# Reliability as a Component of Security for MILS High Assurance Platforms

**Rich Goyette, MEng, PEng**

**Abstract**

In this paper we examine fault tolerance in the context of Multiple Independent Levels of Security (MILS) High Assurance Platforms (HAP). Specifically, we seek to determine whether fault tolerance techniques are relevant and, if so, what classes within the fault tolerance taxonomy should be (or *need* to be) addressed. We then examine what affect the unique constraints of the application space have on the choice and implementation of fault tolerance technique. Finally, we examine several case studies that have been proposed in the literature and critique them against the application space constraints.

## Introduction

In this paper we examine fault tolerance in the context of a very specific application – Multiple Independent Levels of Security High Assurance Platforms or MILSHAP. These devices are currently being designed to meet military user requirements for controlled access to multiple security domains (e.g. Unclassified, Secret, Top Secret) on a single hardware platform. Attempts to do this in the past using the concepts of Multi-Level Security (MLS) failed to deliver the required functionality in the correct form factor for a number of reasons – primarily related to the difficulty in assuring them to the requisite levels. Meanwhile, the requirement has not gone away:

> "The real multilevel-security driver [is] the imperative to integrate different data classes to accomplish a mission. For example, to develop a secret military operations plan, [the commander] must consider intelligence information usually maintained at the top secret level, and then create primarily unclassified sub-plans to logistically support the operation. Similarly, top-secret intelligence reporting needs to employ up-to-date secret and unclassified information. In other words, military missions are inherently multilevel." [SAY04]

Because the requirement remains, the general solution has been to employ multiple, independent networks separated in space or cryptographically compartmentalized. This solution has always been sub-optimal since it imposes two very problematic constraints. First, an end user must have access to one physical data terminal for each independent security level. For deployed operations, where space and power is limited, the requirement to have multiple terminals (separated by minimum emanations security zoning distances) is very hard to achieve and maintain. Deployed operations are also characterized by high levels of stress. This, combined with the fact that the user is required to log-in separately to more than one network with different access credentials, leads to higher stress levels, security breaches and a less responsive organization.

The second constraint is the speed of information flow. Adversaries are defeated by having superior intelligence *and* by moving from decision to action faster than they can. With separate networks, the movement of required information between security domains has been via "sneakernet" [KAR05] or "swivel-chair." This approach slows the decision-action cycle and introduces potential vulnerabilities to the networks.

Recent developments in technology and ideology have spurred a renewed interest in trying again to meet the requirement. The concept of having Multiple

Independent Levels of Security (**MILS**) on a single hardware platform has taken root ([HHO05][FOSS04][JAC04]) and is being facilitated by developments in digital rights management technologies for commodity computing and server virtualization. Old software-based concepts to facilitate separation on a single such platform (such as Rushby's separation kernel and Virtual Machine Monitors [RUS89][NEU06][KARG96]) are receiving new attention. These technologies and concepts are being examined and influenced by the National Security Agency (NSA) in order to see the development of a High Assurance Platform (**HAP**) – one that is capable of enforcing separation between multiple security domains with high assurance of correct operation but that is fabricated predominantly from commodity computing hardware (motherboards, memory, etc).

In this work, we examine the role of design fault tolerance with respect to the ability to assure and trust a **MILSHAP**. Relatively small code bases and simplicity are two features of separation kernels and virtual machine monitors that make them attractive for a MILS architecture. However, to be trustworthy, the separation kernel must also be able to fail-secure in the event of a fault. The ability to incorporate fault tolerance while keeping the separation kernel and platform hardware non-complex appears to be a very delicate balance.

We examine to what extent fault tolerance is mandated for high assurance design and then we proceed to examine what affect the unique constraints of the application space have on the choice and implementation of fault tolerance technique.

The remainder of this paper is divided up as follows: In Section I, we define the application space and the unique requirements that make this area difficult to design in. We demonstrate the requirement for a minimal set of fault tolerant techniques and then review the fault tolerance research domain to establish some of the general trends that have been implemented. In Section II, we examine various hardware and software fault tolerant techniques in light of the MILSHAP problem space. Finally, in Section III, we conclude.

# SECTION I: Background

## The Failure of Multi-Level Secure (MLS) Systems

In [Tan06], the authors assert that today's operating systems have two characteristics that make them both unreliable and insecure. First, they are very large (in terms of the number of lines of code they contain) and second, they have poor fault isolation (in terms of the number of interconnected and interdependent procedures). The authors suggest that microkernels (which had long been dismissed because overhead costs rendered poor performance) might be well positioned for a comeback due to their excellent record with respect to both fault isolation and small code base. The enabling technology for their return is, of course, the high-speed CPU found in all commodity PCs.

The resurgent potential of the microkernel has not eluded notice by the military research and development community who have been trying to solve the Multi-Level Secure (MLS) problem in a practical way for at least the past thirty years [NEU06][FOSS04]. Part of the reason for the failure of MLS to deliver in a time and cost effective manner had to do with the *high level of assurance* expected of them. These assurance levels were first codified in the DoD Trusted Computer System Evaluation Criteria (TCSEC) standards [5200.28]. Significant effort and expense was invested by both industry and academia to produce prototypes that would meet the highest levels of assurance within the TCSEC but the techniques and tools required to provide adequate assurance arguments were still evolving.

Unfortunately for MLS and their military customers, operating system and hardware vendors became distracted by an explosive growth in the personal computer market in the early 90's. As they fought for market share, vendors gradually became uninterested in designing their products to meet the high assurance requirements of the military [SAY04]. As a result, the most popular operating systems on the market today are both large and insecure. Only recently has security been considered somewhat on-par with time-to-market for the dominant OS vendor (MS-Windows).

## Multiple Independent Levels of Security (MILS)

Recently, an approach entitled "Multiple Independent Levels of Security" (MILS) seeks to provide some of the capabilities of MLS while simultaneously addressing assurance requirements. The MILS intent is to accomplish this by taking advantage of the size and fault isolation features of separation kernels which function to provide time-and-space separation between legacy operating systems. Thus, an additional and significant advantage is that MILS promises to invoke little or no change to legacy interfaces.

While there exists a host of non-trivial technical challenges within the MILS approach, the security evaluation (verification to high assurance) is seen to be less demanding than with MLS because the size and functionality of the separation kernel is significantly reduced in comparison to a monolithic OS kernel. In addition, the notion of a separation kernel is to enforce a controlled isolation between security domains and this is perceived to be easier to demonstrate (for assurance purposes) than the subject/object separation of an MLS system.

For high assurance systems, it is critical that the confidentiality and integrity of data within each security domain be preserved and that no information leakage between domains occurs – even in the presence of malicious or benign faults. The separation kernel must, at a minimum, "fail safe" with respect to these security services. However, for a MILS systems, failing safe may not be good enough – availability of certain services in the presence of faults may be necessary to ensure mission success and prevent loss of life. Therefore, such systems must also be *fault-tolerant*.

MILS high assurance platforms (MILSHAP) must therefore balance two potentially opposing forces. The first is the need to remain small and easily understood for verification and assurance purposes. The other is to be reliable and fault-tolerant. These two requirements may be opposing because the inclusion of fault-tolerant capabilities naturally increases the both the separation kernel code size and complexity and may affect the hardware complexity. This paradox is summed up best by Kang et. al. in [KAN98] who were presenting the design and *assurance strategy* for a system that allowed cross-security domain communication:

> "**Error handling.** Error or failure handling is one of the most difficult parts of design because there is no theory or best way to handle errors. One important question is "How smart should the pump be for error recovery?" The smarter the pump, the more complex the software, and the harder it will be to ensure its correct behavior. We designed the pump's error handling with this in mind."

The objective for MILSHAP is therefore to optimize the spectrum of fault-tolerant techniques in order to try to achieve a balance between assurance (size and complexity of the separation kernel and hardware) and fault tolerance.

## Separation Kernels versus Virtual Machine Monitors

In order to understand the mechanism upon which MILSHAP is based, it is necessary to identify the similarities and differences between microkernels, separation kernels, and virtual machine monitors with respect to isolation and assurance.

A *microkernel* is a body of code that provides a very minimal set of operating system functionality such as address space management, process and thread management and inter-process communication [WIKI06]. The microkernel code executes in a privileged state. All other code that is non-essential to the operating system is implemented outside of the kernel as "user services" executing in a less-privileged state.

From a security and reliability perspective, micokernels are extremely attractive. Microkernels can be designed to provide strong "separation" between user space processes. By this we mean that, through intelligent memory, buffer, and register management, the kernel can ensure that no user process can directly influence the data or execution of any other user process. Such kernels are often referred to as "separation kernels"[SKPP].

As a result of this strong isolation and the fact that user services execute in un-privileged modes, overall system reliability is enhanced because the failure or compromise of a user service generally does not translate into a failure of the system overall. It is therefore not surprising that microkernels are found at the core of safety critical embedded systems such as those produced by Green Hills and other embedded applications developers in the aerospace industry.

Inter-Process Communication (IPC) channels exist in microkernels as the primary means of kernel extensibility and communication. They allow for kernel-controlled execution handoff, data transfer, and resource delegation between protection domains [HEI06]. However, the biggest advantage of microkernels from a security and reliability perspective is that, as a result of kernel minimization, a rigorous inspection of the operating system code is possible. As indicated in [TAN06], monolithic operating systems such as Linux and Windows are intractable for the purposes of code inspection since they each contain several million lines of code.

For all of their advantages, why are microkernels not more widely applied? Microkernels were the source of much academic interest throughout the eighties and nineties and, indeed, a number of potentially mainstream operating systems have been developed within the microkernel development mindset (see, for example, [MACH], [L4], [MINIX]). However, by constraining the kernel to a small code base, system performance was often quoted as an issue even though many microkernel developments sport performance comparable to their monolithic cousins [HAND05]. More directly, an operating system is generally deemed successful by its ability to attract mainstream applications and Microsoft Windows was the clear winner throughout the eighties and nineties. This was in large part due to the fact that functionality and ease of integration were valued significantly more than security or reliability. A significant challenge for microkernels has been the effort required to port an existing application for use with a microkernel-provided application programming interface (API) [HEI05]. Without a large user base, there is no financial incentive for software vendors to do so.

## Virtual Machine Monitors

Unlike a microkernel, a virtual machine monitor (VMM) is a software layer that is designed to "partition a single physical machine into multiple virtual machines" [WHI05]. Virtual machines are created by emulating the underlying hardware and providing this abstraction to each new domain. There are several clear security and reliability benefits inherent in the design of VMMs ([WHI05], [UHL05], [ROS05]). The first is that legacy software designed for a particular hardware environment (e.g., Intel CPUs) can be executed out-of-the-box within a virtual machine. This is an extremely important factor for synchronized application migration. A second is that strong isolation between each virtual machine can be designed into the VMM fairly easily. Once again, security breaches or system failures in one virtual machine will not propagate to others because each VM is unaware that the others are present on the same hardware platform. Finally, an inferred benefit is that the VMM is likely to be small [HEI05] and therefore "assurable" since the VMM code does little more than proxy access to hardware and context switch domains.

## Assurance Arguments

As discussed earlier, one of the significant challenges of the MLS era was to design systems that were trustworthy and assured. Unfortunately, the precise meaning of these terms has been difficult to establish in relation to the perceived evidence that is required from both a designer and evaluator perspective. This problem is well summarized in [AK05]:

> "For safety-, mission-, or security-critical systems, there are typically regulations or acquisition guidelines requiring a documented body of evidence to provide a compelling justification that the system satisfies specified critical properties. Current frameworks suggest the detailed outline of the final product but leave the truly meaningful and challenging aspects of arguing assurance to the developers and reviewers."

The conclusion is that assurance arguments are fundamentally subjective. Some arguments might sway one security evaluator but fail to do so with another [MAY04]. To address this ambiguity, two, non-orthogonal approaches have developed. The first, known as *Assurance Cases for Security* [BGM06][BGMW06], is an area of research in the literature that has recognized and taken up the problem and is attempting (with limited success) to develop a framework for security assurance cases.

The second approach to rationalizing security arguments is embodied in the *Common Criteria for Information Technology Security Evaluation* which is better known as simply the Common Criteria (CC). The CC are intended to "provide

some basic guidelines for standardizing the persuasiveness of assurance arguments" [May 04] and are being used *now* in order to generate structured arguments for medium and high assurance devices and applications. While the CC is not perfectly adapted to all designs, it is currently the most widely implemented methodology for establishing assurance [HUN04]. A more detailed introduction to the relevant parts of the CC is provided in Annex A.

## Common Criteria Requirements for Fault Tolerance

The U.S. DoD is currently working on a Common Criteria Protection Profile for a separation kernel to be implemented in environments requiring high robustness [SKPP]. This protection profile specifies two families of fault tolerant functionality – FPT_FLS (fail secure) and FPT_RCV (trusted recovery). The text of the protection profile for fail secure is provided in Figure 1.

> **5.6.3 Fail Secure (FPT_FLS)**
>
> 5.6.3.1 Failure with Preservation of Secure State… The TSF shall preserve a secure state when the following types of failures occur:
>
> [assignment: *list of failures that are detected by tests defined in FPT_AMT.1 and FPT_TST.1 and that require preservation of secure state*].
>
> *Application Note: The ST author is to provide the list of post-initialization failures that can be detected and for which the TSF can respond to and preserve a secure state.*
> *Application Note: TSF failure modes vary and may include "hard" failures such as those associated with hardware failure or unrecoverable software errors, and "soft" failures such as intermittent hardware errors and recoverable software errors.*

Figure 1: SKPP Fail Secure Functional Requirement

The conclusion from text of Figure 1 is that fault-tolerant functionality is *required* but that the range of faults to be responded to is defined by the developer of the separation kernel. Therefore, subject to explicit direction by a certifying authority, there is still some freedom for a developer to pick-and-choose the types of fault tolerance to be included. This choice will be influenced by how much coverage the vendor wishes to achieve versus the amount of complexity introduced to the overall system. Increased coverage can be used as an assurance argument but only if it does not significantly increase the code base and system complexity to the point where it cannot be verified to a trusted level.

As a final note, it should be remembered that protection profiles, in general, can be designed for both hardware and software. The SKPP was specified with a

software environment in mind but this does not mean that security functionality within the SKPP cannot be provided by hardware elements.

## Taxonomy of Faults

In [ALRL04], a thorough taxonomy of faults is presented. Each fault is classified against one of eight basic viewpoints where each viewpoint describes the origin of the fault. A reduced summary of this taxonomy is presented in Table 1.

**Table 1: Taxonomy of Faults**

| Viewpoint | Fault Type |
|---|---|
| Phase of creation | Developmental<br>Operational |
| Dimension | Hardware<br>Software |
| Objective | Malicious<br>Non-Malicious |
| Intent | Deliberate<br>Non-Deliberate |
| Persistence | Permanent<br>Transient |

The designers of a security target for a MILSHAP should be concerned primarily with *operational* faults in *hardware* and *software* that display either *permanent* or *transient* persistence. Transient faults are also known as *soft errors* [XIE05] and they are particularly troublesome from a fault tolerance perspective since they cause upset events (errors) but do not permanently affect the system. Soft errors can cause changes to data resident in memory as well as the flow of execution of a program (random bit flips occurring on cached or queued instructions). Indeed, there is a growing concern that transient faults will become the dominant concern in the near future due to increasing component densities and decreasing feature size, switching voltages, and noise margins ([SAHA06] [LI06][XIE05]). For example, [REI05] reports that 80% of the 200,000 latches on the SPARC64 are covered by some form of fault tolerance technique.

Certification and accreditation of high assurance systems containing commodity chips should take into account techniques used to provide fault tolerance since they will affect both the fail-secure nature of these devices as well as the movement and storage of data (which is important to ensure that covert channels and un-intentional data leaks do not occur).

As a final note, because we seek to minimize the size of the separation kernel (or virtual machine monitor) and then scrutinize that kernel with great care, we can assume that *developmental* faults will be avoided through careful code review. The separation kernel must also be fault tolerant to *accidental* and *deliberate* attempts to circumvent the separation between security domains. Such attempts

have traditionally taken advantage of failures in input validation both from the user and between software modules.

## Taxonomy of Fault Tolerance Techniques

There are many well-researched techniques for implementing fault tolerance in computing systems ([NASA00], [SAHA06], [GOS04], [XIE??]). Figure 2 shows a partial taxonomy of these techniques [NAY06] and includes information, hardware and software classes.

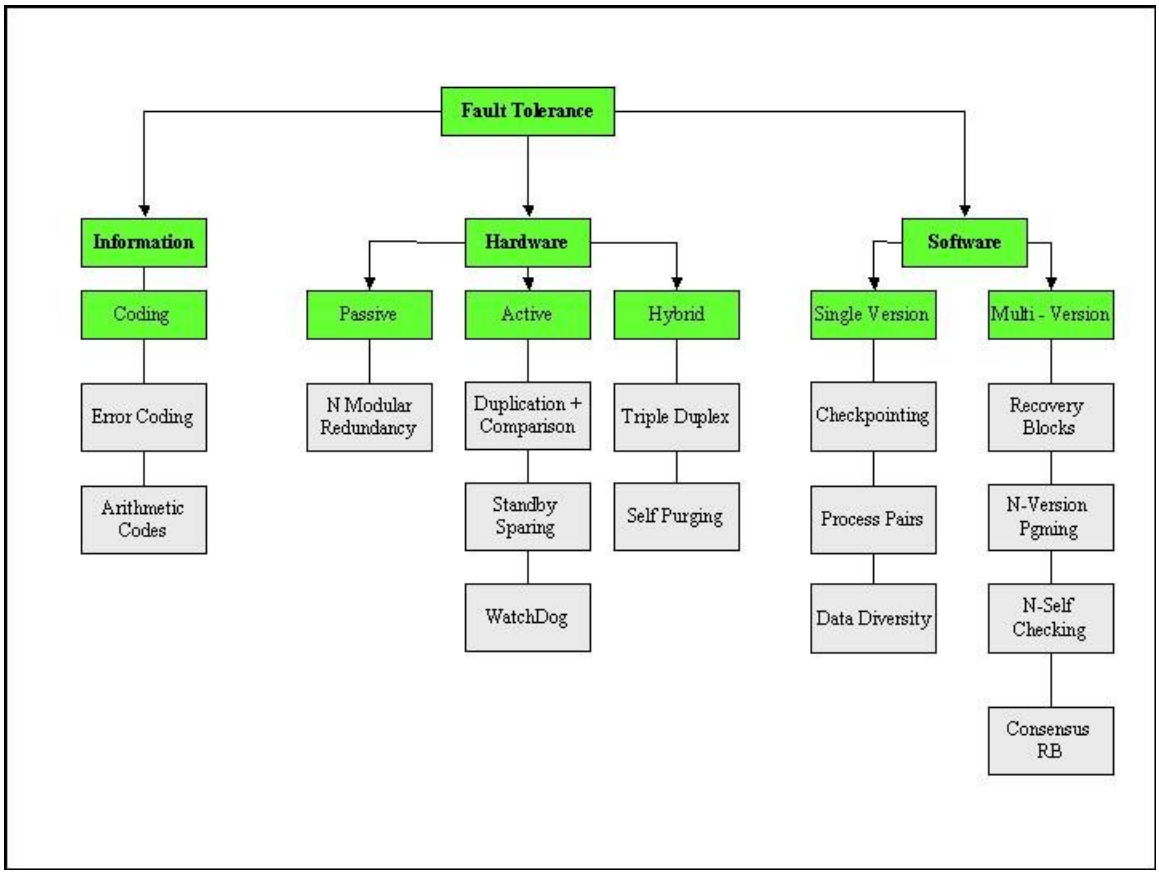


Figure 2: A Partial Taxonomy of Fault Tolerance Techniques

### *Information Redundancy*

Information techniques provide fault tolerance through the application of error coding to operations and data. As a result, this class will be a sub-component of either hardware or software techniques when implemented.

The information class includes error coding and arithmetic codes. Error coding applies various schemes (Berger codes, cyclic redundancy, simple parity, etc) to provide a measure of error detection and correction to data at rest and in-transit.

Arithmetic codes (e.g., AN, RESO, etc) are used to check the results of arithmetic operations. In both cases, there is an additional computation and storage overhead required in order to store and process the redundant information necessary to provide the error detection and correction.

### *Hardware Based Fault Tolerance*

Hardware fault tolerance can be passive, active, or a hybrid of both approaches. In the passive class, *fault masking* is used to hide the occurrence of a fault. N-Modular Redundancy (NMR) is shown in Figure 3 as a representative example of the passive class. In NMR, processing modules are replicated and allowed to run in parallel on the same I/O stream. A "voter" compares the output of all modules and determines if they are all the same. If they are not, the voter selects the output value by simple majority. Triple-Modular Redundancy (TMR) is a common implementation of NMR since this is the minimum number of modules required in order to implement the approach (the number of modules must be odd in order for the voting mechanism to identify a faulty module).

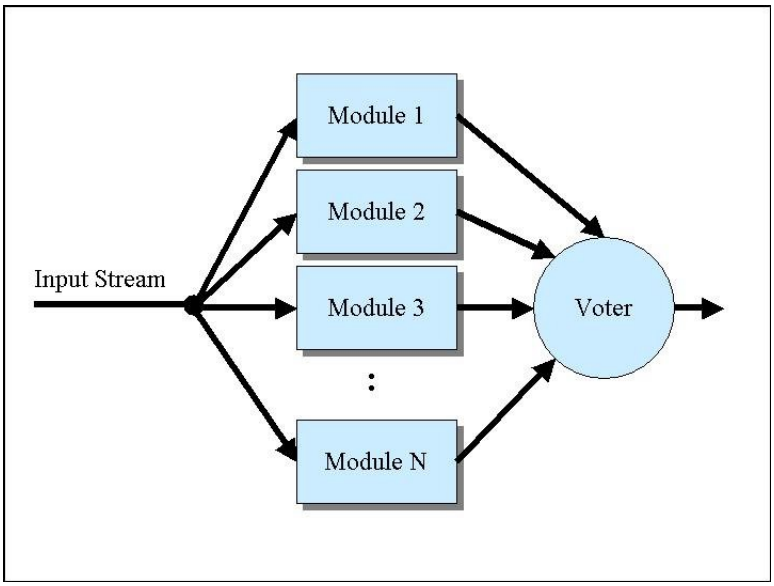


Figure 3: N-Modular Redundancy

The active class is characterized by detecting faults as well as acting to remove them from the system. Figure 4 shows an example of standby sparing in which a module is replicated but remains in "hot-standby" mode. When the main module detects that an error has occurred, it switches execution control to the standby.

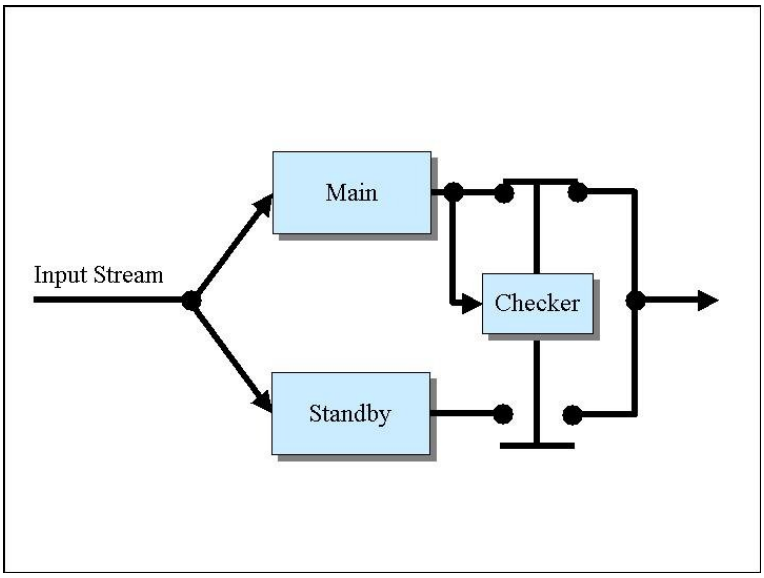


Figure 4: Standby Sparing

Of significance, this approach requires that at least the main module be *self-checking* in order to determine the existence of error conditions. To alleviate that requirement, *Duplication with Comparison* can be used. This approach is equivalent to NMR but with two modules instead of three. Therefore, the voter is capable of determining that an error has occurred but not by which module. In order to be an active technique, some mechanism of identifying the failed module must be employed [SEI90].

Finally, hybrid approaches attempt to draw advantages from both active and passive techniques. For example, Figure 5 (from [NAY06]) shows a self-purging technique that feeds the output of multiple, duplicate modules into a voter circuit. The voter masks faults by simple majority and the voter output is fed back to a comparator switch. If a module's last output was not the same as the value propagated by the voter, then the comparator switch decouples the module from the voter and signals a fault. Of course, greater fidelity would be expected of the comparator switch in order to prevent transient faults from decoupling modules when the error is not persistent.

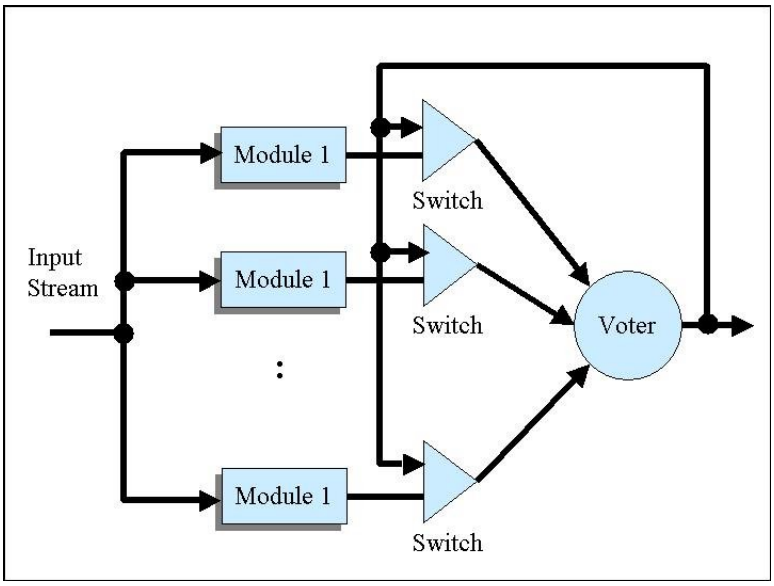


Figure 5: Self-Purging Hybrid [NAY06]

It should be noted that information and hardware redundancy techniques are being closely examined in order to provide some relief from a transient faults at the microprocessor gate-level logic ([WEI06], [REI05] [SAHA06]. Shrinking feature sizes and lower switching thresholds appear to be making highly dense processors susceptible to transient faults caused by cosmic (and other) radiation and electro-magnetic interference.

### *Software-Based Fault Tolerance*

Software-based fault tolerance can be divided into two classes depending on the number of *versions* of a module that have been created. In the Single Version class, only a single version of the software or module is available and fault tolerant techniques must be implemented solely within that module.

Checkpointing and restart and Process Pairs ([BOW93][NASA00][SEI91]) are two methods of providing software based fault tolerance within a single version of software. Checkpointing involves taking periodic snapshots of system state and storing these away for later use. When an error is detected, the fault tolerant response is to make use of the last checkpoint to restart the process from a correct state. In some implementations (e.g. rollforward checkpointing as described in [PRA94]), checkpoint data is used to allow recovery without moving the process back in time. Checkpointing is one of the few general approaches that can be used within a single version of a software module since it is independent of the cause of the fault. It is most successful in dealing with transient faults that appear, cause an error, and disappear just as quickly [NASA00].

Process pairs in software are implemented by forcing a process to execute on two separate processors – a main and an alternate. For example, in the TANDEM Nonstop architecture ([BAR90],[SEI90]), a main processor executes code while periodically transfering checkpoint information (process state) to an alternate processor. The alternate receives and stores the checkpoint information and monitors for error conditions. If an error condition does occur on the main processor, the alternate loads the last checkpoint state from its memory, starts its own "alternate" on another processor, and recommences execution.

In the Multi-Version class, the idea is to compose a software module using different design teams, design techniques, and/or programming languages. Since the modules are designed to produce the same results, it should be possible to compare them in operation and use a voting mechanism to determine the correct output. This technique aims to reduce coding and logic errors by design diversity. N-Version programming is the canonical application of the multi-version class [XIE??].

The Recovery Block approach is the software analogue to hardware standby sparing [XIE??]. In this approach, N versions of a software module are created using different methodologies. The most efficient of these modules is designated as the primary and the output of the module is subjected to an acceptance test as it executes. If the module fails the acceptance test, the process state is rolled back to some safe point and processing is retried on the highest ranking (next most efficient) alternate. This process continues until correct processing occurs or the system fails out.

## Section II: MILSHAP Fault Tolerance Analysis

In this section, we examine hardware and software fault tolerant computing techniques within the constraints of a MILSHAP architecture. As previously discussed, the primary design objective is to achieve the delicate balance between the functionality required to provide fail-secure, fault-tolerant operation and the ability to develop assurance in that functionality. As established earlier, fault tolerance is a necessary (but not sufficient) element of assurance whose implementation can adversely affect the code size and complexity of the system from a certification point of view.

Each subsection starts with a discussion of the applicability of some of the general techniques described in Section I. This is followed by the introduction and description of a few interesting areas of fault tolerance research and an elaboration on how they do or do not fit into the MILSHAP constrains.

### Hardware Fault Tolerance

#### *General Approaches*

Hardware fault tolerance can be employed at many abstract levels. It has already been mentioned that several forms of information and hardware fault tolerance are being used at the gate level on memory and CPU chips to combat a rising rate of transient faults. Programmable logic such as FPGAs can incorporate fault tolerant functionality (for example, [YU01] demonstrates TMR, NMR, duplicate and compare, TMR-Simplex and TMR-Duplex on FPGAs).

Beyond manufacturer gate-level CPU logic, FPGAs and ASICs, hardware redundancy on commodity assets for MILSHAP only makes sense when referring to the use of multiple CPUs or multiple cores within a CPU (or both). Some higher end desktop motherboards and many server motherboards are designed with multiple CPUs on-board. However, the MILSHAP requirement is to reduce the total hardware footprint and overall cost (in terms of money and power) for MILS computing while retaining the ability to process in multiple security domains. Therefore, the use of multi-CPU motherboards or multiple single-CPU motherboards on a common back-plane must be balanced against the fault tolerant return on investment.

If multiple CPUs are available and are addressable on commodity hardware, then a separation kernel can be written to make use of them in any one of the generic fault tolerant modes (e.g. NMR/TMR, standby sparing, duplication and comparison, etc). However, the coupling of these CPUs (synchronization and implementation of voters) would be a required function of the separation kernel instead of dedicated logic. This may make the kernel too complex to evaluate and trust.

The alternative is to custom design small form-factor motherboards from generic components and incorporate custom chipsets (e.g. ASICs, FPGAs) to implement true hardware based TMR/NMR and standby-sparing. The idea would be to make the motherboard look like a single logical processor to the separation kernel and hide the details of the redundancy in the customized logic. From an assurance point of view, this may be the preferred methodology to achieve processor level redundancy since the separation kernel and motherboard can be evaluated separately. The evaluation of the motherboard should be considerably easier since, like its commodity counterparts, the motherboard is not responsible for separating the security domains.

### *Case Study: Fault Tolerant Techniques for FPGAs for Permanent Faults*

Traditional approaches to chip level fault tolerance has relied on information redundancy techniques (e.g., error correcting codes, residue coding, etc) to detect and correct faults. In [CHE06], a survey and comparison of fault tolerance techniques for FPGAs is conducted. This survey focuses on techniques that can tolerate permanent faults but do not require chip-level redundancy (i.e., duplication of the FPGA).

System designers have great flexibility with the implementation of chip-level fault tolerance when ASICs or FPGAs are used to design custom chipsets for specific functionality. In general, custom chipsets and hardware are to be avoided where possible in order to lower the total system cost of ownership and potentially vendor lock-in. However, at both NSA and CSE (the headquarters for national certification of classified systems in the U.S. and Canada respectively), there is a quiet discomfort involved when performing certain operations by any means other than a dedicated hardware component (for example, cryptographic functions). The result is that these functions are invariably committed to specialized chips (ASICS or FPGAs if some reconfiguration is expected in the future).

The survey in [CHE06] considers *device-level* fault tolerance (which is used to increase the FPGA production yield and is therefore of not discussed here) and *configuration level* fault tolerance wherein a “system function” (the function that the device is to perform) is mapped to a set of fault-free resources on the FPGA. When a fault condition is detected, it is removed by re-mapping the system function to non-faulty components. As the authors indicate, the act of re-mapping generally requires the intervention of an external processor although several of the surveyed works attempted to minimize this intervention.

Off the works surveyed, the one that makes the most sense in terms of a high assurance environment is the column-based approach ([HUA01]). In this approach, the “system function” is first decomposed into a number of component functions such that each is capable of fitting into a single column on the FPGA.

The requirement is that at least one column on the FPGA is left unused. Then, using automated FPGA tools, a number of alternative design configuration files are generated such that the location of the spare column(s) is shifted within the design. This concept is shown in Figure 6.

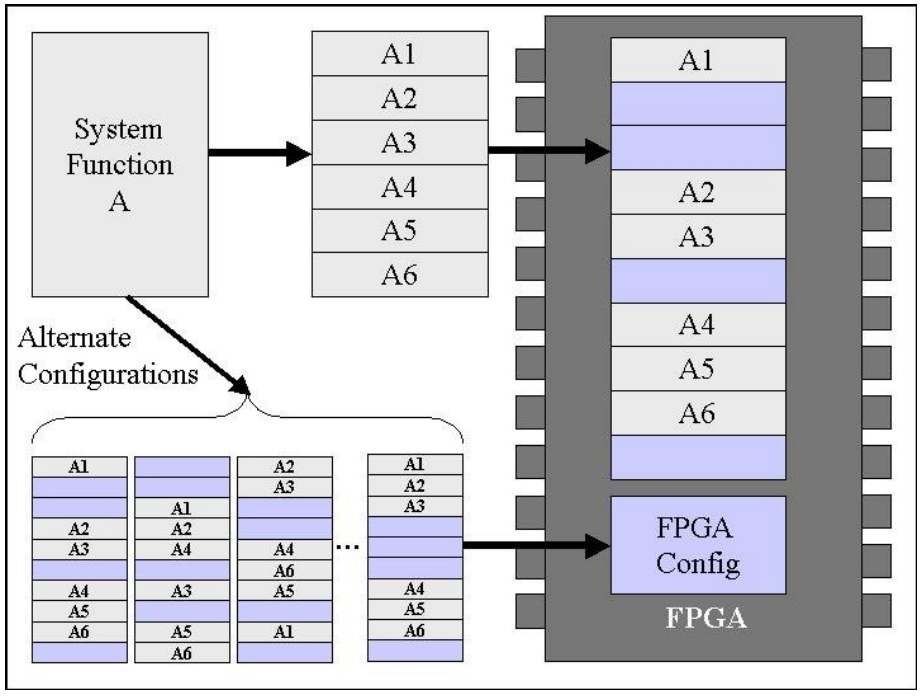


Figure 6: Column-Based FPGA Fault Tolerance

When a fault is detected in a component function, one of the alternate configuration files is selected from the pre-compiled set such that a spare column appears in the column where the fault occurred. The time required to swap the configuration files is short which makes this approach very attractive. The swap time can be made shorter by coding and storing only difference information between configuration files since these files are very similar in nature. Finally, the logic needed to decode and install the configuration file is non-complex and can be implemented easily on the FPGA.

The survey performed in [CHE06] did not consider fault detection schemes in relation to the fault tolerance mechanisms presented. However, detection mechanisms could be part of the design and can include any of the generalized approaches such as TMR, NMR, duplicate and compare or hybrids such as TMR-Simplex and TMR-Duplex [YU01].

Any of the configuration-level fault tolerance methods described in [CHE06] is a potential candidate for use in a MILSHAP environment where FPGAs are required. However, those with the least additional processing overhead and complexity will obviously place better from an assurance perspective. The column-based approach was the least complex to reconfigure (O(1)) whereas the

approach described as Pebble-Shifting had a reconfiguration complexity of $O(P^3)$ where P is the number of programmable logic blocks in the FPGA.

## Software Fault Tolerance

### *General Approaches*

*N-Version programming [TAY99][XIE??],*

As stated earlier, it is assumed that all developmental errors will be eliminated by keeping the separation kernel code small, carefully scrutinizing that kernel, and adhering to good secure design principles (e.g. [BIL05]). Not only must the software code size and complexity be minimized for inspection purposes, the design and coding must be performed in a very controlled fashion to ensure that the entire design path is assured. Doing this multiple times with different design teams and design tools threatens that assurance path and can be cost-prohibitive for very little real return. Therefore, N-Version programming is not viewed as a necessary (or acceptable) approach to software fault tolerance for MILSHAP design.

*Recovery Blocks*

As explained in Section I, recovery blocks are similar to N-Version programming in that multiple modules are coded using different algorithms and designs to arrive at the same solution. However, the modules are ranked and are not executed in parallel – they are the software allegory of hardware-based standby sparing [XIE??]. As each module fails its acceptance test, control is passed to the next module. Recovery blocks in limited form would be suitable for implementation on the most important elements of a separation kernel such as key generation or hash verification as long as these elements were small and as long as there was not security reason to not restart. However, recovery blocks do require a form of checkpoint and recovery in order to be able to back out of a module that has failed an acceptance test. The overhead created by the implementation of even simple checkpointing (e.g. synchronization of data and I/O) may be overkill for the perceived fault tolerant benefit.

### *Case Study: Loosely-Synchronized Redundant Virtual Machines ([COX06])*

A very interesting approach to fault tolerance appears in [COX06]. This approach, called Loosely-Synchronized, Redundant Virtual Machines (LSRVM) is the software analog of hardware N-Modular redundancy but with the significant difference that the modules are entire virtual machines (which include guest operating system and all applications) instead of individual system components or subsystems.

This approach targets transient faults attributable to shrinking feature sizes and noise margins (as discussed earlier) and specifically targets commodity computing in an attempt to avoid expensive, custom design.

The architecture described in [COX06] is based on loose-lockstep processing as implemented in the Hewlett-Packard Nonstop Advanced Architecture (NSAA) [NON05]. Loose-lockstep processing involves the parallel execution of multiple copies of an application and its operating system on different processing units. Synchronization for the purposes of error detection and correction occurs only during I/O and interrupt operations since these are deemed to be the only times when such checks are absolutely required. This is in contrast to tightly coupled approaches that verify the correct operation of each processing unit on an instruction-by-instruction basis and require very stringent timing and coordination as well as significant custom chip-level design.

As indicated in [COX06], loose-lockstepping requires that robust mechanisms for I/O voting, interrupt synchronization, and recovery exist. The general concept is shown in Figure 7.

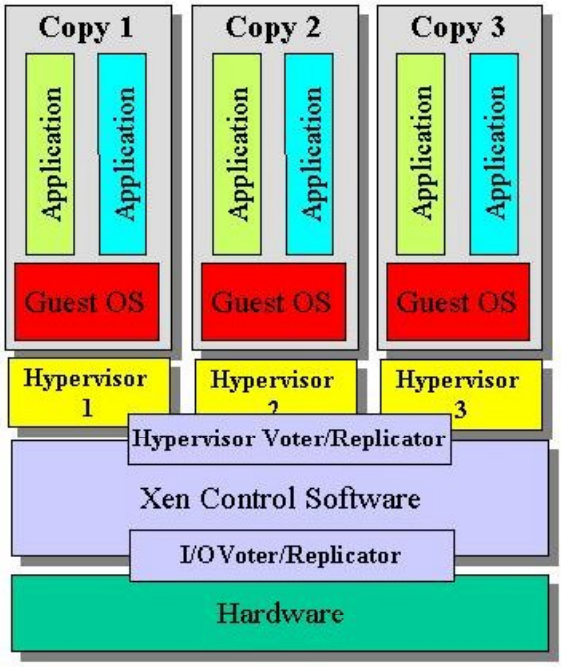


Figure 7: Loosely Synchronized Redundant Virtual Machines

In the LSRVM architecture, Xen control software creates a separate hypervisor for each redundant copy of an execution unit (Figure 7 shows three). When a hardware element generates an interrupt, it is trapped by the Xen control software which then requests each hypervisor to halt at the next common Voluntary Rendevous Opportunity (VRO). VRO's are code segments embedded throughout the individual hypervisors that allow the hypervisor to pause in order to service interrupts. When all hypervisors are synchronized at the same VRO, the Xen control software passes the interrupt value to each of them and processing continues.

When hypervisors attempt to write to memory or to external interfaces, the hypervisor voter/replicator or I/O voter/replicator parts of the Xen control software pause execution, wait until all values have been supplied by the redundant copies, determines (by simple majority vote) which values are correct, and then writes the correct value and releases the hypervisors to continue execution. If one of the redundant copies presents an incorrect value, the copy is assumed to have been corrupted by a fault and is terminated. In this case, a new redundant copy is started from one of the correct copies through Xen's built in live migration facility. This facility allows a process to copy itself from one virtual machine to another in near-real time.

This work represents one of the closest matches to the type of fault tolerance that could be used within a separation kernel environment. Each of the redundant copies are exact duplicates of a single code base which means that N-modular redundancy could be achieved after having evaluated only one copy. The fact that this approach is decidedly independent of specialized hardware (other than commodity CPU support for virtualization) is a significant advantage.

However, this architecture was designed with the concept of executing a single or small group of applications within a single security domain. In a MILSHAP environment, at least two isolated security domains are necessary with several other independent domains being required for specialized security functionality. This translates to additional complexity for the LSRVM architecture such as shown in Figure 8 (the figure portrays a representation of what the complexity could be like).

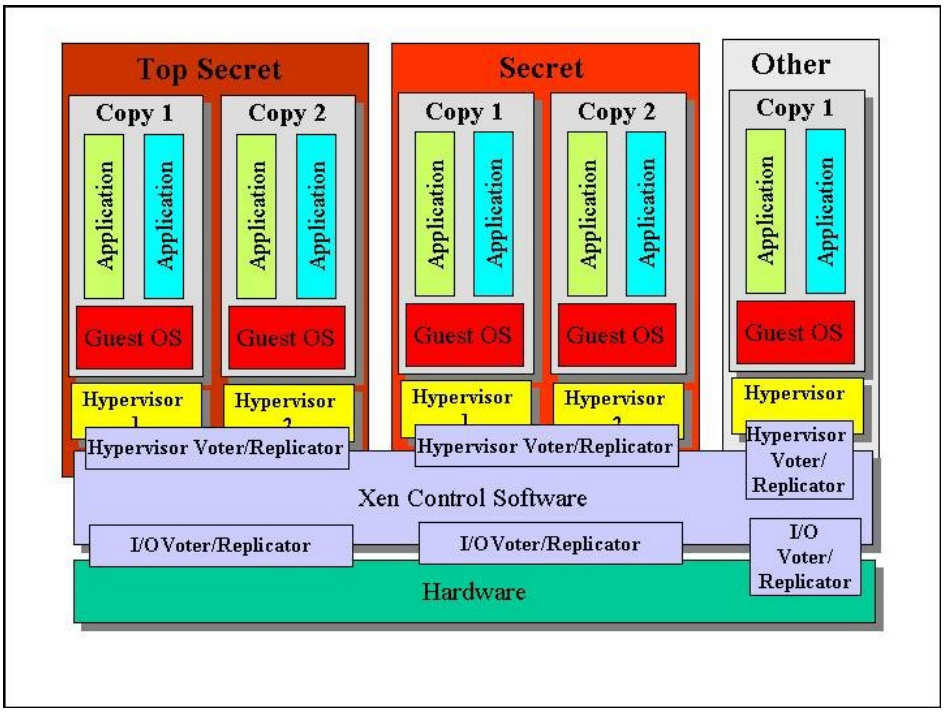


Figure 8: LSRVM Multi-Domain Architecture

The hypervisor voter/replicators for each domain (Top Secret, Secret, Other) must operate independently since each domain should not be aware of any others except through tightly controlled inter-process communication. It is therefore unclear how the complexity of the Xen control software would scale with this additional demand. It is also not clear if a single CPU core could support the context switching demand implied by the additional domains. Despite these apparent unknowns, LSRVM still holds great promise for fault tolerance within a MILSHAP environment.

### *Case Study: SoftWare Implemented Fault Tolerance ([REI05])*

The work described in [REI05] represents an effort to make use of intelligent compilation to build fault tolerant design into software. SWIFT targets “soft” or transient faults on commodity computing platforms that have minimal hardware fault tolerance (which is most of the desktop and laptop market).

SWIFT is an evolution of some clever work done previously in software fault tolerance known as EDDI [EDDI02]. The basic fault detection mechanism behind EDDI is code duplication by the compiler. As a program is compiled, a duplicate is automatically generated with the condition that it uses completely different memory locations and registers to perform the same function. The compiler inserts synchronization points into both the original and the duplicate to allow both to compare results at certain points. Since compilers can perform

optimization, it is possible to schedule the redundant copies such that interference between them is minimized.

The authors of SWIFT made several adjustments to the EDDI algorithm to achieve faster performance, higher control flow error coverage, and smaller overall code size. First, they removed all store instructions from duplicated code modules in order to reduce execution time and cache consumption. This adjustment is possible because the main and duplicate modules always compare results just before storing them so that only the correct result need be retained. The second adjustment involves the use of control flow signatures and other enhancements to control flow verification such that more control flow errors can be detected. The complete SWIFT methodology involves making several performance enhancements to the adjusted EDDI algorithm.

In terms of benchmark performance against the un-enhanced, adjusted EDDI, SWIFT achieves a static size reduction of 13%. However, this is still 2.4 times larger than the equivalent code without any fault tolerance code. Swift also achieves a slightly lower execution time with respect to the adjusted EDDI.

Applied to a MILSHAP environment, SWIFT holds some potential. It is a thoroughly “free” approach in that no custom hardware is required and no assumptions about the availability of CPU features is made. Its error detection and correction performance is certainly better than having no fault tolerance with negligible performance overhead.

However, in MILSHAP, the separation kernel or VMM would be the target source and this would likely be hand optimized already. It is not immediately clear that putting the kernel source through the SWIFT transformation would be successful and, if it was, it is not clear if a code size increase of 2.5 times would be acceptable from an assurance point of view. Of course, if some assurance could be gained in the compiler optimization mechanism, then the original kernel source need only be assured and the resulting object code spot-checked for correctness.

## Section III: Conclusion

In this paper, we introduced the concept of Multiple Independent Levels of Security High Assurance Platforms (MILSHAP) as a potential solution to Multi-Level Secure (MLS) requirements on commodity hardware. We identified a need for MILSHAP within the military community and then determined the requirement for fault tolerance within this architecture. It was determined that assurance arguments are the single most important factor in developing such platforms given that classified domains are to be executed together on the same hardware. The Common Criteria Protection Profile for Separation Kernels in High Robust Environments is a specification that seeks to standardize the type and nature of the assurance arguments that are to be provided in order to succeed in developing a MILSHAP. A brief introduction to taxonomies of faults and fault tolerance techniques was provided. Finally, we analyzed both general and case studies in fault tolerance against the requirements and constraints within a MILSHAP environment.

## References


[5200.28] Department of Defense 5200.8-STD, "Trusted Computer System Evaluation Criteria (TCSEC)," National Computer Security Center, U.S. Department of Defence, August 15, 1983.

[8500.2] Department of Defense Instruction (DoDI) 8500.2 "Information Assurance (IA) Implementation", 6 Feb 2003, http://www.dtic.mil/whs/directives/corres/html/85002.htm

[AK05] Ankrum, T., and Kromholz, A., "Structured Assurance Cases: Three Common Standards," *Proc of 9th IEEE International Symposium on High-Assurance Systems engineering (HASE'05),* 12-14 Oct 2005, pp. 99-108.

[ALRL04] Avizienis, A., Laprie, J.C., Randell, B., and Landwehr, C., "Basic Conepts and Taxonomy of Dependable and Secure Computing", *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, No. 1, Jan-Mar 2004.

[BAR90] Bartlett, J., Bartlett, W., Carr, R., Garcia, D., Gray, J., Horst, R., Jardine, R., Lenoski, D., McGuire, Dix., "Fault Tolerance in Tandem Computer Systems," Technical Report 90.5, May 1990. www.hpl.hp.com/techreports/tandem/TR-90.5.html

[BIL05] Benzel, T., Irvine, C., Levin, T., Bhaskara, G., Nguyen, T., and Clark, C., "Design Principles for Security", Naval Postgraduate School Tech Report NPS-CS-05-101.

[BGM06] Bloomfield, R., Guerra, S., Masera, M., Miller, A., and Saydjari, O., "Assurance Cases for Security", *Workshop on Assurance Cases for Security*, Workshop Report v01c, Washington, DC, 13-15 June, 2005.

[BGMW06] Bloomfield, R., Guerra, S., Masera, M., Miller, A., and Weinstock, B., "International Working Group on Assurance Cases (for Security)", *IEEE Security & Privacy*, vol. 4, no. 3, May/June 2006, pp. 66-68.

[BOW93] Bowen, N., and Pradhan, D., "Processor- and Memory- Based Checkpoint and Rollback Recovery," *IEEE Computer*, Vol. 26, No. 2, July 1993, pp. 22-31.

[CHE06] Cheatham, J., Emmert, J., and Baumgart, S., "A Survey of Fault Tolerant Methodologies for FPGAs," *ACM Transactions on Design*

*Automation of Electronic Systems*, Vol. 11, No. 2, April 2006, pp. 501-533.

[EDDI02] Oh, N., Shirvani, P., and McCluskey, E., "ED4I: Error Detection by Diverse Data and Duplicated Instructions," *IEEE Transactions on Computers ,* Vol. 51, No. 2, February 2002, pp. 180 – 199.

[FOSS04] Alves-Foss, J., Taylor, C., and Oman, P., "A Multi-Layered Approach to Security in High Assurance Systems," *Proc. Hawaii International Conference on System Sciences*, Jan. 2004, pp. 90302.2

[GOS04] Gossens, S., and Dal Cin, M., "Structural Analysis of Explicit Fault-Tolerant Programs," *Proc of 8th IEEE International Symposium on High-Assurance Systems engineering (HASE'04),* 25-26 March, 2004, pp. 89-96.

[HAND05] Hand, S., Warfield, A., Fraser, K., Kottsovinos, E., and Magenheimer, D., "Are Virtual Machine Monitors Microkernels Done Right?", *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Sante Fe, NM, USA, June 2005.

[HEI06] Heiser, G., Uhlig, V., and LeVasseur, J., "Are Virtual-Machine Monitors Microkernels Done Right?", *ACM SIGOPS Operating Systems Review*, Jan 2006, Vol. 40, No. 1, pp. 95-99.

[HHO05] Harrison, W., Hanebutte, N., Oman, P., and Alves-Foss, J., "The MILS Architecture for a Secure Global Information Grid", *CrossTalk: The Journal of Defense Software Engineering*, Vol. 18, No. 10, Oct. 2005.

[HOH04] Hohmuth, H., Peter, M., Hartig, H., and Shapiro, J., "Reducing TCB Size by Using Untrusted Components – Small Kernels Versus Virtual Machine Monitors," *Proc. 11th ACM SIGOPS European Workshop*, Leuven, Belgium, 2004.

[HUA01] Huang, W.-J. and E.J. McCluskey, "Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance," *Proc. 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2001, pp. 137-146

[HUN04] Hunstad, A., Hallberg, J., and Andersson, R., "Measuring IT Security – a Method Based on Common Criteria's Security Functional Requirements," *Proc IEEE Workshop on Information Assurance*, USMA, West Point, NY, 10-11 June 2004.

[JAC04] Jacob, J., "High Assurance Security and Safety for Digital Avionics," *Proc IEEE Digital Avionics Systems Conference*, 2004 (DASC'04), Vol. 2, pp. 8.E.4-8.1-9, Oct, 2004.

[KAN98] Kang, M., Moore, A., Moskowitz, I., "The Design and Assurance Strategy for the NRL Pump," *IEEE Computer*, Vol. 31, No. 4, April 1998, pp. 56-64.

[KAR05] Karger, P., "Multi-Level Security Requirements for Hypervisors", *Proc. 21st Annual Computer Security Applications Conference (ACSAC 2005)*, Tucson, AZ, 5-9 Dec 2005.

[L4] http://os.inf.tu-dresden.de/L4/

[LI06] Li, A., Hong, B., "A Low-Cost Correction Algorithm for Transient Data Errors," *Ubiquity*, Vol. 7, No. 21, 30 May – 19 June 2006, pp. 2-15.

[MACH] http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html

[MINIX] http://www.minix3.org/

[NASA00] Torres-Pomales, W., "Software Fault Tolerance: A Tutorial," NASA Technical Report TM-2000-210616, October 2000.

[NAY06] Nayak, A. Course Notes CSI5134, Fall 2006, University of Ottawa.

[NEU06] Neumann, P., "System and Network Trustworthiness in Perspective," *ACM Conference on Computer and Communications Security 2006 (CCS'06),* 30 Oct – 3 Nov 2006, Alexandria, Virginia, U.S.A.

[NON05] Bernick, D., Brukert, B., Del Vigna, P., Garcia, D., Jardine, R., Klecka, J., Smullen, J., "NonStop Advanced Architecture," *Proc. International Conference on Dependable systems and Networks 2005 (DSN'05)*, 28 June-1 July 2005, pp. 12-21.

[OMG05] Uchenick, G., "Multiple Independent Levels of Safety and Security (MILS): High Assurance Architecture" (presentation), *Proceedings of Object Management Group Workshops*, Arlington, VA, USA, July 11-14, 2005. Accessed 25 Sept 06 at www.omg.org/ news/ meetings/ workshops/ rt_embedded2005.htm

[PRA94] Pradhan, D., Vaidya, N., "Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture," *IEEE Trans. On Computers*, Vol. 43, No. 10, Oct 1994, pp. 1163-1174.

[REIT94] Reiter, M., Birman, K., and Van Renesse, R. “A Security Arcitecture for Fault-Tolerant Systems”, *ACM Trans. on Computer Systems*, Vol. 12, No. 4, November 1994, pp. 340-371.

[REI05] Reis, G., Chang, J., Vachharajani, N., Rangan, R. August, A., “SWIFT: Software Implemented Fault Tolerance,” *Proc. IEEE International Symposium on Code Generation and Optimization (CGO’05)*, 2005.

[ROS05] Rosenblum, M., and Garfinkel, T., “Virtual Machine Monitors: Current Technology and Future Trends”, *IEEE Computer*, Vol. 39, Issue 5, May 2006, pp. 44-51.

[RUS81] Rushby, J. “The Design and Verification of Secure Systems.” *ACM Operating Systems Review*, Vol. 15, No. 5, pp. 12-21, 1981.

[RUS89] Rushby, J., “Kernels for Safety?” in *Safe and Secure Computing Systems*, T. Anderson, (ed), Blackwell Scientific Publications, 1989, chapter 13, pp. 210-220.

[SAHA06] Saha, G. “Software Based Fault Tolerance – A Survey,” *Ubiquity* Vol. 7, No. 25, July 5, 2006

[SAY04] Saydjari, O., “Multilevel Security: A Reprise”, *IEEE Security and Privacy*, Vol. 2, No. 5, Sept-Oct 2004, pp. 64-67.

[SEI90] Seiwiorek, D., “Fault Tolerance in Commercial Computers,” *IEEE Computer*, Vol. 23, No. 7, July 1990, pp. 26-37.

[SEI91] Siewiorek, D., “Architecture of Fault-Tolerant Computers: An Hisotorical Perspecitive,” *Proc of the IEEE*, Vol., 79, No. 12, December 1991.

[SKPP] U.S. Government Protection Profile for Separation Kernels in Environments requiring High Robustness, Version 0.621, July 2004, niap.nist.gov.

[SNO05] Snow, B., “We Need Assurance!”, *Proc. 21st Annual Computer Security Applications Conference (ACSAC 2005)*, Tucson, AZ, 5-9 Dec 2005.

[TAN06] Tanenbaum, A, Herder, J., and Bos, H., “Can We Make Operating Systems Reliable and Secure?”, *IEEE Computer*, May 2006, Vol., 39, No. 5, pp.44-51.

[TAY99] Taylor, D., “Practical Techniques for Damage Confinement in Software”, *Proc. Computer Security, Dependability, and Assurance: From Needs to Solutions 1998*, 7-9 July 1998, 11-13 November 1998, pp. 132-143.

[UHL05] Uhlig, R., Rodgers, D., Santoni, A., Martins, F., Anderson, A., Bennett, S., Kagi, A., Leung, F., Smith, L., “Intel Virtualization Technology,” *IEEE Computer*, May 2005, Vol., 38, No. 5, pp.48-56.

[WEI06] Chen, W., Gong, R., Dai, K., Liu, F., and Wang, Z., “Two New Space-Time Triple Modular Redundancy Techniques for Improving Fault Tolerance of Computer Systems,” *Proc of 6th IEEE International Conference on Computer and Information Technology (CIT’06)*, Sept 2006, p. 175

[XIE??] Xie, Z., Sun, H., Saluja, K., “A Survey of Software Fault Tolerance Techniques,” found at http://homepages.cae.wisc.edu/ ~ece753/INFO.html (does not appear to be published)

[XIE05] Xie, Y., Spainhower, L., Narayanan, V., Mitra, S., “Tutorial: Robust System Design from Unreliable Components,” *32nd Annual International Symposium on Computer Architecture*, 4-8 June 2005, Wisconsin USA. http://www.cse.psu.edu/~yuanxie/ISCA-tutorial.html.

[YU01] Yu, S. and McCluskey, E., "Permanent Fault Repair for FPGAs with Limited Redundant Area," Stanford CRC TR 01-2, 2001.