. It must be possible to control, monitor and sanitize communication before it enters a sensitive network containing classified content. In some case, the

only controlled entities available are the end-point hosts of the communication path. It is even possible that only one of the hosts can be considered as trustworthy.

In order to prevent a trusted host or process becoming compromised, the data to and from the host must be sanitized. In addition, the execution space of the trusted entity must always be protected. Incoming data must go through a sanitization stage to filter malicious content from the data stream. Outgoing data must be checked to ensure that the receiver only gets the information it is eligible to receive and to ensure that the information is properly encrypted or downgraded considering the communication path.

Integration of these tasks into the communicating process is possible in bounded, well-controlled environments, but for dynamic environments it is more appropriate to task a separate entity with communication control. This simplifies the implementation of processes by allowing layered reuse. For example, re-using trusted sanitization processes ensures that communication is always handled exactly the same. Depending on the available hardware and software option such sanitization “stations” can be implemented at a host or at the network level. Application “wrappers” and [14] and communication “mediators” (sometimes called monitors) are two implementation mechanisms to achieve data separability. Wrappers usually control application input through application programming interfaces (API), while the discussed approach monitors traffic en route before being handled by APIs. As such it overcomes some of the shortcomings of such systems and allows also for a more lightweight implementation.

III. WRAPPERS AND LAYERS FOR DATA SEPARABILITY

A. Wrappers

Filtering of data between the application level and the OS can be accomplished through wrappers [14]. Wrappers are inserts to, or replacements of, APIs. Instead of going straight to and from the application, the data is rerouted to a replacement API that implements the same functionality in a more secure or reliable manner. An example of the second type are replacement libraries for APIs that are vulnerable to buffer-overflow and format string attacks, such as *Libverify* and *Libsafe* [15]. However, these approaches target specific APIs. As the number of vulnerable libraries is fairly large [16], this approach is as effective as the library coverage. Also, wrapper solutions are usually installed on the same host as the data to be protected can be compromised if memory space is not completely protected.

B. Mediators

While wrappers usually operate at the application level itself, our work focuses on the control of content and protocol information in isolation from the applications that will eventually process the data stream. Jaeger et. al. discuss the concept of transparent monitors with respect Inter

Process Communication (IPC) monitoring [17]. They suggest a monitor as a kernel extension functioning similar to wrappers at either end of an IPC connection. Similar approaches have been discussed by Kang and Moskowitz [18] and Moore [19] with respect to data transfer across security clearance levels, but their approach relies on additional application wrappers and a special protocol.

In this paper we discuss two experimental projects to implement interprocess data isolation based on the idea of separable resources as first proposed by Rushby [4], [5], [6]. In our projects, we limited ourselves to the use of available technology, without the modification of the OS kernel, to achieve the goal of channel control, allowing transparency and control over all data traffic independent of the protocol, source, or destination.

IV. DATA SEPARATION IMPLEMENTATION EXAMPLES

A. Example 1: Physical Separation for Non-partitioning OSs

The initial test implementation of a separation architecture was done using two networked Linux computers. We used separate computers since Linux does not provide a strong enough operating system level application separation, meaning that application failures may propagate to other applications or even into the kernel. Each PC represents a partition, an isolated execution space that is only allowed to communicate through the transparent mediator, which sanitizes the information stream between the two hosts. The workstations are on physically disjoint subnets that are bridged by the gateway, which serves as the mediator. The mediator, as well, is a host running Linux (Figure 1). TCP/IP networking serves as inter-partition communication. A Common Object Request Broker Architecture (CORBA) client/server application is distributed between the two workstations to produce inter-workstation messages invoking the General Inter-ORB Protocol (GIOP) protocol.

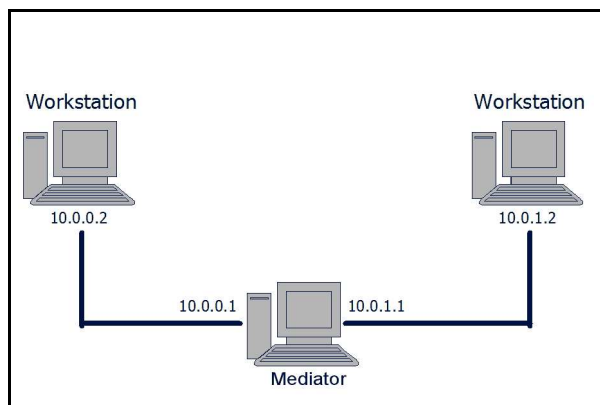


Fig. 1. Experimental Setup; Physical Separation

The workstations are on physically disjoint IP subnets

that are bridged by the gateway, which serves as the mediator. Messages content checking under Linux is possible by using a combination of Linux features like IP forwarding, IP tables, and the libipq library.

- IP forwarding is enabled if the entry in `/proc/sys/net/ipv4/ip_forward` is set to 1. This means that the Linux host is allowed to hand the traffic from one physical Ethernet device to the other.
- IP tables is the network packet filtering mechanism in the Linux kernel that applies tables of filter rules to incoming and outgoing packets [20]. In this example we are attempting to filter all GIOP traffic. However, IP tables can only filter at the transport layer. Since the implementation of the CORBA client and server uses TCP/IP as the underlying transport mechanism, IP tables is configured to filter for all TCP data: `iptables -t filter -A FORWARD -p tcp -j QUEUE`.
- Libipq is a loadable module user-space queuing library that allows network packets traversing the Linux kernel to be passed up to user-space processes [21] after libipq is loaded via: `modprobe iq-queue`.

All TCP packages are handed from the TCP/IP stack, residing in kernel space, via libipq into user space, where the message content filtering occurs. This is done through a set of C functions that parse the packet for the string “GIOP,” which is part of the GIOP message header. Non-GIOP data is handed back done the TCP/IP stack and forwarded through the outgoing Ethernet device unchanged (Figure 2).

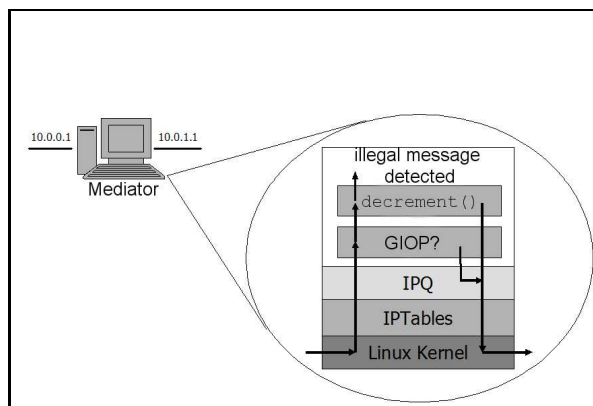


Fig. 2. Mediator Host

The GIOP server in our sample implementation hosts two methods, an `increment()` and a `decrement()` to a counter object. The message filter was coded to allow only `increment()`. If the client attempted to invoke `decrement()` the message was simply discarded.

While this implementation provides valuable insights on how to use a COTS system without kernel modifications to allow for process isolation, the current short coming is the

lack of proper error handling. In order to prevent error handling turning into a “back-flow” covert channel, rules need to be established that enforce proper error response (e.g. the same or similar message will always receive the same error response). The error response has to be structured to obfuscate the existence of the mediator. It must appear to the client as if the error response came from the server or from a non-mediator network mechanism. The error response must also be independent of the exact network path between the sender and the server (i.e., the route through or past other hosts between the sender and the mediator or, in the case of a filtered server response, between the server and the mediator).

This example shows that fine-grained application protocol filtering can be accomplished within a standard OS. Filtering policies can also be added and fine-tuned for different message types. In our example the message type was GIOP, but HTTP, SMTP, FTP, etc., can be used as well. The problem with using Linux is that physical process separation is necessary. It could be argued that the filtering mechanism could be implemented on the host running the trusted service. However, a compromise of the host or the mediator will potentially propagate across the entire host, disabling the mediator service for uncompromised hosts or the server functionality from accessibility through a different mediator gateway. The step from physical to logical process separation is only possible by employing an OS that provides separation as part of its standard kernel functionalities.

B. Example 2: Host-based Separation

The target operating system for the second experimental implementation is Green Hills Software’s DO-178b certifiable Integrity-178B real-time operating system [22], [23]. The actual experimental implementation was done under Integrity 4.0.9a, which is a superset of Integrity-178B. This OS was designed for use within embedded systems employing single-board-computers. Integrity offers strict resource separation by allowing the developer to define multiple isolated execution environments within the system called *address spaces*. The general OS and application layout is shown in Figure 3. Each address space is allocated a fixed amount of system resources (e.g., memory) at compile-time. Unless explicitly relaxed by the system developer, address space resources are completely disjoint and cannot be shared. The “inside” of an address space behaves similar to a Linux host. Multiple processes (called tasks within Integrity terminology) may run within one address space, competing for run-time via priority scheduling. Within an address space there is no memory protection. Protection is only enforced between tasks residing in different address spaces.

Integrity provides a verifiably secure method for inter-task communication using special Integrity objects called

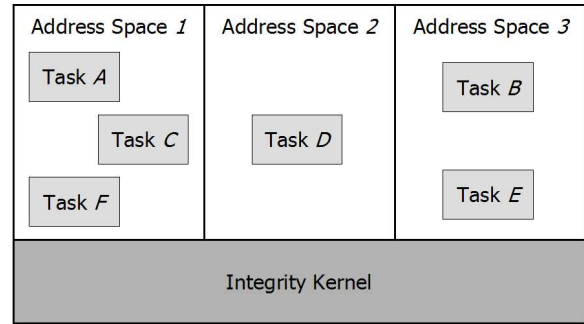


Fig. 3. Partitioning within Integrity

connection objects. When two connection objects are bound together, they form a bi-directional communication channel called an *Integrity Connection (IC)*. Data is then transmitted over the IC by using one connection object for sending and the other for receiving. The API for sending and receiving over ICs is similar to the BSD sockets API. The kernel ensures that data sent over ICs cannot be monitored or tampered with en route. In fact, transmitted data can only be accessed through the connection objects forming the IC. If an IC is comprised of connection objects (i.e. the end-points of an IC) residing in different address spaces, then tasks executing in those address spaces can securely communicate using the IC.

CORBA can utilize several different communication mediums available on different target platforms. The most commonly used transport mechanism is TCP/IP, but transports over operating system specific types of inter-process communication like shared memory and messaging schemes are also available.

Inter-address space communication under Integrity can be performed by invocation of the *Intconn* protocol. It uses the *Integrity Resource Manager (RM)*, an optional kernel library, to mediate connections between CORBA applications. Ideally, communication connections should be fixed and known at compile time in order to prevent covert channels, but as transparency and portability are the main goals of using CORBA, communication connections between client and server address space cannot be assigned at compile time and have to be “discovered” dynamically. In this case we can only ensure that there are no direct connections between the two address spaces. At run time the generation of such direct channels must be prevented.

Initially each address space is assigned an IC to the RM at compile time (Figure 4) by the kernel. The disjunct client and server address space do not have a shared communication interface. In order to create a connection between the two CORBA applications, the server has to announce its “existence” (see Figure 4, Step 1). It does this by registering an end-point of an IC with the RM via the

initial RM to address space ICs. A client would request this end-point as shown in Figure 4, Step 2. If a valid end-point exists, it is handed to the client (Figure 4, Step 3). This results in a direct connection between client and server as shown by Figure 5, Step 4. Eventually a direct IC is created as shown in Figure 5, Step 5.

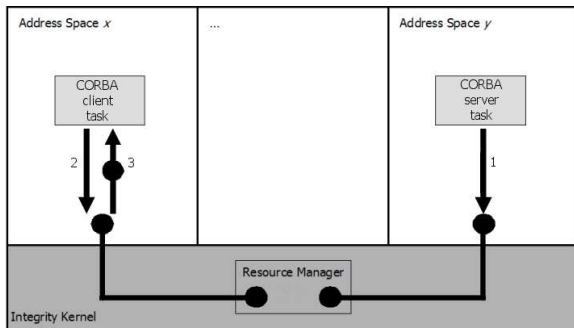


Fig. 4. The Integrity Resource Manager

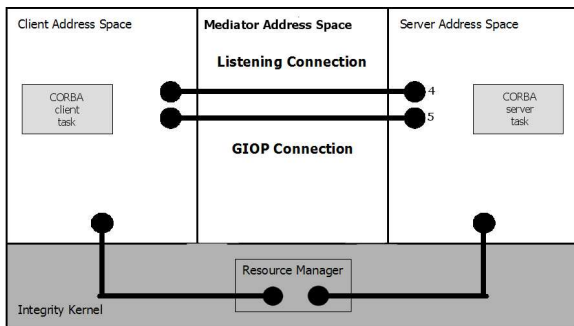


Fig. 5. CORBA and Integrity

In order to prevent the initial server registration and client request issue to the RM, the initial connections generated by the kernel must be re-routed. The mediator, responsible for IC rerouting, is implemented in a separate address space. Upon system boot-up it forces the client's and the server's address spaces to abandon the connection to the RM. It can do so because the system was set up in a way that the mediator address space is the owner of the client and server space. The system implementer has 100% control over the system layout, but only limited control over the content of each address space at run-time. The mediator address space replaces the abandoned connections with connections into its own address space and as a result has complete control over the data sent to and from these two address spaces via the CORBA Intconn protocol.

When the server registers its readiness to serve CORBA requests, the registration message arrives at the mediator in form of a connection end-point (Figure 6, Step 4.1). A client, trying to locate a server-side method, will send its

request to the mediator. Without the mediator, when the RM is used, the client would receive a link to the connection end-point sent by the server from the OS via the RM. By inserting the mediator, if the client is at all allowed to communicate with the server, the mediator will create a new connection and hand one end to the client (Figure 6, Step 4.2). This leaves the client with the impression that it found a valid communication path directly to the server.

The next step in the Intconn communication procedure is for the client to establish the actual data exchange connection. It will generate a new connection and hand one end-point to the server. Again the mediator will actually receive the connection end-point (Figure 6, Step 5.1), generate a new connection and hand one end to the server, tricking it into "thinking" it received a valid data connection from a client (Figure 6, Step 5.2). After the setup of this communication process all IC's point only to the mediator.

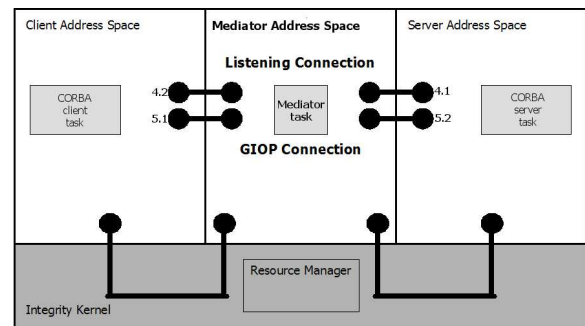


Fig. 6. Communication Channels Re-routed

Similar to our Linux based experiment, any data package sent from the client and the server will have to pass through the mediator. The mediator may parse its content and hand off the data from one connection end-point to another or discard entire messages in order to sanitize the data stream. This set-up can be changed to allow for multiple client and server address spaces. Even though these changes are done at compile time the setup is scalable until computer memory is exhausted.

V. CONCLUSIONS

We have demonstrated a transparent and non-bypassable mediator mechanism between communicating processes. The described system architecture itself can be implemented using physical separation if the OS does not support logical separation. By taking advantage of the high-degree of resource isolation provided by existing separation kernels like Integrity, it is possible to build a system that safely and securely allows separated resources to co-exist and interact on the same micro-processor.

The implementation shows how total process separation and covert channel control is possible on a single host. As the system can control all communication paths, covert channels may be controlled as well. In order to prevent information back-flow through valid channels the system must be extended to systematically address error handling and responses.

The examples presented here were limited to communication protocols supported by CORBA (e.g. TCP and Intconn), but mediated communication can be implemented within any protocol. Covert channel control and transparency are also goals of this implementation. Methods have to be developed that control other communication paths while leaving the protocol and the OS unchanged. If physical separation is used, the isolation of communication end-points is less complex, because all traffic arriving on the interrupted communication medium can be analyzed and intercepted. Physical separation, however, requires extra hardware and complex interactions. The MILS architecture on a single host allows information flow to be secured via inter-partition communication processes with complete transparency.

REFERENCES

- [1] J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor, "The MILS architecture for high-assurance embedded systems," *The International Journal of Embedded Systems (to appear)*, 2005.
- [2] J. Alves-Foss, C. Taylor, and P. Oman, "A multi-layered approach to security in high assurance systems," in *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'37)*, 2004.
- [3] M. Dransfield, W. Vanfleet, B. Beckwith, L. MacLaren, J. Luke, and B. Calloni, "MILS/MLS architecture for deeply embedded systems," in *NetCentric Operations 2004*, 2004.
- [4] J. Rushby, "Design and verification of secure systems," in *Proceedings of the 1st ACM Symposium on Operating System Principles (SOSP-1)*, 1981.
- [5] J. Rushby, "Proof of separability: A verification technique for a class of security kernels," in *Proceedings of the 5th International Symposium on Programming*, 1982.
- [6] J. Rushby and B. Randell, "A distributed secure system," *IEEE Computer*, vol. 16, no. 7, 1983.
- [7] Common Criteria Recognition Arrangement, "Common criteria for information technology security evaluation version 2.1," 2004.
- [8] Department of Defense (USA), "Joint tactical radio system." <http://jtr.s.army.mil>, 2004.
- [9] C. Adams, "Network centric, rush to connect," *Aviation Today - Avionics Magazine*, 9 2004.
- [10] SPG Media, "JSF (F35) joint strike fighter, international." <http://www.airforce-technology.com/projects/jsf/>, 2004.
- [11] C. Adams, "Keeping secrets in integrated avionics, reprint," *Green Hills Software*, 2004.
- [12] B. Ames, "Real-time software goes modular," *Military and Aerospace Electronics*, vol. 14, September 2003.
- [13] L. MacLaren, "New options in embedded computing security," in *Boeing Technical Excellence Conference (BTEC 5)*, 2003.
- [14] N. Mead, R. Ellison, R. Linger, T. Longstaff, and J. McHugh, "Survivable network analysis method," Tech. Rep. CMU/SEI-2000-TR-013 ESC-2000-TR-013, CMU/SEI, 2000.
- [15] A. Baratloo, N. Singh, and T. Tsai, "Transparent run-time defense against stack smashing attacks," in *Proceedings of 2000 USENIX Annual Technical Conference (USENIX'00)*, 2000.
- [16] N. Hanebutte, *Analyses of Security and Survivability as Software Quality Attributes*. PhD thesis, University of Idaho, Moscow Campus, 2004.
- [17] T. Jaeger, J. Tidswell, A. Gefflaut, Y. Park, J. Liedtke, and K. Elphinstone, "Synchronous IPC over transparent monitors," in *Proceedings of the 9th ACM Special Interest Group on Operating Systems (SIGOPS) European Workshop*, 2000.
- [18] M. Kang and I. Moskowitz, "A pump for rapid, reliable, secure communication," in *Proceedings of the 1st ACM Conference on Computer and Communications Security (ACM CCS-1)*, 1993.
- [19] A. Moore, "Network pump (NP) security target," Tech. Rep. NRL/MR/5540-00-8459, Naval Research Laboratory, 2000.
- [20] O. Andreasson, "Iptables tutorial 1.1.19." <http://iptables-tutorial.frozentux.net/iptables-tutorial.html>, 2003.
- [21] J. Morris, "libipq man pages." <http://www.cs.princeton.edu/~nakao/libipq.htm>, 2001.
- [22] "Integrity-178b RTOS." http://www.ghs.com/products/safety_critical/integrity-do-178b.html, 2004.
- [23] RTCA (Radio Technical Commission for Aeronautics), *Software Considerations in Airborne Systems and Equipment Certification (RTCA DO-178b)*. RTCA Inc., 1992.