D 21.1

MILS Architecture

Project number:	318353
Project acronym:	EURO-MILS
Project title:	EURO-MILS: Secure European Virtualisation for Trustworthy Applications in Critical Domains
Start date of the project:	1st October, 2012
Duration:	36 months
Programme:	FP7/2007-2013

Deliverable type:	Report
Deliverable reference number:	ICT-318353 / D21.1 / 1.0
Activity and Work package contributing to the deliverable:	Activity 2 / WP 21
Due date:	September 2013 – M12
Actual submission date:	30th September, 2013

Responsible organisation:	SYSGO
Editor:	SYSGO (Holger Blasum)
Dissemination level:	Public
Revision:	1.0

Abstract:	We give a generic description of MILS systems, our MILS architecture template and discuss MILS main components.
Keywords:	MILS architecture, software architecture, information flow, resource management

Editor

Holger Blasum (SYSGO AG)

Contributors (ordered according to beneficiary number)

Sergey Tverdyshev, Holger Blasum (SYSGO AG),
Bruno Langenstein (DFKI / Deutsches Forschungszentrum für künstliche Intelligenz),
Jonas Maebe, Bjorn De Sutter (Universiteit Gent),
Bertrand Leconte, Benoît Triquet (AIRBUS),
Kevin Müller, Michael Paulitsch (EADS Deutschland GmbH),
Axel Söding-Freiherr von Blomberg (OpenSynergy GmbH),
Axel Tillequin (EADS France SAS)

Acknowledgment

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 318353.

Executive Summary

We introduce a generic description of MILS systems (Chapter 2), and the MILS architecture template (Chapter 3). Chapter 4 discusses MILS main components. The practical aim of this document is two-fold: (1) to get a common understanding of MILS terms and definitions, and (2) to provide a framework to derive the information flow, access control and resource allocation of the demonstrators from individual MILS components.

Contents

Chapter 1 Introduction

EURO-MILS claims that "*the project's cornerstone is MILS (Multiple Independent Levels of Security), a high-assurance security architecture that supports the coexistence of untrusted and trusted components, based on verifiable separation mechanisms and controlled information flow*" [Cordis12]. While MILS is well established in practice, and products claiming MILS compliance do exist since the mid 2000s, it so far has not been standardized or given a formal definition, in particular "*there is no standard that defines which functionalities reside in a MILS-compliant system and how a MILS kernel should be designed.*" [DPF09, p. 4].

In the absence of such a standard, to reflect meaningfully MILS, a common understanding of some terms related to "architecture" is helpful. We introduce a generic description of MILS systems (Chapter 2), and the MILS architecture template (Chapter 3). Chapter 4 discusses MILS main components.

Chapter 2 MILS concepts and state of the art

In publications on MILS such as [AFHOT06] reference is made to high-assurance safety (in particular avionics) and high-assurance security. We briefly recapitulate both backgrounds, based on well available material, without claiming to completely cover each development until the present. We also introduce certification and architectural decomposition and modelling aspects of MILS.

2.1 Modular high-assurance safety in avionics

Safety assurance levels: [ARP4754] introduces a notion of safety assurance levels: if the failure of an application would have an impact that causes severe damage (e.g. loss of aircraft), the application is of a high assurance level. Otherwise, if the failure of an application would have an impact that causes a minor nuisance (e.g. loss of passenger entertainment system), the application is of a low assurance level. Applications at a high safety assurance level have stronger process requirements (planning, development, verification) than applications at a low safety assurance.

IMA: Integrated Modular Avionics (IMA) is an architectural concept for modular avionics software systems that has been inspired from previous architectural concepts for physically modular hardware systems that consisted of LRUs (Line Replacement Units). IMA replaces multiple instances of separate and dissimilar LRUs with fewer common processing modules, and provides shared power supplies, housing and communication links. IMA decomposes an IMA *system* into (1) an IMA *platform* consisting of hardware and core software doing resource management and process scheduling, and (2) IMA *applications*, which are software components interacting with the IMA platform.

IMA systems are designed to host several applications with appropriate isolation on a set of shared hardware and software resources. In IMA, applications execute in an environment generally called a set of partitions. A *partition* is a unit of separation regarding resource (i.e., CPU, memory, etc.) allocation in space and time domains. The IMA architecture dictates the underlying operating system (OS) to be developed for hard real-time, safety critical avionics applications. One of the functional requirements applied to such an OS is to host multiple independent aircraft applications while the computing platform shall not introduce significant common failure modes between those applications; evidences of the mechanisms providing isolation between those applications shall be demonstrated. One upside is that this enables incremental qualification, under which one application can be upgraded without requiring the others to undergo new certification.

IMA-related standards include a common interface for applications [ARINC653], and guidance for the development and certification of systems [DO-297].

[ARINC653] requires an operating system to manage partitions and a rich set of interfaces to manage their inter-partition communication, periodic assignment of CPU time to a partition, applications (“processes”) within a partition, memory allocation, and a health monitor responsible for reporting hardware, operating system software and application failures. [ARINC653] provides implementable interfaces for the above-mentioned functionalities (e.g., parameters and return values including error codes are defined).

Processes for the system development, certification planning, requirements determination, safety assessment, implementation verification and process assurance have been developed for complex integrated systems in avionics in general [ARP4754]. Similarly, [DO-297] describes the IMA-specific aspects of design assurance for all parties involved in development, integration, verification and validation of IMA systems. As considerations of the IMA platform, [DO-297, p. 11] lists availability (functional performance and resource management, health monitoring), integrity (including protection features, fault detection and partitioning), safety (appropriate architecture and design assurance), fault management and composability. [DO-297, p. 14] defines the aim of "robust partitioning" to provide an equivalent level of functional isolation and independence as a federated system implementation. A partitioning analysis demonstrates that "no application or sub-function in a partition could affect the behaviour of a sub-function or application in another partition in an adverse manner". [DO-297] splits validation, verification, configuration management and certification processes into tasks done at the application level, the platform level, and the system level.

IMA design is made to provide high-assurance safety systems for avionic industry. However, IMA requirements and development do not include security aspects, only random hardware faults and involuntary design errors are considered without taking into account failures due to malicious actions.

2.2 Modular high-assurance computer security

Security assurance levels: In computer security, the Common Criteria for Information Technology Security (CC, [CC12]) standard states that owners of assets (something valuable, e.g. a component in an aircraft or important data) place value on the assets. The risk of a threat to an asset "*depends on the likelihood of the threat being realised and the impact on the assets when that threat is realised*" [CC12, Part 1, p. 39]. Similar to the concept of safety assurance levels, an application which, under attack, impairs assets of high value (e.g., confidentiality of top secret data, integrity of a critical system) needs to provide a high security assurance and an application which, under attack, impairs assets only of low value needs to provide a low security assurance. Security assurance levels for individual components are especially used for standards that typically analyze distributed systems such as [ARINC811] for avionics and [ISA62443] for industrial automation.

Evaluation assurance levels: However, there is an additional difference in computer security versus safety: safety assurance usually considers probabilities of faults (e.g., ARP 4761, ISO-26262), and in systems, combined and dependent probabilities (e.g. "fault tree analysis"). In computer security, security risks are more "all or nothing": for example, once an attacker knows that access to an asset is possible by exploiting two weaknesses successively, he/she will perform those actions in the required order. In particular this also holds if a larger system is incorrectly *specified*, and exploits against the larger system can be derived simply by analysis of the specification. In [CC12, Part 1, p. 41] the sufficiency of the countermeasures against a threat is thus shown by analysis in a document (the "Security Target"), and the correctness of a product is shown by evaluation in a graded evaluation process. If a product has undergone an extensive evaluation process, it gets assigned a high evaluation assurance level (EAL). If a product only has undergone a more limited evaluation process, it gets assigned a low evaluation assurance level.

Security policy and security policy levels: To build systems on consistent specifications, a security policy is imposed upon a system. A security policy often assigns *security policy*

levels to elements of a system. A widely applied security policy for confidentiality was Bell-LaPadula that assigned to each component a label indicating a security level such as "public", "classified", "secret", "top secret", where "public" is less than "classified", "classified" less than "secret" and "secret" less than "top secret". On such a policy, each pair of components can be compared (a set with such features is also called a "total order"). It allows implementing schemes such as Bell-LaPadula, which (in simple terms) says that no-one is allowed to "read up" (read information of a higher security level than his/her classification) or to "write down" (write information to a lower security level than his/her classification). Bell-LaPadula was also chosen as the reference model for the Orange Book [Dod83]. Similarly, the Biba integrity policy can be seen as inverting the labels ("no write up", "no read down").

Multi-level secure systems (MLS): An MLS system maintains multiple security policy levels at the same time, often by assigning security labels to its components and resources. Systems implementing the afore-mentioned Bell-LaPadula or Biba models have been called MLS systems [And08]. A broader definition of the term MLS will be discussed under "*MLS versus MILS nowadays"* below.

Operating systems: Much early work in high-assurance modular computer security has been on secure operating systems [MP97]. The earliest uses of computers involved programs directly operating on hardware, addressing individual memory cells directly and exclusively using the entire hardware. However, maintainability concerns lead to the development of a more modular usage, by installing an operating system on the hardware. An operating system is a software system that (1) simplifies access to underlying hardware by providing appropriate abstractions to applications, (2) provides resource management (e.g. memory) and in particular is able to allocate CPU(s) to applications (scheduling). Operating systems also can provide networking or file system infrastructure to computers.

Security kernels: Many secure operating systems have used security kernels [MP97]. Security kernels have a small implementation, and thus can be more easily reviewed than a complex operating system. Security kernels target integrity, availability, and (usually a lesser concern in safety) confidentiality of applications and data and impose a security policy on the system. Security kernels, for example Honeywell's Scomp [Fr83], supporting a security policy with multiple security policy levels had usually been subsumed under "multi-level secure" (MLS) systems.

In a security kernel, applications that are running at a certain security policy level fixed for each application are called "single-level secure" (SLS). If multiple instance of one SLS implementation are deployed in a system while each of those instances processes a different security level it will lead to "multiple single-level secure" (MSLS) components. Applications may implement security policies on completely different features than security kernels, so policies provided by applications versus policies provided by the separation kernel cannot always be directly compared. However, an implicit requirement on security kernels is that their security *assurance* level is at least as high as or higher than the highest security assurance level found in any application.

Classification of applications in a security kernel: Unless otherwise specified, the applications are SLS. Applications spanning multiple security policies are also MLS, such as a downgrader. For a collection of classifications, see Table 1. The underlying idea of such classification is that, from an information flow policy and resource sharing viewpoint only MSLS and MLS components need to be verified [AFHOT06].

SLS: Single-Level Secure Components [Alv98, AFHOT06, ZAV06]	A *Single Level Secure Component* is a component that every time processes data of one security level.
MSLS: Multiple Single-Level Secure Component [AFHOT06, ZAV06]	A *Multiple Single-Level Secure Component* is a special kind of SLS component that processes data of multiple security levels, but always maintains separations between classes of data by exclusively processing only one security level during its runtime instance. For example this separation can be implemented by allowing access to a different security level only when the component has rebooted with different parameters. Also deploying multiple instances of one SLS component processing different single security levels turn this SLS component into an MSLS component. Note: in [Alv98] this was restricted to temporal separation, "at a single time-point, only handles information from one component". If such a single-level process is to be implemented as untrusted process [Alv98], it can be supplemented by an appropriate labelling and filtering of messages. Moreover, in [Alv98] SLS and MSLS are subsumed under "secure single-level process".
MLS: Multi-Level Secure Component [Alv98, AFHOT06, ZAV06]	A *Multi-Level Secure Component* is a component that handles information of with different security levels concurrently during one runtime instance. An example of an MLS component is a separation kernel [MPT+12] or a downgrader [ZAV06].

Table 1: Levels of components

Multiple independent levels of security (MILS): Encoding rich functionality into a central component raises the question of how to design a security kernel that is itself secure. Therefore, the functionality of security kernels has been broken up into a more structured design. To differentiate such systems from "MLS" systems, the term "MILS" (multiple *independent* levels of security) has been introduced. It describes systems where different partitions hosting applications are either independent from each other or connected by communication channels without an explicit hierarchical ordering policy that would require attaching global security policy levels to each partition.

The MILS architecture approach was popularized by John Rushby in 1981 ([Rus81], "Design and Verification of Secure Systems"; at that time, Rushby did not use the term MILS), which started a formalisation of MILS concepts. In his approach, the system is designed as a distributed one and is based on a special kind of operating system using a separation kernel (SK). He proposed that the security should be achieved partly through physical separation, partly through the use of components and partly through trusted functionalities performed within some components. The purpose of the separation kernel is to allow such a "distributed" system to run within a single processor. This is achieved by offering a very strong separation between the different partitions except for very carefully controlled information flow between them.

The basic idea of MILS is to make the security-critical part of the system (i.e., SK) small enough and with specific functionality so it can be certified at high assurance levels.

Traditional operating system services like device drivers, file system, etc. are pulled out of the separation kernel and run in non-privileged mode; the only part of the MILS system running in privileged mode is the SK. Safety and security policies must be enforced at each level: by the separation kernel and by any other component needed by the applications hosted in the partitions, but also by the applications themselves. A key MILS objective is to enable the evaluation and certification of a complex system to be modularized into a number of independent, small evaluations.

MILS separation kernel security assurance characteristics: In practice, MILS principles largely match the requirements imposed by users and producers of IMA systems who, in addition to their IMA safety requirements, had an additional need for security requirements.

In the MILS literature, explicit concerns for security assurance have been formulated as "NEAT" [BBH+05, KW08, UV05], as follows:

- **Non-bypassable:** Policy enforcement functions cannot be circumvented.
- **Evaluatable:** Policy enforcement functions are small enough and simple enough that proof of correctness is practical and affordable.
- **Always Invoked:** Policy enforcement functions are invoked each and every time.
- **Tamperproof:** Policy enforcement functions and the data that configures them cannot be modified without authorization.

Similar definitions exist elsewhere, e.g. "evaluatable", "always invoked", "tamperproof" for reference monitors in [And72, p. 22].

Objectives and threats in MILS systems: In computer security, a *threat* is characterized by some adverse action achieved by an attacker who attacks system assets. The objectives of computer security are to counter threats in order to mitigate the risk of a threat scenario.

Assets for MILS system and its components can be formulated in a straightforward way:

- for each component itself,
  - with the objectives of the preservation of its confidentiality, integrity, and (possibly) availability,
- for each resource the component *uses*,
  - with the objectives of the preservation of its confidentiality, integrity, and (possibly) availability.

Threats can be named against the preservation of each the security attributes:

- for confidentiality, the threat is disclosure,
- for integrity, the threat is modification,
- for availability, the threat is depletion.

MLS versus MILS nowadays: Earlier in this section ("*Multi-level secure systems (MLS)*") a strict hierarchically ordered security policy based on security policy levels had been discussed in the context of MLS. One insight gained by the MILS approach was that several components on the same platform have safety and security requirements that are just "different" in a wider sense. This insight had led to (1) applying the term MLS also in that wider sense [DCS+04, LRP+11], and (2) to use MILS to describe an architectural decomposition approach of an MLS system into components [Alv98, AFHOT06, ZAF08].

For the rest of this document we use the term MLS for systems based in the wider sense (1) and MILS for the architectural decomposition approach (2).

2.3 Certification aspects

For IMA, DO-297 describes how to perform incremental certification [DO-297, WP08]. A case study on compositional certification of a system built on a separation kernel using Common Criteria approach is given in [MPS+12].

The Open Group plans to develop a catalogue of components under the "Mils(TM)" (this spelling) trademark that are backed by an Open Group Mils protection profile. [RD07, Del10] list protection profiles for MILS components such as console system, a network system and a file system, and suggest to specify the allocation of trust of specific MILS components to a MILS Integration Protection Profile (MIPP); however, these PPs are in draft form and are not public.

[SKPP] was a protection profile for separation kernels running on hardware. Successful certification was achieved for the Green Hills Integrity system running on PowerPC 750CXe PCI extension card [Gre08]. However, [SKPP] has been retracted ("sunsetted") by NSA in September 2011. The published rationale for the sunsetting includes the considerations (1) that the NSA "*will focus on specific government systems using separation kernels rather than general OS evaluation*" [Wis11], and points [Hou11] to that (2) in the project "one box one wire" (OB1) "*the underlying commodity workstation (as part of a separation platform) does not appear to be appropriate for SKPP certification due to its complexity*" and that "*the problem with commodity desktop platforms comes down to the fact that too many developers and vendors are interdependent*" [SNAC10]. In balance, in the same document, it is pointed out that "*commodity workstations may present a completely acceptable risk profile given available options*" and the "*findings in this document do not condemn OB1 or the use of separation kernels in commodity workstations*", [SNAC10, also discussed in NG12]. Note: concerning (1), this policy change does not apply to Europe, concerning (2), our certification approach for the separation kernel component does not include the hardware. That is, we assume that either the hardware has been certified by the CC, or it is trusted to be reliable for other reasons, e.g. by evidence from the hardware vendor that the hardware is suitable for the security-critical purpose intended.

For partitioning communications systems (PCS), a protection profile draft exists [Uch05] (available on demand from the author) which extends the PIFP (partitioned information flow policy) from [SKPP] to distributed environments. The High Assurance Security Kernel protection profile [HASK] also addresses distributed communication systems in the style of a PCS.

2.4 Architectural decomposition and modelling

Since a long time research on security software architecture has emphasized principles that also can be found in MILS systems. For example, discussing mechanisms and techniques that define who may use or modify the information stored in a computer, Saltzer and Schroeder have pointed out that the design shall be kept "as simple and small as possible" [SS75, p. 1282], that "every access to every object" shall be checked and that the design shall be open (not secret). As they are widely known, we will revisit the [SS75] design principles and the extent to which they are fulfilled later (in Section 5.2).

In the context of *general* research on software architecture, the MILS approach with its strong emphasis on how a system is composed would subsumed under a structural model which is characterized by components, connectors and additional constraints [BCK03, SG95, ZAF06, ZAF08]. A MILS channel is a "connector" and the additional constraint on the system ("other stuff" in [SG95]) is non-interference. For component-connector type systems, [CBB+03, Section 4.7] proposes documentation in the form of either Architecture Description Languages or UML. If UML is used, [CBB+03, Section 4.7] discusses how to represent components and connectors in UML and note that connectors can be either expressed as dependencies between a component, and the ports/interfaces realized by the component or as components themselves (p. 162). [ZAF06, ZAF08] discuss decomposition patterns for components such as "product pattern", "cascade pattern", "feedback pattern" and several instances of "aggregation patterns".

The secure refinement of a downgrader with regards to information flow properties is demonstrated by a paper-and-pencil argument in [CVdM09].

MILS architectures have been expressed in Architecture Analysis and Design Language (AADL), verified by the REAL tool [GH08], and then been used for code generation by [DPK10]. MILS components have been expressed in the LOTOS language by [Alv98]. In [BBH+05], boundary flow modelling and secure UML are listed as possible support to the system integrator. The software engineering tool Specware for the breakup of a system has been used by [MWTG00]. [Cof11] discusses identification of architecture design patterns on an IMA system.

Chapter 3 MILS representation adopted by EURO-MILS

This chapter presents the EURO-MILS project view of a MILS architecture template using a top-down approach (Section 3.1), followed by a bottom-up approach giving definitions of terms considered useful to describe the MILS architecture template (Section 3.2). We conclude this chapter with an example (Section 3.3).

3.1 MILS architecture template

Figure 1 presents a high-level view of a MILS architecture template. This is the template we adopt in the EURO-MILS project. The term "*MILS architecture template*" names a template encompassing many possible MILS systems, whereas the term "*MILS architecture*" (without "*template*") refers to the architecture of the implementation of a concrete MILS system.

From the outside (i.e., external world, which could be a larger system comprising the MILS system), the MILS system is seen as a system that handles information from multiple components with different security and safety levels concurrently, in other words, an MLS system. The MILS system's internal architecture is not visible from the point of view of the infrastructure around the MILS system (it is like a black box). Thus, a MILS system can be used as a base to build a system that has different safety/security requirements for different components, called an MLS system.

In the rest of the section, we are discussing in more details each part shown in Figure 1.

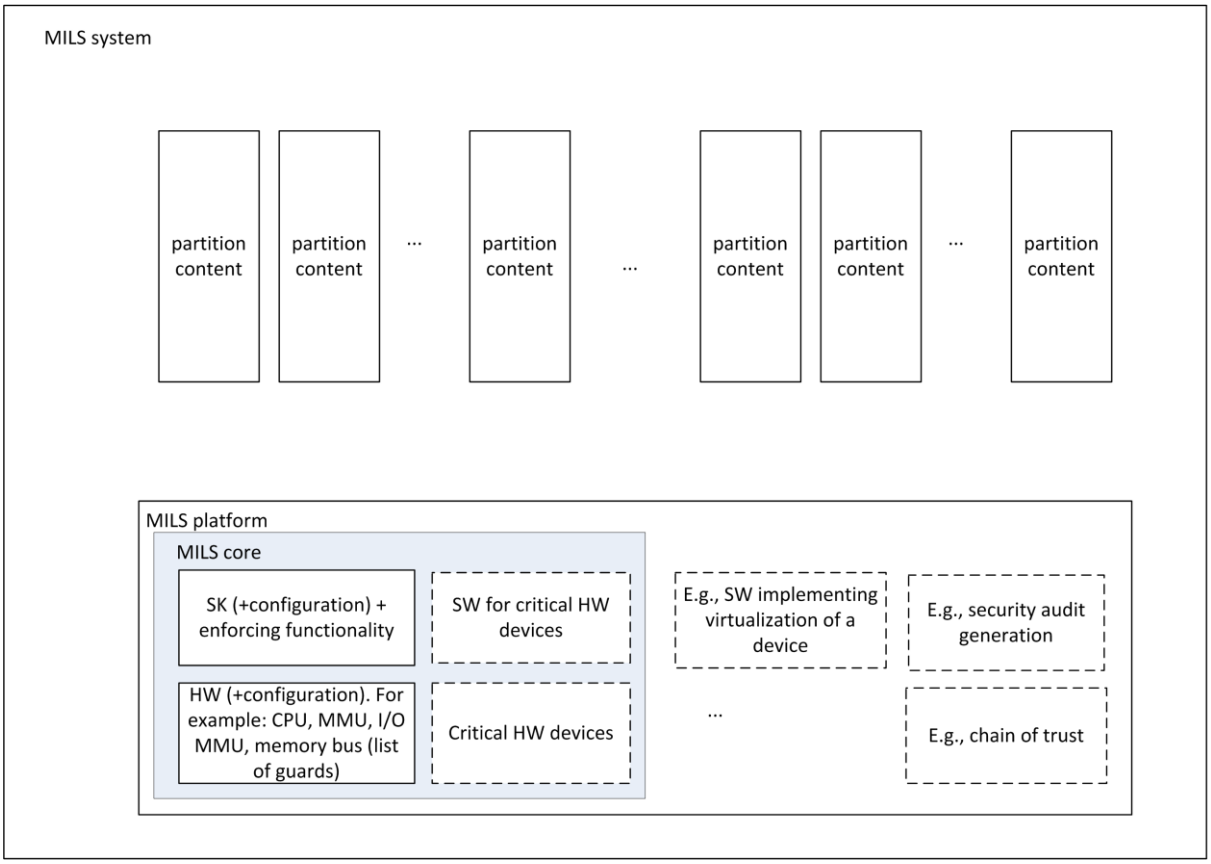

Figure 1: MILS architecture template (components in dashed lines are optional).

3.1.1 MILS system

We define a MILS system as a system where its MILS architecture is visible to the person composing the MILS system from its components, i.e., the *system integrator*.

A MILS system consists of components interacting with each other. We define three main components in a MILS system:

- MILS core (Section 3.1.2)
- MILS platform (Section 3.1.3)
- Partition (Section 3.1.4)

3.1.2 MILS core

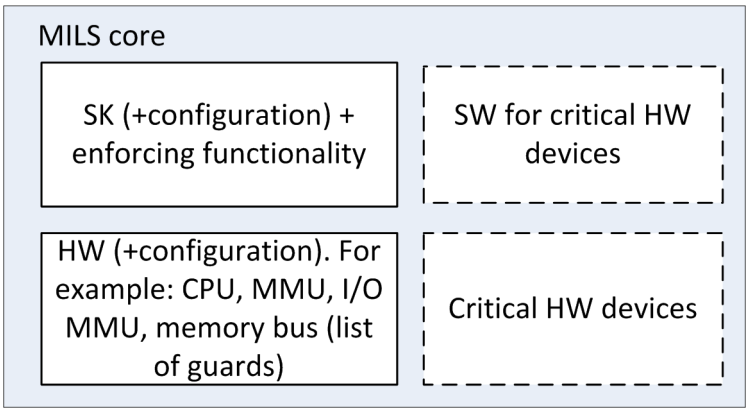

Figure 2: MILS architecture template: MILS core

The only goal of the MILS core is to provide separated partitions with controlled information flow between them. Thus, the MILS core provides the primary security functionality of a MILS system. The MILS core (Figure 2) consists of components that implement and enforce the separation both in space and time*: separation-supporting hardware* and the *separation kernel*. Depending on use-case implementation, the MILS core may also include hardware *critical devices* and *software for these hardware devices*.

- Separation-supporting hardware.

  This hardware consists of implementation (gates in silicon) and configuration/initialization.

  The hardware shall support separation, e.g. CPU with different privilege modes, MMU, memory bus, IOMMU. Hardware consists of interconnected components. A hardware component's interactions with other hardware components can be restrained by a guard. For example

  - Let's consider a CPU, memory, and MMU. Assume the CPU is working in a user mode. In this mode, the CPU can only access memory if the access has been permitted by the MMU. Thus, the MMU is the guard for the CPU.
  - Let's consider a device, memory, and IOMMU. Assume the device accesses memory as a DMA. In this case, DMA access will happen only if the IOMMU permits it. Thus, the IOMMU is the guard for this device.

  Configuration/initialization is software that performs hardware-specific initialization and configuration of hardware, e.g. firmware and/or bootloader and/or stand-alone software.

- Separation kernel.

  The separation kernel guarantees separation and controlled information flow by enforcing the security policy.

  Examples of enforced security polices are

  - resource allocation policy (e.g. allocation of CPU time and memory to partitions),
  - access control policy (e.g. access rights to objects under control of separation kernels),
  - information flow policy (e.g. communication rights of partitions).

Separation kernel functionality relies on the hardware supporting functionality.

A separation kernel may further configure hardware with the respect to a given security policy. For example, it configures guards, creates page tables and sets MMUs.

- Critical hardware parts/devices.

  These devices can bypass the enforcement mechanisms of the separation kernel. For example, DMA capable devices without guards (i.e. without IOMMU) can bypass the separation kernel. To have such critical devices is optional. However, if such device is present, its associated software acting as a guard for it must be also present in the architecture.

- Software for critical hardware parts/devices.

  This software is the guard for a critical device. It provides an API to partitions to work with the device. Therefore, this software implements and enforces part of the separation. We assume that if the software correctly works with the device, the device will not bypass the separation kernel security policies.

3.1.3 MILS platform

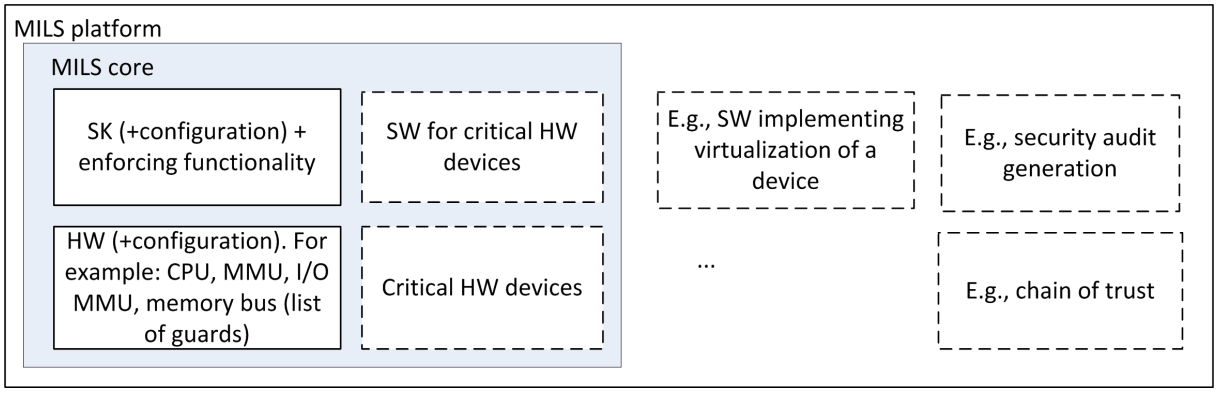

Figure 3: MILS architecture template: MILS platform

The MILS platform (Figure 3) consists of the MILS core and optional software and/or hardware components that provide secondary security functionalities and do not contribute to the enforcing of separation. These are security services that can be used based on the use-case needs.

These optional components are part of the platform because they

- contribute to the system security, however, they do not enforce separation between partitions,
- can be used by several partitions,
- may be realized by different implementations for a given optional security service (use-case dependent),
- might need a tighter integration with separation kernel or hardware.

Examples of such optional components are:

  - security audit (Section 3.1.5),
  - crypto functionality shared between partitions,

- software implementing virtualization of devices (e.g., multiplexing of accesses for the network interface, shared graphics or shared audio).

3.1.4 Partition

A partition is a unit of the separation created by the MILS core. A partition will get resources as specified in a security policy and enforced by the MILS core. A partition is a container that hosts executable and/or non-executable data. An executable in the partition can use allocated resources, communicate with the MILS core, and communicate with other partitions under control of the MILS core if such communication is explicitly allowed by the security policy.

Partitions also may include hardware that is not separation-relevant. For example, an FPGA doing cryptography can be under full control of a single partition.

3.1.5 Security audit

Security audit, if it exists, is part of the MILS platform. Security audit is the trustworthy gathering of audit records. The audit records can be generated by the MILS core components or applications hosted by partitions.

A security audit component processes incoming data by adding trustworthy security related information such as time stamps and source of audit record. It can be local and managed by the audit component but also exported to an external media, this aspect being use-case implementation dependent.

3.1.6 Middleware

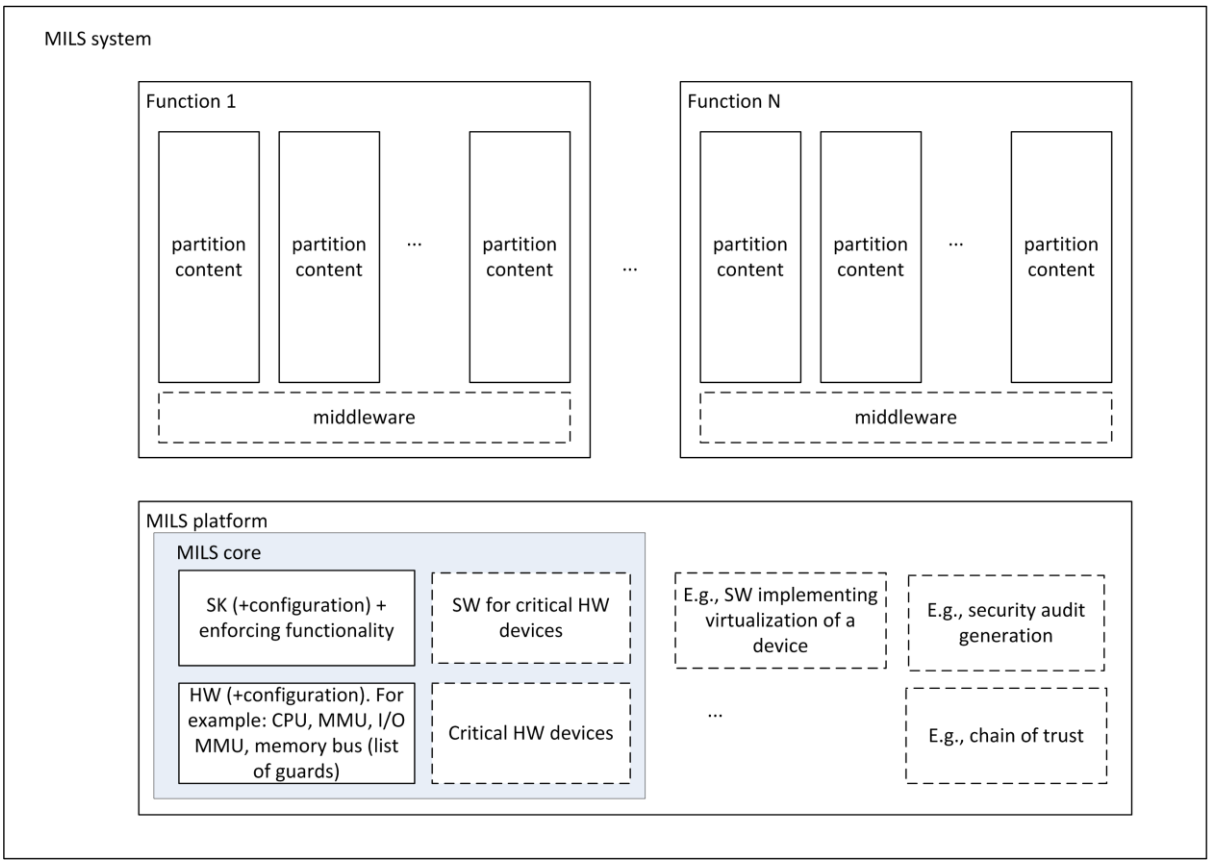

Figure 4: MILS architecture template with middleware: components in dashed lines are optional.

The term middleware is generally not well defined and its meaning always depends on the context.

We define middleware as a set of services that are used by several partitions. Middleware does not contribute to the separation enforced by the MILS platform and is itself under control of the MILS platform, thus it is a unit (a partition) under control of the separation kernel. The system integrator of a MILS platform for a MILS system can decide to have middleware or not. Middleware can be a partition providing some functionality for several other partitions or be a part of a partition (e.g. libraries, run-time environments guest operating systems). For instance, the habitat of middleware is also restricted to be *within* a partition in [Win13, p. 3, Figure 2].

By introducing the concept of the middleware, we acknowledge that it can be useful to express that some partitions can be part of a bigger function (see Figure 4), and thus, need common infrastructure, which is not related to the MILS core or the MILS platform. In the generic MILS architecture template, we agreed to not use the term middleware to avoid any misunderstanding because it depends on the use-case where a MILS system is employed.

3.2 MILS terminology

3.2.1 Component

A *component* is a term to describe the decomposition of a (in general, *any*) system into meaningful self-contained parts. For example, a (yet to be defined) MILS system consists of components. In general, components may be implemented by (1) hardware, (2) software, or (3) a combination of hardware and software [CBB+03, DO-297]. A component provides a given functionality that can be configured according to a given use-case.

3.2.2 Resource

A *resource* is anything (processor such as a CPU or a processing core, memory, software, data, network, etc.) internally used or exported by a component. A resource may be physical (a hardware device) or logical (a piece of information). A resource may be shared by multiple components or be dedicated to a specific component.

Exported resources are those resources to which an explicit reference is possible via a component interface, e.g., the programming or configuration interface. *Internal resources* are those resources used exclusively by the component, and which have no explicit reference via a component interface.

For example, internal resources of an operating system usually comprise physical memory space, I/O memory space, the set of processors the applications can run on, allocated processor time for each processor (at least, when the operating system is a real-time operating system), and interrupts. A resource commonly exported by an operating system is a “file”. The operating system enforces an access control policy on the file. Internally, it uses memory to export the file. Another exported resource exported by an operating system is time slices, and the operating system enforces a scheduling policy (a resource management policy). Internally, the operating system uses CPU time that itself has access to.

3.2.3 Communication object

A *communication object* is an exported resource provided by a component. It can be shared between components. Communication objects are used by components to communicate between them.

3.2.4 Security policy

A *security policy* is a set of rules to be enforced by a component. Examples of security policies are:

- Resource allocation policy (Section 3.2.5)
- Access control policy (Section 3.2.6)
- Information flow policy (Section 3.2.7)

In our context, all three policies describe rules for granting or denying some "treatment" of exported resources, with "treatment" of a resource standing for to be able to eventually "read", "write", or "execute" the exported resource. The distinction between the resource allocation policy and the access control policy is which interface the rules for access are applied on.

The term information flow policy has more than one usage, the most simple one is to use it as an umbrella term for "access control policy" and "resource allocation policy" combined. For most components, in the scope of this document, we adopt this simple interpretation, making these three security policies closely related. We decided not to merge the three policies in order to allow a precise characterization of components where needed later. Moreover, a more "complicated" usage of the term "information flow policy" will be encountered when the separation kernel is described (Section 3.2.13).

An operation might be governed by several policies: we consider both operations of "opening a file" and "reading/writing" to be involving access control to the file, however, the operation, depending on the implementation, could also be governed by a "resource allocation policy" such as the exclusive ownership of memory for the file descriptor to the component opening it. Similarly, the virtualization of a network component could comprise "access control" to Single Root I/O virtual functions and "resource allocation" if some of the virtual functions, after proper reinitialization, are assigned to different components during different periods of a time cycle (say 20 milliseconds each 100 milliseconds).

Note: We have observed that making the distinction between resource allocation policy and access control policy mixes concerns of interface (functional requirements) into policy requirements. However, the interface available to an attacker defines the possible malicious operations of the attacker. Thus, making the distinction allows to differentiate attacks based on resource depletion (attacking the resource allocation policy defined below) and attacks against confidentiality/integrity of the resources (attacks against the access control policy defined below).

A security policy can be dependent on system state, yet be bounded. For example, writing to a file may depend on that, statically, access to the file is allowed, and that, dynamically, a file descriptor is available after "opening" the file. Sometimes, in a usage that, after discussion, we do *not* follow in this document, the term "resource allocation" is used for initial establishment of a dynamic state, e.g. "opening a file" would be considered "resource allocation", whereas "reading/writing" the file would be governed by access control. For this

document, as outlined above, we consider both operations of "opening a file" and "reading/writing" to be involving access control to the file.

When describing the protection of assets in the system, one can assume that every action that is eventually possible by configuration will be used by an attacker, even if some initialization of the state is needed. Thus, the static configuration describes a bound for the behaviour that is dynamically possible. Section 4.3 further discusses system configuration.

3.2.5 Resource allocation policy

A component's *resource allocation policy* acts on the component's interface used to manage exported resources. This interface is characterized by that a request for a resource is made without knowing in advance how the resource is "named" or "addressed". The request is made for a quantity of the resource, and then the component decides whether to grant or deny the request to export that resource in the desired quantity. The resource allocation policy defines which of the component's resources are kept internal to the component and which are exported to which other components. When a resource is exported to more than one other component, the resource is *shared*. A resource allocation policy can be in the "space" domain, when resources can be used simultaneously but are kept in different spatial (e.g. memory) locations or in the "time" domain, where resources are used sequentially, but kept in different time slices. An example for resource allocation in the "time" domain is the allocation of a CPU to a component for a limited period of time.

3.2.6 Access control policy

A component's *access control policy* acts on the component's interface used to manage exported resources. In this respect it is identical to the aforementioned resource allocation policy (Section 3.2.5). However, the interface is characterized by that a request to the resource includes an explicit reference to the resource (e.g. the resource's name or a numerical identifier). Identically to the aforementioned resource allocation policy (Section 3.2.5), the access control policy defines which of the component's resources are kept internal to the component and which are exported to which other components. When a resource is exported to more than one other component, the resource is *shared*. The access control policy is in the "space" domain.

Note: as observed in Section 3.2.4, the resource allocation policy (Section 3.2.5) and the access control policy (this section) differ in the interface offered on the exported resources and they differ in the threats (exhaustion versus violation of integrity/confidentiality). For resource sharing, the threats a shared resource is exposed to are different: a resource shared under a resource allocation policy, e.g. a memory allocator that can be used by different components, can be exhausted ("denial of service"), but a resource shared by an access control policy, e.g. a piece of memory at a fixed address that is marked as accessible to several components, cannot.

3.2.7 Information flow policy

The term *information flow policy* has more than one usage,

(1) the most simple one is to use it as an umbrella term for "access control policy" and "resource allocation policy" combined or

(2) to express policies where pieces of information (messages) are written to one or several communication objects(s) by a *sender* and subsequently these messages are

read from the communication object(s) by a *receiver*. Such policies may include rules based

(2a) on the sender/receiver of the messages and/or

(2b) on the *content* of these messages.

Note: for most components, interpretation (1) is used. (2a) will be used in the context of a separation kernel (Section 3.2.13). The enforcement of (2b) is a typical task of security gateway (discussed as an example in Section 3.3). An information flow policy in the sense of (2a) is either explicit, based on identities of components between which information flow is allowed, or implicit, as unambiguously defined by the resource allocation policy and access control policy.

3.2.8 Configuration

The *configuration* of a component contains the component's identity, and it defines any security policy (access control policy, resource allocation policy, information flow policy) enforced by the component. An information flow policy configuration also may be implicitly configured by resource allocation policy configuration and access control policy configuration.

3.2.9 Application

An *application* is one or more executable(s).

3.2.10 Domain

A *domain* (or "security domain") is a unit of separation created and maintained by any MILS component, for example by an application (Section 3.2.9), a function (Section 3.2.12), or the MILS core (Section 3.2.14), which is enforcing a security policy on exported resources.

In particular, a domain is a "space" domain, if exported resources can be used simultaneously but are kept in different spatial (e.g. memory) locations. A domain is a "time" domain, if exported resources are used sequentially, but kept in different time slices.

3.2.11 Partition

A *partition* is a component that serves to encapsulate application(s) and/or data. Thus, the content of a partition is application(s) and possibly other data. A partition is a unit of separation with respect to

- resource allocation in the space and time domains,
- an access control policy and an information flow policy in the space domain.

In a MILS system, partitions are created and maintained by the MILS core (see definitions below) based on security policies defined for a given use-case.

Note: this bottom-up definition of a partition has a different emphasis than the previous top-down characterization given in Section 3.1.4, but does not contradict it.

A partition is a domain, but a domain is not necessarily a partition.

3.2.12 Function

A *function* is a logical group of partitions for achieving common objectives. The implied partitions may be connected using information flows.

3.2.13 Separation kernel

A *separation kernel* is a component that enforces a resource allocation policy and an access control policy on its exported resources (partition, resources allocated to a partition, communication objects). Communication objects allow for controlled information flow between partitions. A separation kernel may have an explicit or an implicit information flow policy on its partitions (see definition of information flow policy for details).

The separation kernel uses separation-supporting hardware to provide the separation between partitions in a MILS core.

Examples:

- A resource allocation policy might assign a certain amount of time, for example 20 milliseconds periodically every 100 milliseconds, of the resource CPU access to a certain partition, for example partition number 5.
- An access control policy might assign communication object *C* as writable to partition *A* and readable to partition *B*, defining an implicit information flow policy from *A* to *B*.
- An explicit information flow policy for a separation kernel could consist of the specification that only partition *P* via whatever interface may send information to partition *Q*.

3.2.14 MILS core

By MILS core we refer to the minimal set of components needed for separation of partitions on a MILS platform. The only goal of *the MILS* core is to provide separated partitions with controlled information flow between them. Thus, the MILS core provides the primary security functionality of a MILS system. The MILS core (Figure 2) consists of components that implement and enforce the separation both in space and time.

3.2.15 MILS platform

A *MILS platform* consists of the MILS core and optional software and/or hardware components that provide secondary security functionalities and do not contribute to the enforcing of separation.

3.2.16 MILS system

A *MILS system* is a concrete deployment of a MILS platform with a defined set of partitions.

3.2.17 MLS system

An *MLS system* is a system with different security requirements for different components. It can be implemented by a MILS system.

3.2.18 Terminology rationale

The term *component* is a standard term for the description of software architectures (see also Section 2.4). On what can be a *component* we note that some presentations of MILS systems such as [UV05] come with a fixed number of layers. Others argue that, in principle, components themselves can contain MILS systems, allowing recursive compositions [Del12a, p. 56].

Resource: In software interface documentation, when specifying a component, we can describe what resources the component provides and what resources the component uses [CBB+03, p. 229]. In [SKPP, p. 21] resources that are provided by the component are called "external resources" whereas resources that are required by the component are called "internal resources". From a resource usage perspective, resources can either be hardware or resources provided by other components as in [Tan07, p. 432] where "*a resource can be a hardware device (e.g. tape drive) or a piece of information (e.g. a locked record in a database)*" or "*Any element of a data processing system needed to perform required operations; for example: storage devices, input/output units, one or more processing units, data files, and programs.*" [ANS01]. The use of "resource" for describing hardware is also established in virtualization [PG74]. We have not found a stand-alone definition of the term "resource" in the MILS literature, but for separation kernels the hardware notion it appears close to [AFHOT06, p. 3] where the term is not explicitly defined. In the context of a description of a separation kernel, the term "shared resources" is expanded to "*microprocessors, system registers etc*." whereas the "piece of information" aspect appears to be addressed in [Rus08a, p. 10].

In [Rus08a], our *resource allocation policy*, *access control policy*, and *information flow policy* are equated to a "*resource sharing"* + (information flow) "*policy"*. Also [SKPP] does not have any notion of an access control policy. We prefer to keep the three terms, because it simplifies mapping to [CC12], where the resource allocation policy can be mapped to the functional requirement class FRU_RSA, the access control policy can be mapped to FDP_ACF, and the information flow policy can be mapped to FDP_IFF. That resource sharing implies information flows and that conversely resource sharing analysis supports information flow analysis is widely accepted [Kem83, AFOB+12]. Resource allocation policies versus access control list-based policies, e.g. the need to maintain resource exhaustion quantifiers to enforce resource allocation quotas, are discussed in [Ste91, p. 228].

Our definition of *application* is based on [ANS01]. It avoids any notion of user, as mentioning the term "user" at an early stage of the introduction could create the misunderstanding that users are limited to human beings using the system interactively.

The use of the term *domain* for environments where a security policy is imposed by a component can be found, for example, in [Lam71]. The same paper also shows (p. 428) examples for hardware-imposed domains (supervisor and software states) and software (user environments in an operating system).

Our definition of *partition* is close to [AFHOT06, p. 2] where a partition is defined as "a collection of data objects, code and system resources". [SKPP, p. 20] points out that the term is motivated from its use in mathematics, where a partition of a set *A* is used to describe the split of a set into disjoint subsets, so that each element of *A* belongs to exactly one of the subsets.

Our definition of *function* (logical group of partitions for achieving common objectives) is what in [DO-297] is called an application.

MILS platform + partitions content = MILS system: this is emulated after IMA, where an IMA platform + partitions give an IMA system.

MLS system: We identify a MLS system with a system having different security requirements for different components. In safety, the term "mixed criticality" is often used for this. As discussed in Section 2.2, historically, there exists also a more restrictive usage, where a MLS system has a transitive security policy [BDR+08]. In line with many others (e.g. [DCS+04, LRP+11]), we do *not* adopt that more strict definition.

3.3 Example of a MILS system

In this section the terminology of Section 3.2 is applied to a concrete MILS system described in [MPT+12]. The paper explains a gateway architecture implemented using the MILS principles for the purpose of controlling the content of the information flow between the hosted applications. Those applications process data of different security classification logically grouped into a green domain and a brown domain. As foundation, the gateway uses a separation kernel, which provides the functionality of partitioning and controlled non-bypassable information flow. Thus, the separation kernel applies a Resource Allocation Policy and maintains an Access Control Policy and a basic Information Flow Policy, defining the partitions that are allowed to communicate among each other. However, this Information Flow Policy of the separation kernel is not able to ensure additional constraints on content of the data that is transferred using the communication objects. The gateway enhances this Information Flow Policy by this capability using the available foundations.

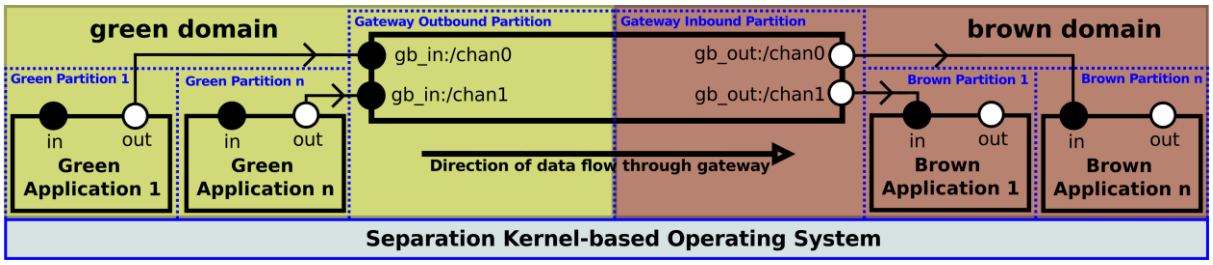

Figure 5: Gateway architecture of a MILS system [MPT+12, Figure 3].

- Applying a black-box view from the outside onto the system in Figure 5, the system appears as a *MLS system*, since it processes data belonging to the green domain and data belonging to the brown domain concurrently.
- Having a closer look into the architecture of the system, the system is a MILS system, since it uses a *MILS platform* (the Separation Kernel-based Operating System) and partitions identified by the blue dotted lines in Figure 5.
- The *MILS platform* comprises the Separation Kernel-based Operating System (the *MILS core*) plus the Auditing Module mentioned in [MPT+12, Section II.D].
- The *MILS core* is the Separation Kernel-based Operation System [MPT+12, Section II.D] plus some unspecified hardware (that is not further described in [MPT+12]) but used and managed by this separation kernel.
- As *Separation Kernel* the example uses PikeOS [MPT+12, Section II.D]. This separation kernel enforces the *Resource Allocation Policy* and *Access Control Policy* to form *partitions* (the blue dotted boxes) based on the available *resources*, such as memory, CPU cores and the program binaries. The separation kernel exports some of those resources for building communication objects. By controlling the accesses of the

partitions to those *exported resources* the separation kernel creates and applies a basic *Information Flow Policy*. Examples for exported resources are: ARINC 653 ports or file providers [MPT+12, Section IV.C].

- *Partitions* are provided by the separation kernel. The gateway relies on this crucial element for implementing its function.
- Within the partitions the example executes several *Applications*, which is the content of the six blue dotted boxes in Figure 5. The paper specifies applications running within the "Gateway Outbound Partition" and within the "Gateway Inbound Partition". Other applications are the Brown Applications and Green Applications.
- As *components* the modules of [MPT+12, Section IV.II], such as the modules with specific functionality for filtering packets (named "Viewer Module"), reading/writing filtered packets across partitions ("Border-crossing Module") or making decisions on packet routing ("Routing Module") can be identified. The paper does not explicitly identify hardware; however it is assumed that the system contains at least a CPU, which again is a (hardware) component.
- The purpose of the gateway is to enhance the basic Information Flow Policy of the Separation Kernel by the ability of controlling the content of the information flow (unidirectionally) [MPT+12, Section IV.I] and [MPT+12, Section V]. For achieving this logical *function*, the gateway uses the collaboration of two partitions: the Gateway Outbound Partition and the Gateway Inbound Partition. Other functions are given by the applications located inside the two security domains, which "can comprise one or more partitions" [MPT+12, Section IV]. Functions are depicted as a black solid boxes in Figure 5.
- The system contains *configurations* of different applications:
    - The configuration of the gateway for defining the enhanced Information Flow Policy.
    - The configuration of the separation kernel for defining the Resource Allocation Policy and the Access Control Policy for the gateway components [MPT+12, Section IV.II]. [MPT+12, Section IV.III] forces the system integration to provide "enough buffer space" for the exported communication objects. [MPT+12, Section IV.IV] discusses the scheduling configuration of the system. Non-bypassability of the gateway's enhanced Information Flow Policy is ensured by the separation kernel.

Chapter 4 MILS main components

The following characterization of components does *not* include all MILS components, but rather discusses the security properties of MILS components that are common to MILS platforms and occur frequently. We begin with software components (Section 4.1), followed by hardware components (Section 4.2) and discuss the configuration of MILS systems (Section 4.3).

4.1 Software components

4.1.1 Separation kernel

A concise characterization of a "separation kernel" already has been given in Section 3.2.13. In this section, we look at the "separation kernel" as a MILS software component.

4.1.1.1 Services

4.1.1.1.1 Pictorial view

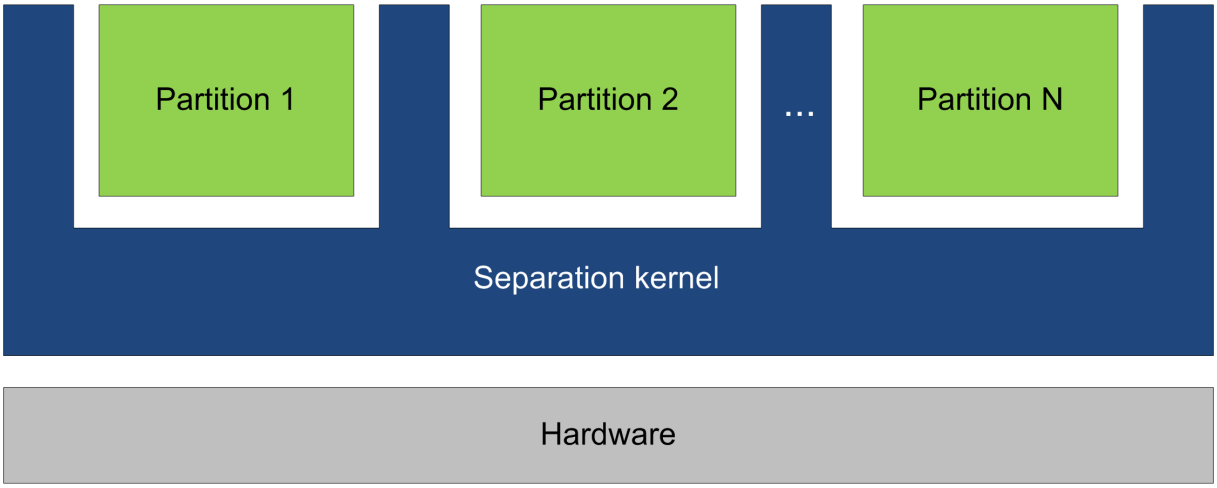

Figure 6: Generic picture of a separation kernel with several partitions.

The pictorial view is the most commonly found way to describe the services of a separation kernel. Figure 6 shows that each partition is under control of the separation kernel, in the sense that the separation kernel enforces the system configuration upon all their communication and resource requests in a non-bypassable way, while it is not inspecting or protecting what happens within the partition itself. For example, if a partition is authorized to communicate over a network and to use the HTTP protocol, the SK will not protect the application against infection by a virus introduced into the HTTP payload.

4.1.1.1.2 Classical approach

In some of the early work such as [BBH+05, UV05, AFHOT06] a strong emphasis on the implementation of information flow and its absence has been taken.

"*The only tasks assigned to a MILS separation kernel are the partitioning of processes and failure containment. Consequently, we can represent the safety and security requirements for a separation kernel by four simple foundational policies:*

• ***Data Isolation***: *Information in a partition is accessible only by code running in that partition. Private data remains private.*

• ***Control of Information Flow***: *Information flow among partitions is from an authenticated source to authenticated recipients. The source of information is authenticated to the recipient. Information goes only where intended.*

• ***Resource Sanitization***: *Usage of the microprocessor and other hardware, such as networking hardware, cannot be used as covert channels to leak information.*

• ***Fault Isolation***: *A failure in one partition is prohibited from cascading to any other partition. Failure detection, containment, and recovery are performed locally*" [UV05].

Similar formulations are found in an early draft of an SKPP predecessor (defining "data isolation", "control of information flow", "resource sanitization") [WOM02].

4.1.1.1.3 Policy-based description

For convenience, we repeat our definition from Section 3.2.13.

"A separation kernel is a component that enforces a resource allocation policy and an access control policy on its exported resources (partition, resources allocated to a partition, communication objects). Communication objects allow for controlled information flow between partitions. A separation kernel may have an explicit or an implicit information flow policy on its partitions (see definition of information flow policy for details).

The separation kernel uses separation-supporting hardware to provide the separation between partitions in a MILS core."

We think this description with an emphasis on policies fits better in a systematic exposition. A description based on policies has also been adopted in the "MILS constitution" [Rus08a], another attempt to systematically explain MILS.

This characterization is isomorphic to the characterization of Section 4.1.1.1.2: "resource sanitization" and "damage limitation policies" are implied by the requirement of complete information flow control. "Data isolation" is the default of the access control policy, resource allocation policy and information flow policy on internal resources, whereas "control of information flow" addresses the access control policy, resource allocation policy and information flow policy on external resources.

4.1.1.1.4 Description of functionality grouped according to where separation is made (space/time)

In the following paragraphs, we present the approach taken in [TBF13] then we comment it versus previous sections (i.e., classical approach and policy-based description).

Separation in space*: Applications can be hosted in different partitions. Partitions get assigned memory resources (i.e. space). In this way, the separation kernel enforces its configuration: that is, access control on partition content, per-partition provision of physical memory space and I/O memory space. By confining applications into partitions, the*

separation kernel enforces that these applications can affect neither applications in other partitions nor the separation kernel itself.

***Separation in time**: Applications can be hosted in different partitions. Partitions get assigned CPU time (i.e. time windows). In this way, the separation kernel enforces its configuration: that is the allocation of a predefined amount of the CPUs' time to partitions. Several partitions can share the same time window. On a partition switch CPUs will be reused. The separation kernel enforces that no residual information is in CPU registers or memory caches according to the configuration. The separation kernel assigns a priority to every subject to allow priority based scheduling within one time window.*

***Provision and management of communication objects:** Applications hosted in different partitions can get assigned a set of communication objects under control of the separation kernel. A communication object is an object exposed to one or multiple partitions with access rights as defined in the configuration data, thus allowing communication between partitions.*

***Separation kernel self-protection and accuracy of security functionality:** Separation kernel self-protection and accuracy of functionality supports reaching and keeping a safe and secure state of the MILS system. The separation kernel statically assigns automatic invocations of error handling functions to recover from or respond to error conditions.*

Again, this characterization is isomorphic to the characterization of Section 4.1.1.1.2 and Section 4.1.1.1.3. Like the one of Section 4.1.1.1.2, it is optimized to be stand-alone and concrete. It splits up the data isolation of Section 4.1.1.1.2 of into "separation in time" and "separation in space". The "resource sanitization" of Section 4.1.1.1.2 is subsumed under "separation in time". "Control of information flow" is represented by "provision and management of communication objects". "Fault isolation" is subsumed under "separation in space" and "self-protection".

Also in the SKPP, while claimed as security functional requirements, "fault containment" and "resource sanitization" are *not* listed explicitly in the introductory high-level characterization of "*core* functional requirements" [SKPP, p. 25].

4.1.1.1.5 Virtualization services on top of separation kernels

Virtualization is *not* a necessary part of separation kernels. However, because many separation kernel deployments provide support for virtualization services, the concept is described here. We discuss these concepts in form of tables (Table 2 and Table 3), juxtaposing a generic description of virtualization and the analogous or differing complement in a separation kernel.

A *virtual machine (VM)* consists of software that imitates a physical hardware machine. The virtual machine will for example give the illusion of a physical CPU and physical memory to an operating system that is running in it. An operating system running in a virtualization environment is called *"guest"*. In the MILS context, a virtualized operating system is a special case of an application (the term "application" was defined in Section 3.2.9).

A *virtual machine monitor (VMM)*, also called a "*host*" (for type 2 VMMs) or "*hypervisor*" (for type 1 VMMs, see Table 2 for type 1 and 2 explanation), is the software managing virtual machines.

Requirements:

Table 2 lists virtualization requirements in general and their fulfilment or non-fulfilment by a MILS separation kernel.

Virtualization Requirement in General	**Virtualization Requirement Compliance in a MILS Separation Kernel**
An operating system running on a VMM is characterized by: (1) the resource control property, that the VMM is in complete control of system resources, [PG74]	Concerning (1), the resource control property: In MILS systems, the resource control property is implemented by the separation kernel via its security policies.
An operating system running on a VMM is characterized by: (2) the sufficiency property, that a VMM provides an environment for the operating system which is sufficient for running it.	Concerning (2), the sufficiency property: The sufficiency property means that the API provided by a MILS system to its applications does not have to provide the same API as in virtualization of a machine, e.g. for a MILS system it is sufficient to provide communication channels instead of, for example, a network interface, but it need not necessarily provide a full replica of another machine. In a VMM, the API is the full CPU instruction set. When under a VMM, a VM attempts to execute an instruction that only runs in supervisor mode of CPU, VMM intercepts this attempt and VMM tries to emulate the instruction as faithfully as possible. In a separation kernel, when an application in a partition executes an instruction that only runs in supervisor mode, the SK traps it, and usually its execution is rejected. Instead, the separation kernel offers explicit additional interfaces to allow partitions to do certain things (e.g. create new thread within a partition, use a new address space within a partition, access a shared resource etc.). The main difference would be that VMM tries to create virtual environment but SK does not.
An operating system running on a VMM is characterized by: (3) the isolation property, that is applications running in different VMs do not interfere with each other	Concerning (3), the isolation property: this is provided by the fact that the separation kernel enforces temporal and spatial separation properties on applications.
An operating systems running on a VMM is	Concerning (4), the efficiency property:

Virtualization Requirement in General	**Virtualization Requirement Compliance in a MILS Separation Kernel**
characterized by: (4) the efficiency property that programs run on VMM with only minor decreases in speed [PG74]	While, in practice, the efficiency property is probably fulfilled by most MILS systems, the emphasis is less on good average application performance but rather on guaranteed real-time worst case execution time bounds.
(5) While virtualization has traditionally been focusing on the isolation of virtual machines hosted by the same hardware platform, controlled resource sharing, such as for example a common storage, can also be a desired feature [Kar05].	Concerning (5) controlled resource sharing: it is well supported by communication objects.

Table 2: Virtualization requirements: in general and their compliance with MILS SK

(Note: instead of the sufficiency property and isolation property [PG74] gives the stronger equivalence property, that a VMM provides an environment for programs which is essentially identical with the original machine, except for timing effects. Our definition is broader to allow for paravirtualization, see below.)

Implementation:

Table 3 lists virtualization implementation characteristics in general and their applicability or non-applicability in a MILS separation kernel.

Virtualization Implementation Characteristics (of an Operating System) in General	**Virtualization Implementation Characteristics (of a MILS application) in a MILS Separation Kernel**
(1) Since [Gol73], it is customary to distinguish between Type 1 VMMs that run on bare-metal hardware (e.g. Microsoft's Hyper-V, IBM's System z Processor Resource/System Manager (PR/SM), bare-metal version of VMWare) and Type 2 VMMs that run on top of another operating system (e.g. VirtualBox, user-space version of VMWare). An extensive list of VMMs and their classification can be found at [Wik13].	Concerning (1), the VMM type: MILS platforms are always of Type 1. Contrary to virtualization techniques where safety/security requirements do not matter, in MILS systems, there is an additional emphasis on deployability in domains with safety/security requirements, e.g. that a MILS system, is "NEAT", which is not necessary for VMMs in general. For example, if safety/security requirements are not a primary concern, VMMs are not only provided by stand-alone systems but also running on COTS operating systems (e.g. a VirtualBox running a Windows on a Linux or vice-versa).
(2) A virtual machine can be run as an emulator, intercepting *all* instructions from	Concerning (2), running a virtual machine as emulation: while the exception, this can be

Virtualization Implementation Characteristics (of an Operating System) in General	**Virtualization Implementation Characteristics (of a MILS application) in a MILS Separation Kernel**
the operating system running on it, this comes at a high performance price [PG74].	done by a separation kernel, e.g. to run a legacy system designed for slower hardware, so that the performance cost is acceptable.
(3) Alternative to (2), a virtual machine can be run in a way that it runs an operating system directly on a CPU and the VMM only intercepts the operating system when needed, that is when invoked either by a trap coming from the application or from elsewhere (e.g. a system timer interrupt).	Concerning (3), running a virtual machine directly on hardware: also MILS applications can be run by a separation kernel directly on a CPU, and the separation kernel intercepts the MILS application only when certain traps arrive (e.g. a system timer interrupt).
(4) Alternative to (2) and (3), hardware virtualization support (also known as full-virtualisation) introduced by AMD and Intel in the mid-2000s ensures that all instructions that need to be intercepted can be trapped and it increases efficiency, by providing support for per-VM page tables.	Concerning (4), hardware support: a separation kernel can make good use of hardware support for virtualization when the running application is an operating system, simplifying page table management.
(5) Paravirtualization is a technique allowing to adapt the VM operating system and, if needed, the applications running on such VM to avoid instructions that are either inefficient or, on some architectures cannot be trapped (see "Note on imperfect virtualization support on hardware" below). Recall that, as applications can comprise virtualized operating systems, invocations of instructions to be run in supervisor mode is frequent. Paravirtualization replaces these supervisor mode instructions. Paravirtualization allows applications to run more efficiently or allows running applications that otherwise would not be running at all.	Concerning (5), paravirtualization: the technique of paravirtualization also can be applied to applications running within a partition of a separation kernel, e.g. a paravirtualized Linux operating system, that, in the MILS context, is just an application. The paravirtualization technique may enable applications otherwise not runnable on the separation kernel to run on the separation kernel, or make them more performant. From a security point of view, paravirtualization does not add any value to the security properties of a MILS system but it introduces a threat vector of attacks, which needs to be taken into account when a MILS system is configured.

Table 3: Virtualization implementation: in general and compliance with MILS SK

Note on imperfect virtualization support on hardware: Most modern CPUs enable to restrict the privileges of untrusted applications ("supervisor" versus "user" mode). This feature to restrict user applications to "user" mode is fundamental to general-purpose operating system design [Tan07, p. 1]. Integrity *is* a design goal of general-purpose operating systems and their CPUs, but the complete control of information flow channels *is not necessarily* a design goal neither for general-purpose operating systems nor CPUs they run on.

For example, [AA06, AFOB+12 (p. 153), RI00] and others have noted that, on some ia32/ia64 architectures, such as the Pentium, some instructions expose privileged state (such as reading out the global descriptor tables). Information flow can be mitigated if data, e.g. in global descriptor tables, is kept static. A second type of problem occurs when user applications are simply denied operations, but the CPU does not trigger any trap for the VMM to handle [AFOB+12, p. 149] also discusses similar caveats for another processor, the Cell Broadband Engine Architecture (CBEA) processor developed by Sony, Toshiba, and IBM that consists of a POWER architecture core and coprocessors elements for e.g. 3D multimedia acceleration.

4.1.1.2 Architecture

A separation kernel uses the interfaces of the hardware components it has been assigned in the MILS system in order to provide the services described in Section 4.1.1.1, enforcing its security policies according to configuration.

4.1.1.3 Assumptions on the environment

Hardware components are used by the separation kernel function as specified and provide policy enforcement as specified.

4.1.2 Generic device abstraction component

A *generic device abstraction component* is a MILS component having the purpose of abstracting the access mechanism of a special purpose hardware device to a defined set of connected partitions. In the simplest realization, this component mediates accesses from one partition to one hardware device only. The connected partition uses as interface to the component a standardized interface. More difficult realizations of this component allow connecting more than one partition to the component. This form requires a software-based virtualization strategy of the hardware component's functionality, which is supposed to be shared and impossible to be virtualized in hardware (e.g. by SR-IOV devices). In other words, all functionality that is not virtualizable by hardware shall be virtualized by software to provide the sharing functionality. As an example, communication based on an ethernet protocol optimized for avionics reliability requirements, Avionics Full Duplex Switched Ethernet (AFDX) requires sometimes to spread payload to multiple partitions. This is a functionality usually not supported by common (self-virtualizing) network hardware, since those devices can route data to one partition, only. Thus, the multiplication and distribution of payload needs to be done in software.

4.1.2.1 Services

Functionally correct implementation of the abstraction mechanism to the hardware devices.

Functionally correct implementation of the separation mechanism (resource allocation policy and/or access control policy) if more than one device is using this instance of the component.

4.1.2.2 Architecture

Other partitions interact with this component using the abstraction mechanism, it is the service provided by the component. For example, you have the POSIX standard interface (e.g. "read", "write") on the one the side and real hardware register accesses on the other side. By this, the Generic Device Abstraction Component abstracts the accesses. This component interacts with other component, i.e. hardware devices via their interfaces.

4.1.2.3 Assumptions on the environment

A separation kernel is available. The hardware device's interface to the component managing and abstracting it is not accessed directly by another component.

4.1.3 Console system component

Historically a console is a workstation at which a human operator can control a computer and interact with one program in a text-oriented (line or page) or graphical fashion. When interaction was simple and diagnostics means were primitive, a program would issue messages to the console, and the operator would grab the attention of the program from time to time. At the point in time the operator inputs commands, the program will usually answer by resuming its flow of messages. Progress in computing made it desirable to be able to address multiple programs at once, giving rise to a separation of the concept of a message console and that of the console or terminal used by an operator, and to the concept of multiplexing multiple virtual consoles over one physical one (or even within multiple layers of virtual consoles, in a tunnelling fashion).

The message console concept will be addressed by the audit system component (see Section 4.1.6). Here we focus on the console as a channel for interaction between an operator and programs. Note that on systems where users in the computer sense are not tied to human beings, a console is often absent, or hidden and used mainly for diagnostics and maintenance.

Therefore, a *console system component* connects applications to human interface devices, and thus is an instance of the Generic Device Abstraction Component.

If a console presents one program at a time, or several programs that belong to one security domain, then there will be no ambiguity for the human operator regarding the security classification. It is up to the human operator to ensure that he is controlling the right partition. If a console presents an operator with multiple security domains at the same time, then there has to be a non-bypassable mechanism such that the operator can always tell which domain he/she is interacting with.

It typically has one of the following forms:

- Physical, including specific displays, input devices [RD07, Del10]. In [Del12a], in addition to a specific monitor and console, a USB interface is also considered. Nordbotten and Gjertsen built a system where a console manager and a display manager are each encapsulated into a partition [NG12].
- Virtual, providing one console channel to one program or to a group of programs belonging to a single security domain, but running itself within some form of transport that can multiplex multiple such virtual consoles. Such transport can route to a local physical console or to something else, say, over a network connection offering adequate security properties.

4.1.3.1 Services

Input, output (e.g. display) streams

Multiplexing of streams

A physical console, in addition to a display device and human-machine input devices, can provide physical connection ports for external devices. Unlike external ports that would be associated with the computer itself, these external ports are meant to be associated with the

current operator. HMI devices such as displays, controllers, audio devices, usually are of this nature and are simply managed by making them available to the program or programs of the current operator. If the console can be switched between operators, then a policy must be devised for switching these devices as well, or not.

Some devices can be connected to a console, that are themselves concerned with multi-domain security. An example would be a mass-storage device through the file system component. Policies that make sense include:

- Mapping the device to the computer rather than the console, e.g. in the case of a mass-storage device, honouring file permissions and ownerships inside the regular file system component.
- Mapping the entire block device to the programs of the current operator and letting them access arbitrary locations in the device, which now cannot be trusted by other programs.

4.1.3.2 Architecture

Data and control streams are separated [Del12a, p. 48], and passed from its clients to hardware for input and output. If not all channels are dedicated, then resources are scheduled for reuse ("multiplexing"). The architecture avoids information flow when a resource is reallocated.

A console capable of serving multiple security domains at the same time can disambiguate which one or ones are presented to the user by:

- Reserving a trusted portion of the display for telling what is displayed on the rest of the display and allowing the selection thereof. This must be "always invoked" in a very literal sense, meaning that a full-screen application cannot be supported, or an auxiliary display must be added.
- Providing a "secure access key" that cannot be overridden by applications, that lets the user invoke a trusted status/selection panel that is overlaid on applications' displays. One must be very careful that operators are trained to ignore what they see if they are not positive that they invoked the trusted status/selection panel, as a malicious application could impersonate that panel, effectively realizing a Trojan horse, since applications have access to the display area where the trusted panel is shown. This also requires a guaranteed response time for showing the trusted status/selection panel after pressing the secure access key, otherwise there would still be a temporary opening for a Trojan horse.

4.1.3.3 Assumptions on the environment

A separation kernel is available. The separation kernel does not bypass the console component.

4.1.4 Network system component

A *network system component* is a MILS component having the tasks (1) of abstracting the used network infrastructure and topology connecting the MILS system with other platform-external systems and (2) of abstracting or hiding the physical location of a partition's communication partners. Usually the network system component also (3) abstracts the access mechanism to the network device and, thus, is a special purpose instantiation of the Generic Device Abstraction Component. Note that a network system component can be very complex

and may by implemented by multiple partitions running encapsulated sub-functions for handling this complexity. For example, partition *A* could contain TCP/IP stack *A'*, partition *B* could contain TCP/IP stack *B'* and partition *C* could make the decision to route packets either through *A* or *B*.

For fulfilling task (1), the abstraction of the used network infrastructure and topology, the component has to implement the used network infrastructure protocols. The border between application-level protocols and infrastructure protocols is usually fluent, depending on the required means of communication. However, as example one could draw the border between layer 4 and layer 5 of the OSI model, i.e. that the network system component implements the protocol stack up to UDP, TCP, … and leaves the implementation of higher layers up to the connected partition. The network system component is mentioned in [UV05], with e.g. implementing CORBA, DDS, HTTP, SOAP. The task of the network system component on the ingress data traffic is to analyse the routing information and to route the ingress data to the associated connected partition correctly. This may or may not include reassembling of the data stream, depending if the connected partitions require lower protocol stack levels for their purposes or not. However, for full abstraction of the network infrastructure, the network system component should reassemble the data stream and provide only the application-level payload to the connected partitions. For the egress traffic, the partition provides the application-level payload to the network system component, which generates valid data network packets and transmits them to the correct partition (if on the same platform) or transmits them via the network link.

Task (2), the abstraction of the communication partner's physical location, is another task performed by the network system component. From the application point of view encapsulated in the boundaries of its partition, the application does not know whether the communication partner is located on the same hardware platform or platform-externally. The task of the network system component is to determine the location of the communication partner and the correct routing of the data stream.

Task (3), the abstraction of the device interaction (i.e. the driver), applies only if the MILS system is actually connected to a network. To this task also applies to virtualize the network device to allow network sharing among the connected partitions.

By implementing all three tasks, the network component is required to ensure separation of data stream, in particular if one instance of this component handles data streams of different criticality (thus the component is MLS). Having such an MLS implementation may also require considerations on load-balancing and Quality of Service on the network link. For reducing complexity, the system designer should consider to implement multiple instances of network system components handling data of only one criticality (SLS components). However, this is only feasible if the system possesses multiple network devices or the network device is capable to support hardware virtualization technology.

For ensuring separation, it is also conceivable to use other MILS components, such as crypto components, which apply cryptographic methods to the data stream beforehand sending it to the network component.

Similar proposals for a network protocol component occur as MILS network system protection profile (MNSPP) [RD07, Del10, Del12a]. Other related work mentions a Partitioning Communications System (PCS) [AFOB+12, Uch07] or MILS Message Router (MMR) [AFOB+12, AFHOT06, ZSP+12]. The described functionality of those components is similar to a subset of the network system component. However, it is difficult to draw a clear

line between the functionality of the PCS compared to the one of the MMR. For avoiding complexity in terminology, we find it more intelligible to use the network system component to consolidate and cover the functionality of the PCS and the MMR.

4.1.4.1 Services

Functionally correct implementation of network infrastructure protocols.

Functionally correct implementation of the data routing to connected partitions including its reassembling (if applicable) of ingress data traffic.

Functionally correct segmentation of egress data streams received by connected partitions.

Functionally correct implementation of the device interaction and its abstraction.

4.1.4.2 Architecture

The network system component is a component interacting with other partitions using it. If the MILS system possesses network devices, the network system component interacts with a subset of the device's interfaces.

4.1.4.3 Assumptions on the environment

A separation kernel is available. The separation kernel does not bypass the network protocol component.

4.1.5 File system component

A *file system component* is a MILS component and an instantiation of the Generic Device Abstraction Component that implements file system services. It is described in [RAV07]. The purpose of the File system component is the abstraction of the access mechanism and the physical location of the block devices storing data permanently. For decoupling the physical location of the storage, the component could use the services of the Network component. To maintain the separation properties, the component has to ensure separation in a physical or logical (or both) way:

- Physical Separation: by storing data of different partitions on different physical locations of the storage volume (i.e. using the hard disk partitions) or on different storage volumes.
- Logical Separation: by applying cryptographic methods (e.g. provided by a crypto component) or special storage patterns using the same storage partition (e.g. gap storage with different offsets, special file system formats, …).

4.1.5.1 Services

Functionally correct implementation of the applied separation mechanism to ensure data separation of stored data.

Functionally correct implementation of the access mechanism to the device (i.e. driver), if the storage device is located on the same hardware platform.

4.1.5.2 Architecture

The file system component is a component interacting with other partitions using it. If the storage device is located remotely the file system component may interacts with other components as well.

4.1.5.3 Assumptions on the environment

A separation kernel is available. The separation kernel does not bypass the file system component.

4.1.6 Audit system component

An *audit system component* is a MILS component that implements audit services that can be used by other components [Del12b, p. 24].

4.1.6.1 Services

Functionally correct implementation of audit system.

4.1.6.2 Architecture

The file audit component is an optional component interacting with other partitions using it. The benefit of audit can be (1) to document that an entity has received a piece of information (non-repudiation) and (2) to monitor the MILS system, e.g. for information flow policy violations by components, (3) get event notifications from partitions to the audit system.

4.1.6.3 Assumptions on the environment

A separation kernel is available. The separation kernel does not bypass the audit system component.

The separation kernel supports auditing [Del12b, p. 24].

A messaging system is available [Del12b, p. 24].

The compilation of memory structures is supported [Del12b, p. 24].

The audit system is able to retrieve information about the origin of the audit information it is supposed to store.

4.1.7 Generic application component

4.1.7.1 Services

The generic application component implements any functional service required by an application.

4.1.7.2 Architecture

No statement can be made on the architecture of a generic application component. The system integrator can choose to configure a generic application component so that it is confined to a precise time-slot, limited memory and tightly controlled communication, so that it is not needed to trust its developer of the application, even if he is malicious. This kind of application is usually called “untrusted application”. In other scenarios, it may be meaningful to give the application strong access to the system, and even trust it do enforce a security policy for other applications, such as an information flow policy, e.g. when the application acts as a downgrader. This application is usually called “trusted application”. A trusted application can serve as guard to any application, whereas an untrusted application only can serve as guard to applications that are even less trusted.

4.1.7.3 Assumptions on the environment

The generic application component may assume the existence of other components, e.g. network component, other generic device abstraction component.

4.2 Hardware components

4.2.1 Introduction

[SKPP] formulates hardware requirements for separation kernel in the non-standard class "platform assurance" (APT). They are again discussed in [AFHOT06]. [AFOB+12] discuss the security needs of separation kernels with regards to existing multicore architectures.

[Tri12] discusses in particular on the topic of hardware requirements for mixed-criticality systems (safety and security) from the perspective of aviation computer systems and formulates current research directions.

In general hardware requirements for MILS systems are dependent on the MILS architecture itself and the external interfaces required by the system's functionality. If the MILS architecture relies on a separation kernel as fundamental component for implementing the separation and information flow property of MILS, the basic hardware requirements are defined by the separation kernel. In general separation kernels rely on common hardware protection units as the Memory Management Unit (MMU) and recently also Input/Output MMUs (IOMMUs). In addition, separation kernels also use hardware timers.

Those units are essentially the only functionally indispensable hardware elements for a separation kernel that are specified to be robust against attacks through illicit information flows, i.e. internal partition interference or malicious flows by misusing external interfaces (remote attacks). Any added hardware elements exist rather for in-depth defence, for added safety against (random) hardware failures, or for robustness against physical local attacks (mechanisms such as authenticated boot and OS code, storage for secrets, etc.).

4.2.2 Processing units

Processing units, such as processor cores or special purpose co-processor, are essential parts of MILS systems. Processing cores are responsible for processing the software-based MILS components by using other system resources. By following the control flow encoded in the software component's programming code the cores are able to achieve the intended component's objectives, usually by interacting with other hardware resources, such as memory or devices. Even if the major purpose of processing units is their ability to execute the binary code, they also have requirements with respect to spatial and temporal separation. In particular this applies to the interaction with the memory hierarchy comprising of various cache-levels and system's memory. However, it also applies to the internal processing flow of the processing cores, which have to ensure separation, too.

During partition runtime especially challenging are concurrent memory and device accesses of novel multicore processing platforms, due to the measureable interferences in access times depending on the amount of active cores [NP12]. During partition switch for the purpose of ensuring spatial and temporal isolation software (usually the separation kernel) has to ensure the proper sanitization of the (shared) resources used during processing the control flow. This includes the flush of core-internal pipelines or caches to prevent cache attacks [YF13, SBY+13].

One important mechanism for ensuring spatial and time separation is the provision of different execution modes for commands processed by the processing units. For example changing critical configuration of other hardware component, like MMUs settings, needs to

be restricted in a way that only privileged software, e.g. the SK can execute the commands for modifying those settings.

4.2.3 Memory Management Units (MMUs)

MMUs translate virtual addresses used by the processors into physical addresses required for interacting with the resource memory. In general this component can also be used for protecting certain memory area from processor accesses, it thus enforces an access control policy. Dependent on the architecture of the MMU and its way to maintain the translation tables, the MMU can be configured in a static or dynamic way:

- Static means that all partition applications have static entries in the MMU's translation table construct, which do not change during system runtime. If identical virtual addresses are used multiple times in various partitions, the hardware has to provide a runtime mechanism for indicating which partition is currently active and indicating the correct MMU translation entries (e.g. runtime identifier or reconfiguration of pointers to the translation tables). Such a static MMU configuration also implies a static spatial separation of the memory without dynamic (re)allocation of memory regions for partitions.
- Dynamic means that the separation kernel has to reconfigure the translation tables during partition switch. This approach does not require the previously mentioned hardware platform identifiers but might require additional processing cycles during partition switch.
  Dynamic MMU configuration also allows realizing dynamic (re)allocation of memory during application runtime. However, the necessary increased trust in the reallocation mechanism is essential for assuring the security properties of the separation kernel (e.g. zeroing memory after memory release). Additionally, dynamic MMU configuration can be useful for implementing performant inter partition communication, since the ownership of communication pages can be shared or transferred between partitions for purpose of avoiding the overhead of data copying.

Note that some available separation kernels use a combination of both mechanisms, e.g. for realizing a static spatial separation of the memory but also allowing shared pages for fast inter partition communication.

A security vulnerability of current MMUs is their level of trust put into the reliable operation of its configuring software, e.g. the separation kernel. More specifically this means that the separation kernel is able to interact with memory pages actually belonging to partitions "privately", without being visible to the SK. [JH11] discusses this issue and provides hardware improvements for future MMUs. For example, [JH11] propose that a VM can mark its page as private (in hardware) after allocation from a hypervisor (analogous to a separation kernel in our context). Having the private bit set this page can only be accesses by the VM and the hypervisor only can sanitize it as soon as the VM allows it. Encoding new features into MMU hardware, of course again raises the problem of ensuring that the hardware realization of this approach is correct.

4.2.4 Input/Output Memory Management Units (IOMMUs)

An IOMMU provides transparent, isolated access to virtual instances of I/O devices to one or more partitions [KS08]. These virtual device instances can be used just like a physical instance of the same I/O device by these partitions. Other partitions have no access to these

virtual devices, nor can the virtual devices access memory spaces of partitions other than the ones they have been assigned to.

If the system's functionality demands to use external DMA-capable devices, hardware components as IOMMUs are helpful to protect the system memory from invalid DMA triggered by the device and thus, to achieve spatial separation. The task of IOMMUs is similar to the one of MMUs. However, there are two differences to MMUs:

1. MMUs are placed between the processor and the system memory. The location of IOMMUs is between devices and the system memory.
2. The intention to apply MMUs into hardware was to increase the performance for address translations between virtual and physical addresses. Later on, its use for memory protection has been introduced. The motivation of using IOMMUs is the other way around. Primary IOMMUs have been deployed for memory protection reasons but can also be used for address translation. However, using the address translation mechanism smartly can open the opportunity of sharing hardware devices usually not intended to be shared, e.g. by reconfiguration of the address tables on partition switch. Thus, an IOMMU can provide transparent, isolated access to virtual instances of I/O devices to one or more partitions [KS08]. Nevertheless, this approach is only possible for stateless devices with immediate and short processing which only perform DMA on behalf of cores, e.g. external FPU or vector processing engines.

IOMMUs are getting required in a system in which DMA-capable devices shall be directly assigned to an untrusted partition, i.e. an untrusted driver shall be allowed to interact without additional software-based checks of the separation kernel (e.g. for performance reasons).

Since in such a design the untrusted driver can access the entire memory abusing the directly assigned device by triggering DMA to addresses outside of its allowed memory resources, the hardware requires a component to restrict those accesses. This is the task of the IOMMU.

For proper hardware architectures with IOMMUs it is necessary that the IOMMU identifiers used for device's identification are provided in a secure way. In particular [SLN+10], [SV10], [WR11] and [MIM+13] discuss attacks using DMA and harming IOMMU-based hardware designs. One class of those attacks abuses Message Signalled Interrupts (MSIs) to trigger interrupts which do not belong to the device. These attacks are possible since former IOMMUs only mediated transfers based on (1) the accessing device, (2) the involved addresses and (3) the operational code for the transaction but ignoring the data content of the transaction. For example, Intel counteracts the class of attacks by a technology called "Interrupt Remapping", which validates also the interrupt vectors (messages) of the MSI [Int11]. Another class of attacks uses a vulnerability of PCI to PCIe bridges, where the identifier is added by the bridge but not by the devices connected to the bus "behind" the bridge. More generic views on this issue introduce discussion on suitable device interconnect topologies. The interconnect topology should provide the separation kernel possibilities to uniquely identify the physical hardware interface (e.g. card slot) the device is connected to. In general a bus strategy achieves this requirement worse than a star topology.

In addition IOMMUs usually do not apply countermeasures against devices performing timing attacks, like exhausting bandwidth, interrupt bombing or uninterruptible long bus transactions (a timing attack on latency that can alter real-time properties without needing to saturate the bus). Some timing attacks again various in their utilization on the used interconnect topology.

4.2.5 I/O sharing

A special case of directly assigned device interfaces is the approach of using self-virtualizing devices. With this technology it is possible to securely share a device without requiring trusted software components for runtime device interactions (runtime driver). For example it may also allow transferring parts of the functionality of the network component into the hardware. Using this technology the hardware device provides a physical interface for configuration purposes and a various number of virtual interfaces appearing as runtime interface to the partitions that shall interact with the device instance. A special standard called Single Root I/O (SR-IOV) [SRIOV] extends the PCI Express (PCIe) standard and defines the hardware interface for PCIe devices. To restrict DMA of virtual functions to the assigned partitions only, an IOMMU is essential. Further investigation on platform requirements using PCIe SRIOV is provided in [MIM+13].

Work on performance comparison of software-based and hardware-based I/O sharing approaches are provided by [YYW08] and [WR08]. Both publications conclude that hardware-based sharing using IOMMUs and direct mapping almost performs with native performance. However, [WR08] additionally investigates on the provided granularity of memory protection (inter- and intra-guest) of software-based approaches compared to different strategies for reconfiguring the IOMMU on partition switch. The final statement of this work is that software-based pre-validation of DMA descriptors performs better than some approaches (not direct map!) for hardware-based late validation of DMA transfers. Also software-based sharing strategies enable enhanced intra-partition memory protection with respect to the granularity. However, the downside of software-based approaches is their inability to protect against device misbehaviours and the required assurance property of the software components.

4.2.6 Timers

Separation kernels are in charge to provide separation properties in time and space for a MILS system. Regarding time separation (e.g. real-time scheduling of applications) the kernel requires a reliable signal defining the unit “time” for the system. For this purpose hardware normally provides a periodic and stable transducer in combination with a counter counting the generated signals. Knowing the frequency of the transducer allows defining the resolution and thus the smallest possible unit of time in the system. Both the transducer and the counter together build the basics for implementing timers. Separation kernels use timers in one of two fashions:

- Inflexible periodic timers that give rise to a so-called «tick» timer in the kernel, periodically fired irrespective of whether there is activity to be carried out or not. A number of OSes have this design because they are backwards compatible with the Intel 8253 Programmable Interval Timer (PIT) that was the only timer chip found in the original IBM PC (discounting the alarm function of the MC146818 RTC chip that does not have a high repeat rate), even though modern PC-compatible hardware has better timers.
- more flexible arbitrarily programmable timers that give rise to a «tickless» kernel that wakes up only when necessary. Intel/Microsoft High-Precision Event Timers (HPET, [Int04]), formerly known as Multimedia Timers because they originated from the need for high-resolution arbitrary timers for sound generation in desktop PCs, provide this capability with a free-running counter and comparators although a subtlety of this

hardware implementation for some software designs is that a timer must be armed in the future only, e.g. it will not trigger if armed too late just based on the fact that the comparison is "now true". Alternative designs typical of microcontrollers involve downcounters with a feature for auto-reloading the timer with the next deadline that was provided by the software ahead of time. This trivially eliminates jitter, whereas downcounters without auto-reload have to be compensated in software by accounting for the time lost between the previous deadline and the time when the software actually loads the next deadline from some interrupt handler, and while it is easy to compensate for absolute drift, jitter or a small lateness can never be completely eliminated on processors where writing fully time-deterministic code is impractical.

One timer is usually sufficient. Having several timers available may yield simpler or faster software, although an implementation can be fairly simple with just one hardware timer if that timer has just the right flexibility.

4.2.7 Chain of trust

A last important fact necessary to mention in this chapter is the topic of trusted initialization of the different layers in a MILS system. Usually these different layers are initialized in a well-defined sequence, e.g. first general boot code, followed by the layer providing the separation property, followed by other layers providing system-specific security functions and applications. To ensure that the entire sequence is not compromised a root of trust is needed at the beginning of the sequence. Usually a special hardware component storing a secret key and a hard coded boot code provides this feature. Thus, also hardware components implementing the root of trust can be necessary [Fre10]. Regarding trust in the software involved in the boot sequence, there have been 2 schools of thought:

- All software from the reset vector (possibly with the assistance of firmware in an internal ROM) is trusted and therefore hardware-assisted mechanisms are provided to verify initial trust, and then it is up to this trusted software to preserve the chain of trust to the next trusted software until usual hardware protection mechanisms (user/supervisor mode and memory protection) are used to allow controlled execution of untrusted code. This is the pattern used by the IBM/Sony Cell BE™ [Shi06], Freescale's Secure Boot [Fre11a] and Trust Architecture [Fre11b] and, to our knowledge, ARM's TrustZone® [ARM13].
- Boot software is not trusted, but hardware mechanisms exist in order to re-establish a trusted context later on, or let trusted software establish that initial software was not altered nor bypassed and therefore could only have taken known action. This is the pattern used by the Trusted Computing Group™'s Trusted Platform Module [TCG11].

4.3 System configuration of components

The configuration of a MILS system comprises the configuration of the separation kernel, and the configuration of other components, such as the configuration of applications, and the configuration of hardware.

4.3.1 Configuration of the separation kernel: configuration space

We have defined the separation kernel to be the main policy-enforcing element of a MILS system, using hardware mechanisms provided by the hardware in the MILS core. Thus, its configuration options to a large extent need to reflect the configuration of a MILS system. In Section 3.2.13 of this document, a separation kernel has already been characterized as

enforcing the resource allocation, access control and information flow policies. Thus, the configuration of a separation kernel equals the configuration of these policies.

The above definition is fairly abstract. Giving an exhaustive, yet product-independent list of configuration parameters is non-trivial, and perhaps not even desirable: If we start with SKPP, despite the string "configuration" occurs at least 510 times in SKPP, SKPP does *not* give a comprehensive list of configuration data at one place. For example, SKPP mentions system memory and processing time per partition [SKPP, p. 78] and then information flow policy configuration data, audit configuration data, clock settings, and self-test period as other examples [SKPP, p. 175]. Taking into account that an operating system used for IMA (recall Section 2.1) can be provided by a separation kernel ("*a separation kernel is similar to the "partitioning kernels" used in integrated modular avionics (IMA), but is more aggressively minimized*") [BDR+08, p. 9], possibly a better, more concrete, yet still product-independent example can be found in [ARINC-653]. For an IMA operating system, [ARINC-653, p. 22] specifies that, (1) for each partition, its memory requirements, its scheduling parameters (period, duration), identity of messages to be sent/received by the partition are configured by a configuration table, (2) globally, that a configuration table of inter-partition communication objects is kept and a fault handling is configured.

4.3.2 Configuration of other components: configuration space

However, note that the separation kernel configuration only addresses part of the overall MILS system configuration. For example [AFOB+12, p. 181] emphasizes that, in addition to the configuration layer at the separation kernel level, the configuration of a MILS system is also strongly determined by the configuration of its hardware. For example, the configuration of a MILS system includes which PCI slot to use for which PCI card, the memory mapping of hardware and so on.

4.3.3 Configuration management

Configuration management: The need of configuration management for secure systems is addressed by the [CC12] in general and, more particular, for IMA systems in [DO-297, Rom08]. It is emphasized that to reproduce the configuration of a system using a separation kernel, the configuration of each level must be stored, including hardware and configuration data of applications running in partitions managed by the separation kernel. [SKPP, p. 17, 27] defines (1) the generation of an abstract configuration vector by a configuration tool, (2) its transformation to machine-readable configuration data on a boot medium by a load function, and (3) its usage by a boot function during operation. Also, [ARINC-653, p. 22] stipulates that configuration tables of an IMA operating system must be built separate from the operating system and they are not directly accessed by applications; an implementation detail that of course is only binding for a separation kernel if it is to be used for an IMA system. However, except for that mention of separate build of configuration tables that is not a requirement in [SKPP], detailed configuration workflow guidance for an entire MILS system is out of scope and rather scarce in this IMA [ARINC-653] *application* software standard interface description.

Reconfiguration: Reconfiguration of a system is making some change(s) to the configuration of that system; we call that a *configuration change*.

A configuration change modifies the system configuration data. For example, in the separation kernel the Information Flow Policy could be modified. When a configuration change occurs by going the system offline and reboot, the change is called a *static*

configuration change. When the change occurs on-line without reboot during an execution, then the configuration change is *dynamic* [SKPP, p. 16, p.40; NLI, Section 5.2]. If configuration change capability is not built-in into a separation kernel, it can be implemented by the component of the MILS platform, for example select or upload another image of the separation kernel into the MILS platform or a partition component that specializes in doing this. Another example would be the dynamic configuration of virtualization hardware, which e.g. could be done from within a partition. In this case, you have already configured the virtual interfaces for the partition in the separation kernel, and then you connect the virtual device hardware to them.

4.3.4 System update

Related to topic of configuration management is the treatment of system updates of the MILS components. A common automotive use case for reconfiguration is a software update of possibly every software component in the system. The security policy for system updates typically specifies that system updates cannot be done by the internet but only locally via the on-board bus.

However, many automotive manufacturers (OEMs) tend to require software updates ‘over the air’ and request for improved methods to guarantee (1) fail safety (robustness in case of failures during the update procedure), (2) integrity (updating sources other than originated by the OEM must be rejected) and (3) security (the software update mechanism must be resistant against attacks). Since access control policies themselves may be subject of software updates, hence modification, special care must be taken to self-protection.

Chapter 5 Conclusion

5.1 Overview of component policies and reuse

An overview of policies enforced and usage of services by other components is given in Table 4.

In Table 4, for "provides" or "used-by" relations, an "M" means "provision/use is mandatory", an "O" means "provision/use is optional". In the case of "M" for "component *X* used by component *Y*" component *X* is meant only as mandatory for component *Y*, if the MILS system has component *Y* at all (this also may not be that case). As it is always an implementation option, for brevity, we do not consider self-use or self-invocation in this table. A component is a guard if it enforces some resource allocation policy, access control policy and/or information flow policy in the sense (2a) or (2b) of Section 3.2.7.

	Component provides resource allocation policy	Component provides access control policy	Component provides information flow policy	Component is used by separation kernel	Component is used by console component	Component is used by network system component	Component is used by file system component	Component is used by audit system component	Component is a guard
Software components									
Separation kernel	M	M	M		M	M	M	M	M
Console system component		M		O				O	M
Generic device abstraction component	O	O	O	O	O	O	O	O	O
Network system component	M	M	O	O	O		O	O	M
File system component		M	O	O		O		O	M

	Component provides resource allocation policy	Component provides access control policy	Component provides information flow policy	Component is used by separation kernel	Component is used by console component	Component is used by network system component	Component is used by file system component	Component is used by audit system component	Component is a guard
Audit system component		M		O		O	O		M
Application (trusted)	O	O	O						O
Application (untrusted)	O[1]	O[1]	O[1]						
Hardware components									
Processor		M		M	M	M	M	M	M
MMU		M		M		M/O[2]			M
IOMMU		M		M[3]	M[3]	M[3]	M[3]	M[3]	M
I/O sharing		M				O	O[4]		M
Timer				M					
Chain of trust		[5]		O					M

Remarks:

[1] A trusted application can serve as guard to any application, whereas an untrusted application only can serve as guard to applications that are even less trusted.

[2] MMUs may be needed for network components depending on hardware, e.g. on PowerPC network devices are memory-mapped. Also on Intel, the entire PCI express is memory mapped.

[3] If and only if DMA is used.

[4] E.g. a physical harddisk that is accessed by the file system component.

[5] As a chain of trust denies access if a signature is not provided properly, it can be seen either as access control policy or as integrity policy.

Table 4: Policies enforced and usage by other components

We observe that access control policy is provided by almost any component, a resource allocation policy or information flow policy is more rarely encountered. A timer and a chain of trust do not implement their own access control/resource allocation/information flow policies, but can be used by the separation kernel to support resource allocation and integrity requirements.

5.2 Secure design principles

In Table 5 we compare our MILS experience with the Saltzer and Schroeder Design Principles [SS75] previously introduced in Section 2.4. It can be seen that many principles carry over to MILS systems. Those principles that are not fully carried over are those which clash with the stringent performance and real-time requirements of MILS systems.

Design Principle (as in [SS75])	**Explanation** (as summarized by [Bis00])	**Implementation in MILS**
Economy of Mechanism	The protection mechanism should have a simple and small design.	Some MILS components, such as the separation kernel, are small.
Fail-safe Defaults	The protection mechanism should deny access by default, and grant access only when explicit permission exists.	The default policy in a MILS system is: no information flow and no resource sharing unless specified.
Complete Mediation	The protection mechanism should check every access to every object.	This is implemented by a small reference monitor, the separation kernel.
Open Design	The protection mechanism should not depend on attackers being ignorant of its design to succeed. It may however be based on the attacker's ignorance of specific information such as passwords or cipher keys.	MILS design is comparatively well understood and open.
Separation of Privilege	The protection mechanism should grant access based on more than one piece of information. (e.g., two commanders need to agree to launch a weapon).	For performance reasons, and because this kind of policy is not so common in embedded systems, this is usually not implemented in MILS systems.
Least Privilege	The protection mechanism should force every process to operate with the minimum privileges needed to perform	This is usually only implemented at a partition granularity level in MILS systems (the calculation of

Design Principle (as in [SS75])	**Explanation** (as summarized by [Bis00])	**Implementation in MILS**
	its task.	the "minimum privileges" can be non-trivial).
Least Common Mechanism	The protection mechanism should be shared as little as possible among users. (e.g. shared variables shall be avoided)	An example implementation of this principle is that middleware (user space libraries) is usually put into partitions of a separation kernel.
Psychological Acceptability	The protection mechanism should be easy to use (at least as easy as not using it).	Use of the protection mechanism is implemented by fail-safe defaults. The decomposition of a system into partitions requires some initial effort, but in the long run makes it easier to understand and maintain its functionality.

Table 5: Secure design principles and their implementation in MILS

5.3 Results

We have identified and described the origins where MILS comes from (Chapter 2) and established a foundation we can use for the description of the architecture of MILS systems. For example, we have obtained a common "picture" of a MILS system (Section 3.1). We have also created working definitions for fundamental MILS terms in a bottom-up way, including definitions of closely related security policies such as access control policy, resource allocation policy or information flow policy (Section 3.2). Several iterations were needed to obtain this in a clean, yet understandable way, which may explain why we have not seen this bottom-up approach done elsewhere.

We have also identified some widely used terms we chose to avoid, such as "PCS" or "middleware", because we consider them of little help and even misleading. We were able to apply the terminology to previous work on security gateway (Section 3.3).

We have reached consensus to present hardware and software components as equal citizens and compiled a catalogue of MILS hardware and software components, including a security-centric description (Chapter 4). In particular, we have identified a generic device abstraction component. We have identified several instances where separation kernel policy enforcement depends on guarantees by hardware components. In Section 5.1, we have summarized security policies provided by components, mutual interdependencies of components and classified components as "guards". This could serve as a basis for a more detailed analysis of information flows and their guards of concrete components as proposed in [AFOB+12, Chapter 4]. Section 5.2 establishes that MILS largely follows well-established principles of secure system design. Our document appears to be a reasonable basis for further description of individual components within the EURO-MILS project.

5.4 Acknowledgment

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 318353. We thank Rance DeLong (Open Group) for commenting on some parts of an earlier version of the text.

Glossary

Access control policy: A component's *access control policy* acts on the component's interface used to manage exported resources. In this respect it is identical to the aforementioned resource allocation policy (Section 3.2.5). However, the interface is characterized by that a request to the resource includes an explicit reference to the resource (e.g. the resource's name or a numerical identifier). Identically to the aforementioned resource allocation policy (Section 3.2.5), the access control policy defines which of the component's resources are kept internal to the component and which are exported to which other components. When a resource is exported to more than one other component, the resource is *shared*. The access control policy is in the "space" domain.

Application: An *application* is one or more executable(s).

Audit System Component: An *audit system component* is a MILS component that implements audit services that can be used by other components

Communication object: A *communication object* is an exported resource provided by a component. It can be shared between components. Communication objects are used by components to communicate between them.

Component: A *component* is a term to describe the decomposition of a (in general, *any*) system into meaningful self-contained parts. For example, a (yet to be defined) MILS system consists of components. In general, components may be implemented by (1) hardware, (2) software, or (3) a combination of hardware and software [CBB+03, DO-297].

Configuration: The *configuration* of a component contains the component's identity, and it defines any security policy (access control policy, resource allocation policy, information flow policy) enforced by the component. An information flow policy configuration also may be implicitly configured by resource allocation policy configuration and access control policy configuration.

Console system component: A *console system component* connects applications to human interface devices, and thus is an instance of the Generic Device Abstraction Component.

Domain: A *domain* (or "security domain") is a unit of separation created and maintained by any MILS component, for example by an application (Section 3.2.9), a function (Section 3.2.12), or the MILS core (Section 3.2.14), which is enforcing a security policy on exported resources.

File system component: A *file system component* is a MILS component and an instantiation of the Generic Device Abstraction Component that implements file system services.

Function: A *function* is a logical group of partitions for achieving common objectives. The implied partitions may be connected using information flows.

Generic device abstraction component: A *generic device abstraction component* is a MILS component having the purpose of abstracting the access mechanism of a special purpose hardware device to a defined set of connected partitions.

Information flow policy: The term *information flow policy* has more than one usage,

(1) the most simple one is to use it as an umbrella term for "access control policy" and "resource allocation policy" combined or

(2) to express policies where pieces of information (messages) are written to one or several communication objects(s) by a *sender* and subsequently these messages are read from the communication object(s) by a *receiver*. Such policies may include rules based

(2a) on the sender/receiver of the messages and/or

(2b) on the *content* of these messages.

Note: for most components, interpretation (1) is used. (2a) will be used in the context of a separation kernel (Section 3.2.13). The enforcement of (2b) is a typical task of security gateway (discussed as an example in Section 3.3). An information flow policy in the sense of (2a) is either explicit, based on identities of components between which information flow is allowed, or implicit, as unambiguously defined by the resource allocation policy and access control policy.

MILS architecture: "*MILS architecture*" refers to the architecture of the implementation of a concrete MILS system.

MILS architecture template: "*MILS architecture template*" refers to a template encompassing many possible MILS systems.

MILS platform: A *MILS platform* consists of the MILS core and optional software and/or hardware components that provide secondary security functionalities and do not contribute to the enforcing of separation.

MILS system: A *MILS system* is a concrete deployment of a MILS platform with a defined set of partitions.

MLS system: An *MLS system* is a system with different security requirements for different components. It can be implemented by a MILS system.

Multi-level Secure (MLS) component: A *Multi-Level Secure Component* is a component that handles information of with different security levels concurrently during one runtime instance.

Multiple Single-Level Secure (MSLS) component: A *Multiple Single-Level Secure Component* is a special kind of SLS component that processes data of multiple security levels, but always maintains separations between classes of data by exclusively processing only one security level during its runtime instance. For example this separation can be implemented by allowing access to a different security level only when the component has rebooted with different parameters. Also deploying multiple instances of one SLS component processing different single security levels turn this SLS component into an MSLS component.

Note: in [Alv98] this was restricted to temporal separation, "at a single time-point, only handles information from one component". If such a single-level process is to be implemented as untrusted process [Alv98], it can be supplemented by an appropriate labelling and filtering of messages.

Network system component: A *network system component* is a MILS component having the tasks (1) of abstracting the used network infrastructure and topology connecting the MILS system with other platform-external systems and (2) of abstracting or hiding the physical location of a partition's communication partners. Usually the network system component also

(3) abstracts the access mechanism to the network device and, thus, is a special purpose instantiation of the Generic Device Abstraction Component.

Partition: A *partition* is a component that serves to encapsulate application(s) and/or data. Thus, the content of a partition is application(s) and possibly other data. A partition is a unit of separation with respect to

- resource allocation in the space and time domains,

an access control policy and an information flow policy in the space domain.

Resource: A *resource* is anything (processor such as a CPU or a processing core, memory, software, data, network, etc.) internally used or exported by a component. A resource may be physical (a hardware device) or logical (a piece of information). A resource may be shared by multiple components or be dedicated to a specific component.

Exported resources are those resources to which an explicit reference is possible via a component interface, e.g., the programming or configuration interface. *Internal resources* are those resources used exclusively by the component, and which have no explicit reference via a component interface.

Resource allocation policy: A component's *resource allocation policy* acts on the component's interface used to manage exported resources. This interface is characterized by that a request for a resource is made without knowing in advance how the resource is "named" or "addressed". The request is made for a quantity of the resource, and then the component decides whether to grant or deny the request to export that resource in the desired quantity. The resource allocation policy defines which of the component's resources are kept internal to the component and which are exported to which other components. When a resource is exported to more than one other component, the resource is *shared*. A resource allocation policy can be in the "space" domain, when resources can be used simultaneously but are kept in different spatial (e.g. memory) locations or in the "time" domain, where resources are used sequentially, but kept in different time slices. An example for resource allocation in the "time" domain is the allocation of a CPU to a component for a limited period of time.

Separation kernel: A *separation kernel* A *separation kernel* is a component that enforces a resource allocation policy and an access control policy on its exported resources (partition, resources allocated to a partition, communication objects). Communication objects allow for controlled information flow between partitions. A separation kernel may have an explicit or an implicit information flow policy on its partitions (see definition of information flow policy for details).

The separation kernel uses separation-supporting hardware to provide the separation between partitions in a MILS core.

Shared resource: When a resource is exported to more than one other component, the resource is *shared.*

Single-Level Secure (SLS) component: A *Single Level Secure Component* is a component that every time processes data of one security level.

System integrator: The person composing the MILS system from its components.

Virtual machine: A *virtual machine (VM)* consists of software that imitates a physical hardware machine. The virtual machine will for example give the illusion of a physical CPU and physical memory to an operating system that is running in it

List of Abbreviations

AFDX	Avionics Full Duplex Switched Ethernet
AMD	Advanced Micro Devices
CBEA	Cell Broadband Engine Architecture
CC	Common Criteria for Information Technology Security [CC12]
CDS	Cross-Domain Solution
COTS	Commercial Off-the-Shelf
CPU	Central Processing Unit
DMA	Direct Memory Access
EAL	Evaluation Assurance Level
HW	hardware
IMA	Integrated Modular Avionics
I/O	Input/Output
IO/MMU	I/O Memory Management Unit
IPC	Inter-Process Communication
LRU	Line Replacement Unit
MILS	Multiple Independent Levels of Security
MIPP	MILS Integration Protection Profile
MLS	Multi-Level Secure
MMU	Memory Management Unit
MSI	Message Signalled Interrupt
MSLS	Multiple Single-Level Secure
NEAT	Non-Bypassable, Evaluatable, Always

	Invoked, Tamperproof
NSA	National Security Agency
OEM	Original Equipment Manufacturer
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PCS	Partitioning Communications System
SK	Separation Kernel
SKPP	Separation Kernel Protection Profile
SLS	Single-Level Secure
SW	software
VM	virtual machine
VMM	Virtual Machine Monitor

Bibliography

[AA06] Keith Adams, Ole Agesen, A Comparison of Software and Hardware Techniques for x86 Virtualization, ASPLOS'06, p. 2-13, 2006, ACM, New York, NY, USA, http://www.ittc.ku.edu/~niehaus/classes/750-s09/documents/asplos235_adams-2006.pdf.

[AFHOT06] Jim Alves-Foss, Scott Harrison, Paul W. Oman, Carol Taylor, The MILS Architecture for high-assurance embedded systems, International Journal of Embedded Systems, vol. 2, no. 3--4, p. 239-247, 2006, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.6810.

[AFOB+12] Jim Alves-Foss, Paul Oman, Ryan Bradetich, Xiaohui He, Jia Song, Implications of Multi-Core Architectures on the Development of Multiple Independent Levels of Security (MILS) Compliant Systems, no. 0704-018, 2012, University of Idaho, Center for Secure and Dependable Systems, Moscow, Idaho, http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA568860.

[Air97] Airlines Electronic Engineering Committee, Avionics application software standard interface: ARINC specification 653, January, 1997, Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, MD 21401, http://www.arinc.com/.

[Alv98] Jim Alves-Foss, The Architecture of Secure Systems, Hawaii Interational Conference on System Sciences, p. 307-316, January, 1998, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.6431&rep=rep1&type=pdf.

[And72] James P. Anderson, Computer Security Technology Planning Study, no. ESD-TR-73-51, Oct., 1972, Deputy for Command and Management Systems HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, http://seclab.cs.ucdavis.edu/projects/history/papers/ande72a.pdf.

[And08] Ross Anderson, Security engineering, 2008, J Wiley & Sons, http://www.cl.cam.ac.uk/~rja14/book.html.

[ANS01] American National Standards Institute, ANSI X3.172-1996 American National Standard Dictionary of Information Technology (ANSDIT), Release 16, 2001, http://www.incits.org/ANSDIT/Ansdit.htm.

[ARINC653] Airlines Electronic Engineering Committee, Avionics application software standard interface: ARINC specification 653, January, 1997, Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, MD 21401, http://www.arinc.com/.

[ARINC811] Airlines Electronic Engineering Committee (ARINC), Commercial Aircraft Information Security Concepts of Operation and Process Framework, no. ARINC specification 811, January, 2005, Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, MD 21401, http://www.arinc.com/.

[ARM13] ARM Ltd., http://www.arm.com/products/processors/technologies/trustzone.php.

[ARP4754] Society of Automotive Engineers, Safety Assessment for Airborne Systems, Equipment Committee, ARP4754: Certification Considerations for Highly-Integrated Or Complex Aircraft Systems, 1996, Society of Automotive Engineers, SAE World

Headquarters, 400 Commonwealth Drive, Warrendale, PA 15096-0001 USA, http://www.sae.org.

[Avi08] Avionics designers choose SYSGO real-time embedded software for A400M cargo system, Avionics Intelligence, 10 Dec 2008, http://www.avionics-intelligence.com/articles/2008/12/avionics-designers-choose-sysgo-real-time-embedded-software-for-a400m-cargo-system.html.

[BBH+05]. William Beckwith, Carolyn Boettcher, Mark Hama, Jahn Luke, Tod Reinhart, High Assurance Safe and Secure Distributed Systems and Information Sharing, Infotech@Aerospace Conferences, 2005, American Institute of Aeronautics and Astronautics.

[BCK03] Len Bass, Paul Clements, Rick Kazman, Software Architecture in Practice, 2nd ed, Addison-Wesley 2003.

[BDR+08] Carolyn Boettcher, Rance DeLong, John Rushby, Wilmar Sifre, The MILS Component Integration Approach to Secure Information Sharing, Digital Avioncis Systems Conference (DASC), 2008, http://www.csl.sri.com/~rushby/abstracts/dasc08

[Bis00] Matt Bishop, Saltzer's and Schroeder's Design Principles, 2000, http://nob.cs.ucdavis.edu/classes/ecs153-2000-04/design.html.

[CBB+03] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford, Documenting Software Architectures: Views and Beyond, Addison-Wesley 2003.

[CC12] Common Criteria Sponsoring Organizations, Common Criteria for Information Technology Security Evaluation. Version 3.1, revision 4, vol. 1--3, September, 2012, http://www.commoncriteriaportal.org/cc/.

[Cordis12] CORDIS document server http://cordis.europa.eu/search/index.cfm?fuseaction=proj.document&PJ_RCN=13197414

[Cof12] Darren Cofer, Complexity-reducing design patterns for cyber-physical systems, 2011, Rockwell Collins, http://www.darpa.mil/uploadedFiles/Content/Our_Work/TTO/Programs/AVM/Rockwell%20Collins%20META%20Final%20Report.pdf.

[CVdM09] Stephen Chong, Ron van der Meyden, Using architecture to reason about information security, 2009, http://www.cse.unsw.edu.au/~meyden/research/arch-filter.pdf.

[DCS+04] John Detombe, Darin Cowan, Mike Smith, John O'Brien, Survey of Multi-Level Security (MLS) Products, no. CR 2004-268, 2004, Defence R & D Canada, http://cradpdf.drdc-rddc.gc.ca/PDFS/unc82/p523341.pdf.

[Del06] Rance DeLong, MLS with MILS?, slides, 2006, http://www.cisr.us/events/downloads/guests/delong.pdf

[Del10] Rance J. DeLong, An Evaluation and Certification Scheme for MILS, Fourth Annual Layered Assurance Workshop (LAW 2010), 2010, http://fm.csl.sri.com/LAW/2010/law2010-09-DeLong.pdf.

[Del12a] Rance DeLong, The Mils™ Architecture -- a Foundation for Dependable Systems, The Open Group Conference: Real-Time & Embedded Systems Forum, 2012, http://www.opengroup.org/public/member/proceedings/q212/23RT.htm

[Del12b] Rance DeLong, MILS Integration Protection Profile (MIPP) and the MIPP Commentary (slides), The Open Group Conference, Barcelona, Spain, 2012.

[Dod83] Department of Defense, Trusted computer systems evaluation criteria (Orange Book), DoD 5200.28-STD, 1983, http://csrc.nist.gov/publications/secpubs/rainbow/std001.txt

[DO-297] RTCA SC-200 / EUROCAE WG-60, DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations, November, 2005, Radio Technical Commission for Aeronautics (RTCA), Inc., 1828 L St. NW., Suite 805, Washington, D.C. 20036.

[DPF09] Julien Delange, Laurent Pautet, Peter Feiler, Validating safety and security requirements for partitioned architectures, Reliable Software Technologies--Ada-Europe 2009, p. 30-43, 2009, Springer, http://julien.gunnm.org/data/publications/article-dpf-rst09.pdf.

[DPK10] Julien Delange, Laurent Pautet, Fabrice Kordon, Design, Verification and Implementation of MILS systems, Proceedings of the 21th International Symposium on Rapid System Prototyping, 2010, http://pagesperso-systeme.lip6.fr/Fabrice.Kordon/pdf/2010-RSP.pdf.

[Fr83] Lester J. Fraim, Scomp: A Solution to the Multilevel Security Problem, Computer, vol. 16, no. 7, p. 26-34, 1983, IEEE.

[GH08] Olivier Gilles, Jerome Hugues, Validating requirements at model-level, IDM'2008 5-6 juin Mulhouse, 2008, http://www.idm08.uha.fr/actes/p5.pdf.

[GN09] Tor Gjertsen, Nils Agne Nordbotten, Multiple independent levels of security (MILS) - a high assurance architecture for handling information of different classification levels, 2009, Norwegian Defence Research Establishment (FFI), http://rapporter.ffi.no/rapporter/2008/01999.pdf.

[Gol73] Robert P. Goldberg, Architectural Principles for Virtual Computer Systems, 1973, Ph Thesis, Harvard, Cambridge, MA, http://www.dtic.mil/cgi-bin/GetTRDoc?AD=AD0772809.

[Gre08] Green Hills Software, INTEGRITY-178B Separation Kernel Security Target, no. IN-ICR750-0100-GH01ST, May, 2008, http://www.niap-ccevs.org/cc-scheme/st/vid10119/.

[HASK] Bundesamt für Sicherheit in der Informationstechnik (BSI), Sirrix AG security technologies, Protection Profile for High-Assurance Security Kernel: Version 1.14, June, 2008, http://web.archive.org/web/20110726034516/http://www.sirrix.com/media/downloads/54500.pdf.

[HHOAF05] W. Scott Harrison, Nadine Hanebutte, Paul W. Oman, Jim Alves-Foss, The MILS Architecture for a Secure Global Information Grid, The Journal of Defense Software Engineering, Crosstalk: The Journal of Defense Software Engineering, vol. 18, no. 10, p. 20-24, Oct., 2005, http://www.crosstalkonline.org/storage/issue-archives/2005/200510/200510-Harrison.pdf.

[Hou11] Carol S. Houck, Publications and Future Support for Separation Kernels, May, 2011, http://www.niap-ccevs.org/announcements/SKPP%20Email%20to%20Vendors.pdf.

[Int04] Intel Corporation, “IA-PC HPET (High Precision Event Timers) Specification”, 2004.

[Int11] Intel Corporation, “Intel® Virtualization Technology for Directed I/O”, 2011.

[ISA62433] International Society of Automation, Security for industrial automation and control systems, ISA-62443, 2013, http://isa99.isa.org/Documents/Drafts/.

[JH11] S. Jin and J. Huh, “Secure MMU: Architectural Support for Memory Isolation among Virtual Machines,” in 41st International Conference on Dependable Systems and Networks - Workshops (DSN-W), 2011, pp. 217-222.

[Kar05] Paul A. Karger, Multi-Level Security Requirements for Hypervisors, Computer Security Applications Conference, 21st Annual, 2005, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.101.6161.

[Kem83] Richard A. Kemmerer, Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels, ACM Transactions on Computer Systems, vol. 1, no. 3, p. 256-277, 1983, http://www.cs.ucsb.edu/~sherwood/cs290/papers/covert-kemmerer.pdf .

[KS08] P.A. Karger, D.R. Safford, "I/O for Virtual Machine Monitors: Security and Performance Issues," Security & Privacy, IEEE , vol.6, no.5, pp.16,23, Sept.-Oct. 2008

[KW07] Robert Kaiser, Stephan Wagner, Evolution of the PikeOS Microkernel, MIKES: 1st International Workshop on Microkernels for Embedded Systems, 2007, http://ertos.nicta.com.au/publications/papers/Kuz_Petters_07.pdf.

[KW08] David Kleidermacher, Mike Wolf, MILS Virtualization for Integrated Modular Avionics, Digital Avionics Systems Conference (DASC), p. 1.C.3-1-1-C.3-1-8, 2008, IEEE.

[Lam71] Butler W. Lampson, Protection, Proc Fifth Annual Princeton Conference on Information Sciences and Systems, p. 437-443, 1971, Princeton, http://research.microsoft.com/en-us/um/people/blampson/08-Protection/Acrobat.pdf.

[LRP+11] Joseph Loyall, Kurt Rohloff, Partha Pal, Michael Atighetchi, A Survey of Security Concepts for Common Operating Environments, Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on, p. 244-253, 2011, https://dist-systems.bbn.com/people/krohloff/papers/2011/Loyall-WORNUS-CameraReady-Paper1.pdf.

[MIM+13] Daniel Münch, Ole. Isfort, Kevin Müller, Michael Paulitsch, Andreas Herkersdorf. Hardware-Based I/O Virtualization for Real-Time Embedded Avionic Systems Using PCIe SR-IOV. International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS XIII) (in submission), 2013.

[MP97] Donald Mackenzie, Garrel Pottinger, Mathematics, Technology, and Trust: Formal Verification, Computer Security, and the U.S. Military, IEEE Annals of the History of Computing, vol. 19, no. 3, p. 41-59, 1997.

[MPS+12] Kevin Müller, Michael Paulitsch, Reinhard Schwarz, Sergey Tverdyshev, Holger Blasum, MILS-Based Information Flow Control in the Avionic Domain: A Case Study on Compositional Architecture and Verification, Digital Avionics Systems Conference (DASC) proceedings, 2012, IEEE.

[MPT+12] Kevin Müller, Michael Paulitsch, Sergey Tverdyshev, Holger Blasum, MILS-Related Information Flow Control in the Avionic Domain: A View on Security-Enhancing Software Architectures, Workshop on Open Resilient human-aware Cyber-physical Systems (WORCS 2012), 2012, IEEE, http://dx.doi.org/10.1109/DSNW.2012.6264665.

[MWTG00] W. Martin, P. White, F. Taylor, A. Goldberg, Formal Construction of the Mathematically Analyzed Separation Kernel, Proc 15th International Conference on Automated Software Engineering, p. 131-141, 2000.

[NG12] Nils Agne Nordbotten, Tor Gjertsen, Towards a certifiable MILS based workstation, 2012, Norwegian Defence Research Establishment (FFI), http://www.ffi.no/no/Rapporter/12-00049.pdf.

[NLI06] Thuy D. Nguyen, Timothy E. Levin, Cynthia E. Irvine, High robustness requirements in a Common Criteria protection profile, Innovative Architecture for Future Generation High-Performance Processors and Systems, International Workshop on, p. 66-78, 2006, IEEE Computer Society, Los Alamitos, CA, USA, http://calhoun.nps.edu/public/handle/10945/7141.

[NP12] Jan Nowotsch, Michael Paulitsch, "Leveraging Multi-Core Computing Architectures in Avionics," *European Dependable Computing Conference (EDCC)*, 2012.

[PG74] Gerald J. Popek, Robert P. Goldberg, Formal Requirements for Virtualizable Third Generation Architectures, Comm. ACM, vol. 17, p. 412-421, July, 1974.

[Pri92] P.J. Prisaznuk, Integrated Modular Avionics, National Aerospace and Electronics Conference (NAECON), p. 39-45, 1992.

[RAV07] Jeffrey Choi Robinson and Jim Alves-Foss, A High Assurance MLS File Server, 2007.

[RD07] John Rushby, Rance DeLong, MILS Integration Protection Profile, 2007, http://www.csl.sri.com/users/rushby/slides/mipp-jan07.pdf.

[RHN+07] Jeffrey Choi Robinson, W. Scott Harrison, Nadine Hanebutte, Paul Oman, and Jim Alves-Foss, Implementing Middleware for Content Filtering and Information Flow Control, CSAW '07, 2007.

[RI00] John Scott Robin, Cynthia E. Irvine, Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, Proceedings of the 9th USENIX Security Symposium, 2000, www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA423654.

[RKG07] R. Ramaker, W. Krug, W. Phebus, Application of a Civil Intergrated Modular Architecture to Military Transport Aircraft, Digital Avionics Systems Conference (DASC), 2007, p. 2.A.4-1 to 2.A.4-10, 2007.

[Rom08] George Romanski, Management of Configuration Data in an IMA System, Digital Avionics Systems Conference (DASC), p. 1.B.5-1 - 1.B.5-10, 2008, IEEE.

[Rus81] John Rushby, Design and verification of secure systems, Eighth ACM Symposium on Operating System Principles, p. 12-21, 1981, http://www.sdl.sri.com/papers/sosp81/sosp81.pdf.

[Rus01] John Rushby, Formal Verification of McMillan's Compositional Assume-Guarantee Rule, 2001, SRI International, http://ftp.csl.sri.com/users/rushby/papers/mcmillan.pdf.

[Rus08a] John Rushby, Separation and Integration in MILS (The MILS Constitution), SRI-CSL-08-XX, February, 2008, SRI International, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.140.9324.

[Rus08b] John Rushby, A Formal Model for MILS Integration, no. SRI-CSL-08-XX, May, 2008, SRI International, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.9005.

[SBY+13] Deian Stefan, Pablo Buiras, Edward Z Yang, Amit Levy David Terei, Alejandro Russoa, "Eliminating Cache-Based Timing Attacks with Instruction-Based Scheduling," in *Proc. of the 18th European Symposium on Research in Computer Security (ESORICS) 2013*, 2013, p. 718-735.

[SG95] Mary Shaw, David Garlan, Formulations and formalisms in software architecture, Computer Science Today, p. 307-323, 1995, Springer, http://www-2.cs.cmu.edu/~Compose/ProgCodif.pdf.

[Shi06] Kanna Shimizu, «The Cell Broadband Engine processor security architecture, Hardware solutions to problems insoluble in software», http://www.ibm.com/developerworks/power/library/pa-cellsecurity/, IBM DeveloperWorks®, April 2006.

[SKPP] Information Assurance Directorate, U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness. Version 1.03, June, 2007, http://www.niap-ccevs.org/cc-scheme/pp/pp_skpp_hr_v1.03/.

[SLN+10] F. L. Sang, É. Lacombe, V. Nicomette, and Y. Deswarte, "Exploiting an I/OMMU vulnerability," 5th International Conference on Malicious and Unwanted Software (MALWARE), pp. 7-14, 2010.

[SNAC10] Systems and Network Analysis Center / Information Assurance Directorate, Separation Kernels on Commodity Workstations, March, 2010, http://www.niap-ccevs.org/announcements/Separation%20Kernels%20on%20Commodity%20Workstations.pdf.

[SPL95] Olin Sibert, Phillip A. Porras, Robert Lindell, The Intel 80x86 Process Architecture: Pitfalls for Secure Systems, Security and Privacy, Proceedings, 1995 IEEE Symposium on, p. 211-222, 1995.

[SRIOV] PCI-SIG. Single Root I/O Virtualization and Sharing Specification - Revision 1.01. Technical report, 2010.

[SS75] Jerome H. Saltzer, Michael D. Schroeder, The Protection of Information in Computer Systems, Proceedings of the IEEE, vol. 63, no. 9, p. 1278-1308, 1975, http://web.mit.edu/Saltzer/www/publications/protection/, \urlhttp://www.cs.virginia.edu/~evans/cs551/saltzer/.

[Ste91] Daniel F. Sterne, On the Buzzword `Security Policy', IEEE Computer Society Symposium on Research in Security and Privacy, p. 219-230, 1991.

[SV10] F. L. Sang and V. Nicomette, "Attaques DMA peer-to-peer et contremesures," in In Proc. of Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2011), 2011, pp. 147-174.

[Tan07] Andrew S. Tanenbaum, Modern Operating Systems, 3rd edition, 2007, Prentice Hall, Upper Saddle River, NJ, USA.

[TBF13] Sergey Tverdyshev, Holger Blasum, Igor Furgel, Compositional Assurance: EURO-MILS ST/PP for Separation Kernel Based Virtualization, ICCC 2013, http://www.fbcinc.com/e/iccc/day2.aspx.

[TCG11] Trusted Computing Group™, «TPM Main Specification», http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 2011.

[Til+13] Axel Tillequin, and others, "Project Requirements: Classification, Cross-domain analysis and High-Level Architecture", EURO-MILS Deliverable D11.1

[Tri12] Benoît Triquet, "Mixed Criticality in Avionics", Airbus, in Workshop on Mixed Criticality Systems, European Commission, February, 2012, http://cordis.europa.eu/fp7/ict/embedded-systems-engineering/presentations/triquet.pdf.

[Uch05] Gordon Uchenik, Protection Profile for Partitioning Communications Systems in Environments Requiring High Robustness, V0.85 (available on request from Objective Interface Systems).

[Uch07] Gordon Uchenik, Partitioning Communications System for Safe and Secure Distributed Systems, Digital Avioncis Systems Conference (DASC), p. 2.E.5-1 - 2.E.5-8, 2007.

[UV05] Gordon M. Uchenik, W. Mark Vanfleet, Multiple independent levels of safety and security: high assurance architecture for MSLS/MLS, Military Communications Conference, 2005. MILCOM 2005. IEEE, p. 610-614, 2005.

[Wik13] Wikipedia, Comparison of platform virtual machines - Wikipedia, The Free Encyclopedia, 2013, http://en.wikipedia.org/w/index.php?title=Comparison_of_platform_virtual_machines&oldid=567335721 [Online; accessed 15-August-2013].

[Win13] Wind River, Wind River VxWorks MILS Platform 3.0, 2013, http://www.windriver.com/products/platforms/vxworks-mils/MILS-3_PN.pdf.

[Wis11] SKPP Sunset Q & A, 2011, http://www.niap-ccevs.org/announcements/SKPP%2520Sunset%2520Q%26A.pdf.

[WM12] Carl Waldspurger and Mendel Rosenblum. 2012. I/O virtualization. Commun. ACM 55, 1 (January 2012), 66-73. DOI=10.1145/2063176.2063194 http://doi.acm.org/10.1145/2063176.2063194

[WOM02] Mike Weller, Roger Odell, Lee MacLaren, Partitioning Kernel Protection Profile Report, 2002, http://web.archive.org/web/20031209153634/http://www.omg.org/docs/security/02-11-07.doc .

[WP08] Alex Wilson, Thierry Preyssler, Incremental Certification and Integrated Modular Avionics, Digital Avionics Systems Conference (DASC), p. 1.E.3-1 - 1.E.3-8, 2008, IEEE.

[WR08] P. Willmann, S. Rixner, and A. L. Cox, "Protection Strategies for Direct Access to Virtualized I/O Devices," in 2008 USENIX Annual Technical Conference, 2008, pp. 15-28.

[WR11] R. Wojtczuk and J. Rutkowska, "Following the White Rabbit: Software Attacks Against Intel VT-d Technology," 2011.

[YF13]Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," 2013, pp. 1-9.

[YYW08] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman, "IBM Research Report - Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines," IBM, 2008.

[ZAF06] Jie Zhou, Jim Alves-Foss, Architecture-Based Refinements for Secure Computer Systems Design, Proc. Policy, Security and Trust, November, 2006.

[ZAF08] Jie Zhou, Jim-Alves Foss, Security policy refinement and enforcement for the design of multi-level secure systems, Journal of Computer Security, vol. 16, p. 107-131, 2008, IOS Press.

[ZSP+12] Yinping Zhou, Yulong Shen, Qingqi Pei, Xining Cui, Yahui Li, Security Information Flow Control Model and Method in MILS, 2012 Eighth International Conference on Computational Intelligence and Security