SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–19
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Design, implementation and verification of MILS systems

J. Delange*, L. Pautet and F. Kordon

J. Delange, TELECOM ParisTech - LTCI UMR 5141, 46, rue Barrault, F-75634 Paris CEDEX 13, France

SUMMARY

Safety-critical systems are used in many domains (military, avionics, aerospace, etc.) and handle critical data in hostile environements. To prevent data access by unauthorized subjects, they must protect and isolate information so that only allowed entities can read or write information.
However, due to their increased number of functionalities, safety-critical systems design becomes more complex ; this increases difficulties in the design and the verification of security functions and potentially error in their implementation.
The MILS approach introduces rules and guidelines for the design of secure systems. It isolates data according to their security levels, reducing system complexity to ease development. However, there is no approach addressing the whole development of MILS systems from high-level specification (application components with their security levels) to the final implementation (code that execute application functions and provide security mechanisms).
This paper presents a complete development approach for the design, verification and implementation of MILS architectures. It aims at providing a complete framework to build secure applications based on MILS guidelines. We describe security concerns using a modeling language, verify security requirements and automatically implement the system code generation techniques and a MILS-compliant operating system that provides security functions. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: MILS, AADL, POK, code generation, Ocarina

1. INTRODUCTION

1.1. Context

Safety-critical systems are used in many domains such as military, avionics or medicine. They perform critical functions and contain classified data so that they must be secure and reliable. As they operate in hostile environment, they must prevent data theft and perform only authorized actions.

Usually, safety-critical systems that operate in hostile environments are carefully verified using code analysis/review and/or formal verification. It ensures absence of data leakage and improves confidence in systems functions. However, as requirements grow by the time, verification becomes more complex, tedious and costly so that production costs increase significantly to maintain confidence level.

Twenty years ago, the Multiple Independent Levels of Security (MILS) approach [1] was defined to ease the design of secure systems and address this complexity issue. The main idea is to divide and isolate system components according to their security levels to prevent unexpected interferences. Thus, components are analyzed independently, easing their verification and reducing development costs.

*Correspondence to: Journals Production Department, John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK.

MILS classifies components according to their isolation level and defines analysis rules to detect potential data leakage (for example, deny communication between components that do not share the same security levels). To enforce security at execution time, MILS relies on a secure operating system that isolates components and their communications.

1.2. Problem statement

Over the years, the approach was refined [2] and several efforts were made to design MILS systems. However, these improvements focus on specifications analysis [3, 4] and runtime standardization [5] and do not address the development problem from specifications to the final implementation.

This is a major issue since security issues must be analyzed at each step of the development process. Such a process would verify MILS requirements at a specification-level (as defined in [2]) and map them on a MILS operating system that provides isolation services.

1.3. Solution overview

We propose a unified development process for the modeling, verification and implementation of MILS systems using a backbone modeling language. This process, illustrated in figure 1, is supported by a common modeling framework and focuses on three main steps: **modeling**, **validation** and **implementation**.

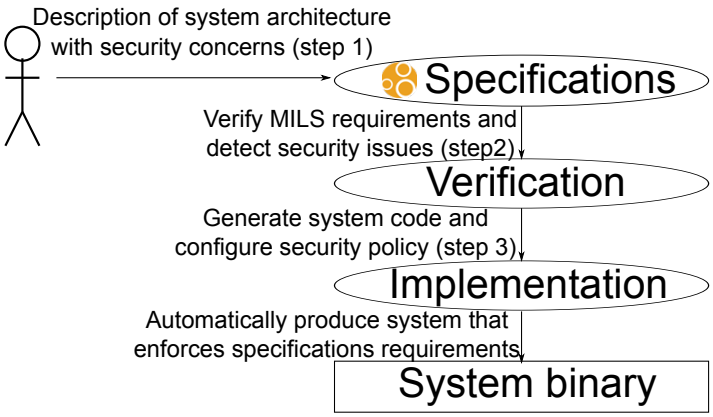

Figure 1. Development process for MILS systems development

System modeling (step 1) provides guidelines to specify system architectures with their security properties and requirements (such as security levels, communication channels, etc.). To do so, we need a modeling language with an appropriate abstraction level to specify security concerns as well as a semantics suitable for system verification and models processing. Our approach uses the Architecture Analysis and Design Language (AADL) [6] because this language fulfills our requirements (our motivations are detailed in 3.1).

Verification (step 2) processes models and enforces MILS requirements and potential security leakage. It inspects system architecture as a whole, analyzes each component, looks for security issues and reports them to system designers so that errors can be found very early in the development process. The National Institute of Standards and Technology (NIST) reports that 70% of errors are introduced at design-level [7].

Implementation (step 3) automatically processes verified specifications to generate executable code. It generates runtime code to execute application-level functions as well as security-related code, such as cipher functions that encrypt/decrypt data.

Then, it integrates produced code with a MILS operating system that isolates components according to their security levels. For that purpose, we design our own MILS operating system, POK [8].

DOI: 10.1002/spe

1.4. Outline

Our paper is organized through 7 main sections. Section 2 gives an overview of the MILS approach. Section 3 presents our modeling guidelines for MILS architectures specification while section 4 details their associated validation rules. Section 5 describes our code generation process and presents our MILS runtime, POK. Finally, section 6 illustrates our approach through a case-study that defines a distributed architecture with several nodes of different security levels.

2. MILS OVERVIEW

The MILS approach is composed of two main parts:

1. Several specification rules for the design of secure systems by isolating security levels. Then, it describes security concerns in a specific formalism and details how security levels are separated.
2. A dedicated operating system (also called MILS runtime) that provides security isolation services to support the established design of specifications produced in the previous step.

Following sections introduce the MILS approach and details each part, explaining its design guidelines as well as its runtime services. In addition, we highlight some security concerns related to the MILS runtime (implementation of device drivers) and explain how they can be adressed.

2.1. Security isolation concepts

MILS isolates security levels as much as possible and limits security levels upgrade/downgrade (writing a data at a given security level in a data classified at a different security level) to avoid covert channels (operations that may break the security policy) and potential security leak.

Security levels isolation eases validation since we can consider each security level independently to focus on critical components.

In the security terminology, an *object* is a data classified at a given security level (e.g. top-secret) whereas a *subject* performs operations (read/write/execute) on *objects*.

A *subject* potentially manipulates several *objects* having various security levels. Such a subject must be verified: they can downgrade information from high to lower security levels. A *"safe" subject* means that it enforces data flows separation according to their security levels.

To distinguish components, MILS classifies them (*subjects*) into three categories according to the isolation level they provide on *objects*:

1. **Single Level of Security (SLS)** components contain *objects* at one security levels.
2. **Multiple Level of Security (MLS)** components contain *objects* classified at multiple security levels without isolation. For example, an MLS component can be a device driver that downgrade a data from a high security-level (top-secret) to a lower security-level (unclassified).
3. **Multiple Single Level of Security (MSLS)** components handle *objects* at different security levels and enforce data flow isolation so that it does not downgrade/upgrade data.

Figure 2 presents an exemple of MILS architecture. It defines a data flow from two SLS components (C1 and C2) at different security levels (top-secret and secret) to an SLS component at the unclassified level (C5).

First, data are produced from two distinct SLS components (C1 and C2) at different security levels. The receiver component (C3) sends received data through two different channels. As it enforces data flow separation, this component is said to be MSLS. Then, component C4 merges received data at different security levels in one communication channel with the unclassified security level. This component does not enforces security levels isolation and so, is said to be MLS. Finally, the last component (C5) receives data at only one security level (unclassified) and so, is considered as SLS.

DOI: 10.1002/spe

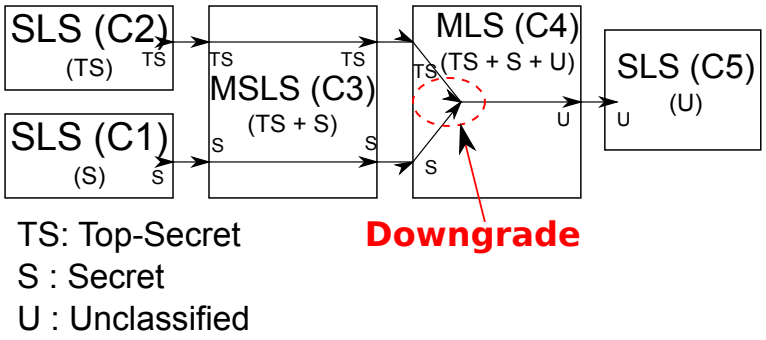

Figure 2. Example of a MILS architecture

To be compliant with the MILS guidelines, component C3 must be verified to check data flow isolation correctness. In addition, if the downgrade operation of C4 is legal (for example, a component that encrypts classified data before sending them into an unclassified network), the downgrade operation must be verified or certified (for example, using formal methods).

To maintain isolation across components, MILS architectures rely on a specific operating system whose characteristics are presented in the next subsection.

2.2. MILS operating system

The main purpose of a MILS operating system is to provide security services to support the design established using the MILS specification rules. To do so, it enforces resources partitioning between components that share different security levels [9]. Each security level is isolated in a partition and each partition is executed as if it runs on a single processor. This partitioning policy prevents potential security leaks and covert channels.

2.2.1. MILS layers

A MILS operating system is divided in two main layers:

1. The **kernel** layer provides core security services. It enforces resources partitioning and maintains isolation across components. This is the most critical part of a MILS architecture as it is responsible to enforce isolation accross partitions (and so, the different security levels).
2. The **partition** layer contains non-critical functions and defines resources and services (runtime library, third-party functions, etc.) required by application components. A partition contains its own environment and is independent from the other partitions running on the same processor.

The figure 3 shows the architecture of a MILS-compliant operating system. It depicts a MILS system containing 3 partitions: two are labelled as top-secret while the last one is unclassified. The MILS kernel, which runs on top of the hardware, enforces time isolation between partitions and executes them in dedicated memory spaces. In the figure, the kernel enforces security levels isolation by connecting only partitions that share the same security level (*top-secret*).

Kernel isolation services

The MILS kernel is composed of few services for partitions isolation. It should be small (in terms of lines of code and complexity) to be amenable to verification/certification (using formal methods, code reviews, etc.). The kernel isolates partitions in space and time:

- **Space isolation** means that each partition has a memory space to store its code and data. So, partitions have an independent memory space other partitions cannot access and no other channel than the one allowed can be established.

DOI: 10.1002/spe

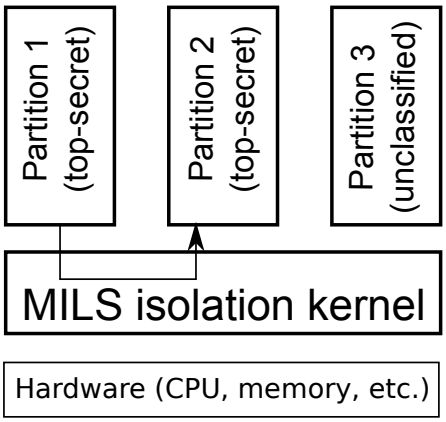

Figure 3. MILS execution platform layers

- **Time isolation** means that each partition is executed during a fixed amount of time. A partition cannot consume more or less time than allowed so that an attacker cannot infer information based on partitions execution time.

In addition, the MILS kernel is responsible to ensure communication isolation across partitions so that it grants only authorized communication channels and avoid creation of covert channels. To do so, it monitors all inter-partitions communications and checks that only channels are established.

In the MILS terminology, these requirements are said to be **NEAT** [4]:

- **N**on-Bypassable: partitions cannot choose to use security functions (i.e: a covert channel).
- **E**valuatable: security functions must be amenable for verification/certification.
- **A**lways-Invoked: security functions are invoked everytime.
- **T**amperproof: security functions as well as data cannot be modified.

2.2.2. Device sharing across partitions

The MILS approach does not define the way device drivers are implemented in a MILS-compliant runtime. Drivers can be implemented either in the kernel or isolated in partitions. However, in context of security separation, it raises two problems:

1. The implementation of drivers in the kernel layer increases its complexity by adding more code. This could impact isolation functions and introduce covert channels as well as security leaks. In addition, as the driver is collocated in the kernel, it gain access to and corrupt partitions data.
2. The sharing of a device between several partitions must enforce data protection so that data at different security levels are not mixed. For example, a network driver used by two partitions at different security levels (e.g. *secret* and *top-secret*) must enforce data separation so that data are not mixed.

Drivers should not be integrated within the kernel: it would add more code in this layer and require more verification (introduction of driver code in the kernel would require to verify its potential impacts on security services) to maintain its level of confidence. To avoid costs associated with such verification activities, drivers must be integrated in independent partitions. It would also keep the kernel as small as possible and amenable to verification/certification.

However, even if device drivers are integrated in partitions, they must enforce separation between security levels. As each partition has its own address space, we cannot ensure this isolation unless the code of the driver is formally verified, as for the kernel layer. For this reason, we cannot provide security isolation within the driver so that partitions hadling classified data must protect them before using the driver (using protection mechanisms such as cipher algorithms).

DOI: 10.1002/spe

To address this issue, we specify devices integration in the specification of the architecture (step 1 in 1.3) and describe the security mechanisms invoked before using any shared device. We also describe security mechanisms used by the partitions that share the device to protect data before sending it to the driver. Then, isolation of security levels is validated (step 2 in 1.3, driver implementation and inter-partitions security mechanisms are automatically generated (step 3 in 1.3).

2.3. Related Work

We distinguish two different kinds of existing work: one about components analysis and classification and one about MILS operating systems. For this reason, we present these works separately.

2.3.1. Architecture analysis

Several approaches were designed about MILS architectures analysis and refinement. Among them, [10] proposes to reduce the amount of components that share different security levels. This significantly reduces verification needs and associated development costs.

Several approaches for the design and verification of MILS architectures use the AADL language [11, 12, 13]. AADL models are evaluated against MILS requirements, validating security isolation enforcement. Other initiatives define modelling language profiles or extensions to specify security requirements within architecture models [14, 15, 16]. Integration of security concerns within models ease system designer work, avoiding use of different formalisms and provide the capability to make system validation before starting implementation efforts.

However, the implementation is not verified against the specification so that there is no proof system execution would enforce properties checked on the model. That is the reason why although model validation is of particular interest, it is necessary to automatically produce the implementation from these validated models.

2.3.2. MILS operating system

A MILS protection profile is currently being written [2] for the Common Criteria [17]. This document describes the services of a MILS-compliant operating system. However, a protection profile for separation kernel [5] exists but this is not specific to MILS.

In addition, several aspects of the MILS runtime is similar to ARINC653 ones. Both isolate applications in partitions and maintain a time and space separation. However, there are two main differences between these two approaches. First, ARINC653 is an established standard for avionics architectures: its API and its services are precisely described, as the functions of MILS-compliant are still described at a high-level (however, the definition of the protection profile [5] would overcome this issue). Then, ARINC653 systems are focused on safety (error detection and recovery) while MILS ones are security-oriented (data protection, security levels isolation).

Finally, all these approaches define services for application isolation, but do not allow easy analysis at a high-level. In particular, we need to abstract security concepts and MILS services for verification purposes. Next sections present our approach that proposes such an abstraction of MILS services and automatically generates secure systems from architectural models.

3. MODELING MILS ARCHITECTURES WITH THE AADL

This section motivates the choice of AADL as a modeling language. It presents the core language and describes our modeling guidelines to specify MILS requirements.

3.1. Motivations for choosing AADL

Several modeling languages exist to represent systems architectures. Popular ones are UML and its associated MARTE profile [18] or its security-specific [19] extensions, SysML [20] or AADL [6].

Our reasons for using AADL are the following:

DOI: 10.1002/spe

• Components and their associated properties are strongly-typed which clarifies architecture concerns.
• Its syntax and semantics are clear and avoid disambiguations. This characteristic is especially important for model analysis support.
• Users can extend the language by adding properties or annexes to the core language.

Despite the advantages of other approaches, the main problem resides in the semantics: many languages use several annexes to refine or extend their semantics. However, introduction of a different notations for similar concepts can lead to inconsistencies so that tools process models using different assumptions, depending on the notation they rely on.

3.2. AADL overview

AADL is a component-centric language for the modeling of software and hardware concerns. It focuses on the definition of block interfaces and separates implementations from their interfaces. The standard proposes both graphical and textual representations.

The AADL standard defines software components (`data`, `thread`, `thread group`, `subprogram`, `process`), execution platform components (`memory`, `bus`, `processor`, `device`, `virtual processor`, `virtual bus`) and hybrid components (`system`).

Components describe elements of the architecture. *Subprograms* model application code. Since it is not an architectural element, it is reduced to a reference to another external piece of code. *Threads* model the active part of an application (such as POSIX threads). *Processes* model address spaces containing *threads*.

Processors model micro-processors and a minimal operating system. *Virtual processors* model a part of a processor. It could be understood in different ways: part of a physical processor, virtual machine, etc.

Memories model hard disks, RAMs. *Buses* model networks, wires. *Virtual buses* are not formally a hardware component, they are bounded to connections in order to describe their requirements. They can be used for several purposes (modeling protocol stacks, security layers, etc.) *Devices* model sensors or actuators.

Systems represent composite components that are made up of hardware components or software components or a combination of the two. For example, a *system* may represent a board with multiple processors and memory chips.

Components hierarchy of an AADL model is composed of several components and sub-components. The topmost component is an AADL system that contains processes, processors and other architecture components.

The interface specification of a component is called its *type* and provides communication functions through *features*. Components communicate by *connecting* their *features* (the *connections* section). Each component describes their internals: sub-components, connections between these sub-components, etc.

AADL allows *properties* to be associated with AADL model elements. Properties are typed and represent name/value pairs that represent characteristics and constraints. Examples are the period and execution time of threads, the implementation language of a subprograms, etc. The standard includes a predeclared set of properties, users can introduce additional properties through property definition declarations. For interested readers, an introduction to the AADL can be found in [21].

3.2.1. Example of an AADL model

A sample AADL model is depicted in figure 4. It represents a producer/consumer architecture: one process (`prs_sender`) executes a thread (`thr_sender`) that produces data (communication ports are represented by arrows). Data is sent to another process (`prs_receive`) and received by a thread (`thr_receiver`) which then consumes the data by executing application-level code (not illustrated on the figure).

Deployment concerns are shown explicitly on this model: connections are bound to buses, process to processors and memories. In this example, readers can notice that both processes are bound the

DOI: 10.1002/spe

same memory component, meaning that they share a common memory address space. This could be potentially a problem from a security point of view if these processes share different security levels.

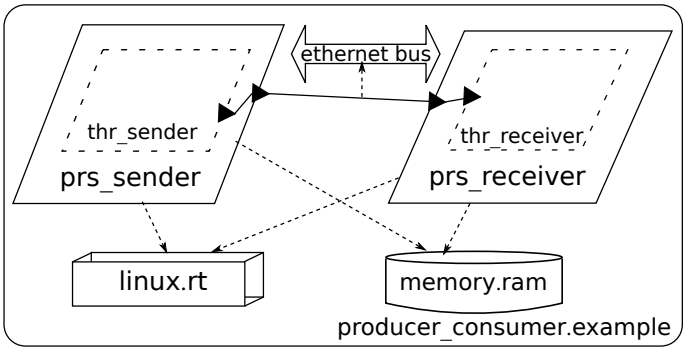

Figure 4. AADL producer/consumer

3.3. *Modeling MILS requirements*

We define an additional AADL property set as well as predefined AADL components (with the namespace `POK` and `poklib`) for MILS modeling. Both are available in the Ocarina [22] toolset.

```
subprogram implementation blowfish_send.i
properties
   Source_Name => "pok_protocols_blowfish_marshall";
end blowfish_send.i;

subprogram implementation blowfish_receive.i
properties
   Source_Name => "pok_protocols_blowfish_unmarshall";
end blowfish_receive.i;

data implementation blowfish_data.i
properties
   Type_Source_Name => "pok_protocols_blowfish_data_t";
end blowfish_data.i;

abstract implementation vbus_blowfish_wrapper.i
subcomponents
   send              : subprogram blowfish_send.i;
   receive           : subprogram blowfish_receive.i;
   marshalling_type  : data blowfish_data.i;
end vbus_blowfish_wrapper.i;

virtual bus implementation topsecret.i
properties
   POK::Security_Level => 3;
   POK::Blowfish_Key => "cipher key";
   Implemented_As => classifier (vbus_blowfish_wrapper.i);
end topsecret.i;
```

Listing 1: Top-secret layer modeling with the AADL

3.3.1. *Security levels modeling*

A MILS security level is described with a `virtual bus` component. It represents both the security-level (using the `POK::Security_Level` property) and its associated mechanisms (for example, cipher algorithms).

Security mechanism implementations are specified in a *"component box"* (an `abstract` component) that contains its required components (data, subprograms, etc.). This abstract

DOI: 10.1002/spe

component is associated with the `virtual bus` component using the `Implemented_As` property and contains all resources required for the implementation (data, subprograms, etc.)

Listing 1 illustrates the modeling of a security layer (called `topsecret`). Property `POK::Blowfish_Key` represents the key used by the cipher algorithm (assuming the use of blowfish) and `Implemented_As` property points to a component that contains all subprograms and data required to implement this cipher mechanism.

Components inheritance and extension allow users to design a hierarchy of security layers. For example, they can define a basic `virtual bus` associated at a given security level with several implementations, each of them providing different security mechanisms to protect the data (different cipher algorithms or cipher keys).

3.3.2. Kernel modeling

A MILS kernel is specified using the `processor` component. It models both hardware and the associated software to isolate partitions. Additional properties (described in later sections) are associated to the component to specify isolation requirements.

```
virtual processor implementation topsecret_runtime.i
properties
   Provided_Virtual_Bus_Class => (classifier (topsecret.i),
                                  classifier (secret.i));
end topsecret_runtime.i;

process topsecret_partition
features
   theport : in data port thetype
{Allowed_Connection_Binding_Class=>(classifier(topsecret.i));};
end topsecret_partition;

process implementation topsecret_partition.i
subcomponents
   athread : thread thr_producer.i;
end topsecret_partition.i;

processor implementation mils_kernel.i
subcomponents
   rt1 : virtual processor topsecret_runtime.i;
   rt2 : virtual processor topsecret_runtime.i;
properties
   POK::Slots => (500ms, 1000ms);
   POK::Slots_Allocation => (reference (topsecret_rt1),
                             reference (topsecret_rt2));
end mils_kernel.i;

memory implementation ram_mapping.i
subcomponents
   seg1 : memory segment.i;
   seg2 : memory segment.i;
end ram_mapping.i;

system implementation mils_system.i
subcomponents
   kernel : processor mils_kernel.i;
   p1     : process topsecret_partition.i;
   p2     : process topsecret_partition.i;
   ram    : memory ram_mapping.i;
properties
Actual_Memory_Binding=>(reference(ram.seg1))       applies to p1;
Actual_Memory_Binding=>(reference(ram.seg2))       applies to p2;
Actual_Processor_Binding=>(reference(kernel.rt1)) applies to p1;
Actual_Processor_Binding=>(reference(kernel.rt2)) applies to p2;
end mils_system.i;
```

Listing 2: Modeling a MILS kernel with two topsecret partitions

DOI: 10.1002/spe

3.3.3. Partitions modeling

A MILS partition is specified using `virtual processor` and `process` components. A `virtual processor` models a partition runtime and its properties (such as the scheduling protocol used to schedule partition tasks). The `process` component models partition contents (`threads`, `data`, etc.) and communication interfaces.

3.3.4. Devices modeling

A device is specified using `device` and `virtual processor` components. As for partitions, the `virtual processor` component models the environment that executes the driver, associated with its resources (functions, libraries, etc.).

Because our approach assumes that devices and their associated drivers are executed as a partition, the `device` component represents a partition that executes the software driver. To model its resources, it is associated with an `abstract` component that contains required resources. These additional components model tasks and data involved in the control of the underlying hardware. Like the regular ones, device partition can communicate with the others using inter-partitions channels. They also rely on this mechanism to transmit to and receive from application components.

This modelling pattern is illustrated in listing 3, which shows the representation of a network device. It defines a device (`rtl8029`) that provides several inter-partitions ports (`incoming_secret`, `incoming_topsecret`, etc.)to interact with other partitions at different security levels. The tasks and resources required by the driver are specified by associating the `driver_container` component to the partition using the `Device_Driver` property. In this implementation, the driver is supported by two tasks (`thread` components) : one (`data_handler`)sends data from the other partitions while another periodically watches for incoming data from the network connection.

```
subprogram init
end init;

process implementation driver.i
subcomponents
   data_handler    : thread thr_handler.i;
   poller          : thread thr_poller.i;
properties
   Required_Memory_Size => 160 Kbyte;
end driver.i;

abstract implementation driver_container.i
subcomponents
   p : process driver.i;
end driver_container.i;

device rtl8029
features
   incoming_topsecret      : in data port types::integer;
   incoming_secret         : in data port types::integer;
   incoming_unclassified   : in data port types::integer;
   outgoing_topsecret      : out data port types::integer;
   outgoing_secret         : out data port types::integer;
   outgoing_unclassified   : out data port types::integer;
   ethernet_access         : requires bus access runtime::ethernet.unsecure;
properties
   Initialize_Entrypoint => classifier (rtl8029::init);
   Device_Name => "rtl8029";
end rtl8029;

device implementation rtl8029.i
properties
   Device_Driver => classifier (rtl8029::driver_container.i);
end rtl8029.i;
```

Listing 3: Modeling a device driver

DOI: 10.1002/spe

3.3.5. *Time isolation modeling*

Time isolation of the MILS kernel is described with the `POK::Slots` (time slots allocated for partitions execution) and the `POK::Slots_Allocation` property (allocation of slots across partitions).

Both properties are defined on the `processor` component (the MILS kernel) and reference its partitions. For this reason, we specify a `virtual processor` (partitions runtime) as a subcomponent of the `processor` (MILS kernel).

The kernel, depicted in listing 2, contains two partitions: the first one fires at 500 ms while the second one fires at 1s. By default, we assume the runtime executes partitions with a cyclic round-robin scheduling protocol.

3.3.6. *Communication channels modeling*

MILS security levels (`virtual bus`) are bound to partitions (`virtual processor`) with the `Provided_Virtual_Bus_Class` property. The security level of each communication interface (`port`) is specified with the `Allowed_Connection_Binding_Class` property. That associates a `port` with a security level (`virtual bus`).

Listing 2 shows the binding of security levels and partitions. It defines partitions that provide top-secret and secret security levels with one incoming data port that communicates at the `topsecret` security level.

3.3.7. *Memory isolation*

MILS memory isolation is specified by binding each partition to one dedicated memory segment. To do so, `process` components are associated to a `memory` component (using the `Actual_Memory_Binding` property).

Listing 2 defines a main memory (`ram_mapping`) divided in two memory segments. Each one is associated with one partition. This one-to-one binding ensures space isolation between partitions.

4. VERIFICATION OF MILS REQUIREMENTS USING AADL MODELS

The AADL model allows the validation of MILS requirements from its specification prior to implementation efforts. To do so, we rely on REAL [23], an AADL-dedicated constraint language. Requirements are expressed through theorems that are validated thanks to a dedicated tool.

4.1. *Security layers conformity*

First, we verify that security layers that having different security levels use distinct protection mechanisms. For that purpose, our theorems browse security layers (`virtual bus` components) and check that they use distinct cipher algorithms and configurations.

4.2. *Security levels usage*

Validation theorems check that two communicating partitions share the same security level (`virtual bus`). This ensures security consistency by checking that sending and receiving interfaces are using the same security mechanisms (same cipher key, etc.). In addition, theorems also enforce that security levels used by these interfaces are provided by partitions.

4.3. *Space isolation*

Theorems check space isolation, ensuring that each AADL `process` (MILS partition) is associated to a single `memory` component. If two `processes` share the same memory component, they must be of the same security levels.

4.4. *Time isolation*

Theorems check that MILS partitions (`virtual processor`) are executed by the kernel at least one time during a scheduling cycle. This ensures that partitions will have time to execute their task. In

Prepared using speauth.cls
DOI: 10.1002/spe

addition, this execution time must be fixed to avoid security threats (some attacks could rely on an analysis of partitions or tasks execution time).

To do so, our validation theorem inspects `processor` components and verify that, for each partition contained in that processor, a scheduling slot is allocated. As a result, it ensures that each partition is executed at least one time in each scheduling cycle.

4.5. Other validation related to AADL models

Verification of AADL architectures is a wide topic and several verifications were issued for different purposes. By using AADL as modeling language, regular validation tools (such as task scheduling [24] or flow latency [25]) can be issued on MILS systems, checking various system requirements and increasing the reliability of the development process.

4.6. Benefits of AADL models validation

Contrary to the initial MILS formalism which uses its own abstract representation of the system (as in figure 2), AADL models also specify configuration and deployment informations.

In our context, this deployment information indicates the security levels used by each partition, describing which level is isolated from the others. Thanks to this precise description of security levels and information sharing, we are able to make a finer analysis and isolate security concerns according to partitions isolation.

Finally, the use of a standardized modeling language as AADL for the description of MILS architectures provides the ability to use them with a wide-range of tools, from model validation (as described in section 4.5) to code generation (described in next sections).

5. AUTOMATIC IMPLEMENTATION

The implementation process is divided in two main parts, illustrated in figure 5:

1. The **code generation** process (supported by Ocarina [22]) creates partitions and kernel code (illustrated with dashed boxes in figure 5) from validated AADL models.
2. The **MILS operating system** (POK [8]) contains runtime services (solid boxes in figure 5). These services are adapted using the generated code and provide isolation support for partitions execution.

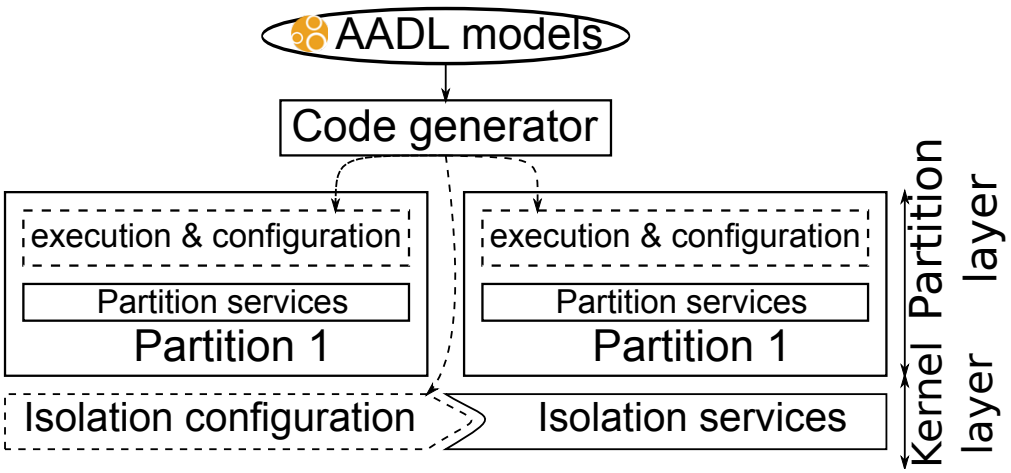

Figure 5. Detail of our implementation process

DOI: 10.1002/spe

5.1. Code generation

5.1.1. Code generation patterns for partitions

The code generator creates code that instantiates/configures partition resources and services. This code is created from AADL components that model partitions (`process`, `threads`, `data`), devices (components associated to the device component using the `Device_Driver` property) and their communication ports (`features`).

For partition or driver resources, generation patterns create tasks, shared data and communication channels. It also generates tasks from AADL `threads` that receive data, execute application function (such as Ada/C code) and send outputs.

From a security-side, the code generator configures cipher algorithms and automatically encrypt/decrypt data before/after sending/receiving it. It ensures that security layers bound to each inter-partition channel (AADL `port`) are used correctly. In addition, it ensure that a classified partition that share a device with another at a lower security level protects its data, ensuring its isolation and avoiding security levels mix.

Finally, this automatic configuration of cipher algorithms ensures data encryption according to the specification. It avoids security threats potentially introduced by developers who can introduce errors in the usage of these security mechanisms. As a result, the automatic configuration ensures that data will be crypted before sharing them over an unsecured bus (such as an ethernet network).

5.1.2. Code generation patterns for kernels

The code generator automatically configures kernel isolation services. This is the most critical part of the code since an error can break the security policy. This is a particular interest since the automatic code generation avoids errors generally introduced by traditional development methods.

Generation patterns analyze AADL components that model isolation requirements (`processors`, `memories` and partitions `features`) and create code that:

- Configure space isolation, ensuring that each memory segment is isolated and allocated to a partition.
- Configure communication channels by connecting partitions interfaces so that the kernel precisely knows which partitions can communicate together.
- Allocate partitions resources (tasks, channels, etc).

Generated code is then integrated to our MILS operating system in order to create the final implementation. Next subsection presents our MILS O/S implementation, POK.

5.2. POK, a libre MILS Operating System

POK is a MILS-compliant operating system functions released under the BSD licence. We detail its architecture (shown in figure 6) through the next paragraphs.

5.2.1. Kernel layer

Our design guidelines lead us to keep it as small as possible for it to be amenable to certification/verification. At this time, its size is less than 3000 Source Line Of Code (SLOC), including the architecture-dependent code (which handles low-level concerns).

It provides time and space isolation services:

- **Space isolation**: it stores partitions code, data and resources in separate address spaces.
- **Time isolation**: it schedules partitions according to a cyclic scheduling protocol with fixed time slices.

The kernel also provides inter-partitions communication service, which is responsible to enable data sharing across partitions according to the system configuration defined by the system designer. This functionality ensures that only allowed partitions can communicate, avoiding covert channels. However, even if POK services avoid the use of undeterministic and non-secure functions, these services would be verified/certified to be used in an industrial context.

5.2.2. Partition runtime and device drivers functions

Due to its lower criticality, this layer contains more services than the kernel. The core

DOI: 10.1002/spe

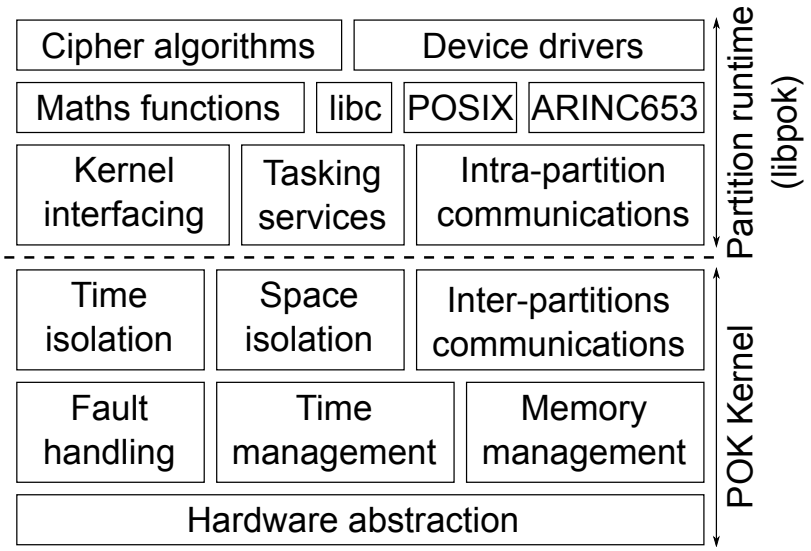

Figure 6. POK architecture

layer (services depicted at the bottom in figure 6) provides tasking management, intra-partition communication functions and data locking mechanisms (semaphore, mutex, etc.). This set of core functions are then reused by several compatibility layers that support API of well-known standards. The partition layer of POK supports POSIX or ARINC653 so that developers can directly reused existing code and link it with the partition runtime librairy.

This layer also includes protection mechanisms used for data encryption. It is based on a port of OpenSSL [26], a reference implementation of several cipher algorithms. These functions are also automatically configured by the generated code to crypt/decrypt data transmitted through inter-partitions ports.

Finally, this layer provides device drivers functionalities to interact with the underlying hardware. These functions are used to perform input/output or to access hardware buses (PCI, USB, etc.) within a partition. These functions are designed so that only one partition can have an access to a specific device, avoiding its use by several partitions.

6. CASE STUDY

6.1. Case study overview

Our case study, illustrated in figure 7, defines a system that share data at top-secret, secret and unclassified security levels. Data classified at these levels are transferred through an unclassified channel, which is typical study when we share data over a network such as ethernet that does not provide any isolation mechanism.

SLS components in S1, S2 and S3 are subprograms that produce integers at a given rate (for example, their associated tasks have a period of 100 ms). SLS components in R1 and R2 print received data.

SLS components S4 and R3 represent software that manages network hardware (device and their associated driver). It is responsible for sending/receiving packets on the network. They are not verified and so, they transport data only at an unclassified security level.

To send classified data at an unclassified security level, we downgrade/upgrade data security-level. It means that classified entities (top-secret, secret) send/receive data to/from unclassified ones. To prevent security leaks, cipher algorithms are used: before sending a classified data to an unclassified entity, encryption is performed, ensuring that receiving task cannot read it. This is also

DOI: 10.1002/spe

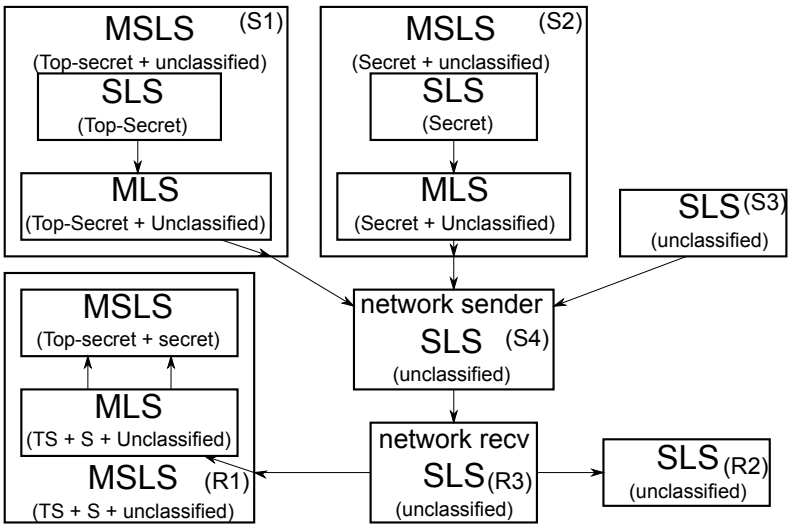

Figure 7. Overview of our case study

done on the other side : when a classified entity receives data from an unclassified one, it decrypts it automatically, recovering the value sent by the original classified entity.

This case-study illustrates the relevance of our model-based approach for the validation and the implementation of security concerns, ensuring security levels configuration consistency and absence of security leaks within a distributed architecture. Nevertheless, the following considerations must be taken into account:

1. Actual implementations of security policies is more complex and often rely on several cipher keys and/or algorithms. However, no matter the implementation, same concepts as our approach apply, ensuring configuration consistency and security leakage avoidance.
2. The code generation step transforms specification into a configuration and runtime code for partitions and their associated isolation kernel (POK). In the context of safety-critical systems, such a code has to be certified. This concerns would typically be addressed either by validating/certifying the (a) generated code or the (b) code generator. However, despite their importance, such topics are out of the scope of this article.

6.2. AADL model

We map this abstract architecture in AADL using our modeling rules (cf. figure 8). The resulting model is a distributed system deployed on two nodes, sharing classified information over an untrusted network:

1. *The sender node* isolates components S1, S2, S3 and S4 in partitions that runs on the same processor
2. *The receiver node* isolates components R1, R2 and R3 in partitions on another processor.

This model defines the security layers by means of `virtual bus` components. It associates appropriate security layers to partitions and communication interfaces: the top-secret security layer is associated to partitions S1 and R1, secret layer to S2 and R1 and the unclassified layer to S3 and R2. When a partition or a communication port is not associated to a security layer, it is bound to the unclassified security layer by default (no encryption mechanism is used).

As S1, S2, S3, R1 and R2 partitions contain application code, we model them using `processes` that contain their resources (`thread`, `data`, etc.). On the other hand, S4 and R3 represent devices that send/receive data through the network. Thus, we model them using a `device`

DOI: 10.1002/spe

component.Nonetheless, these components also model a partition, except that they have an access to the hardware they control (the network device).

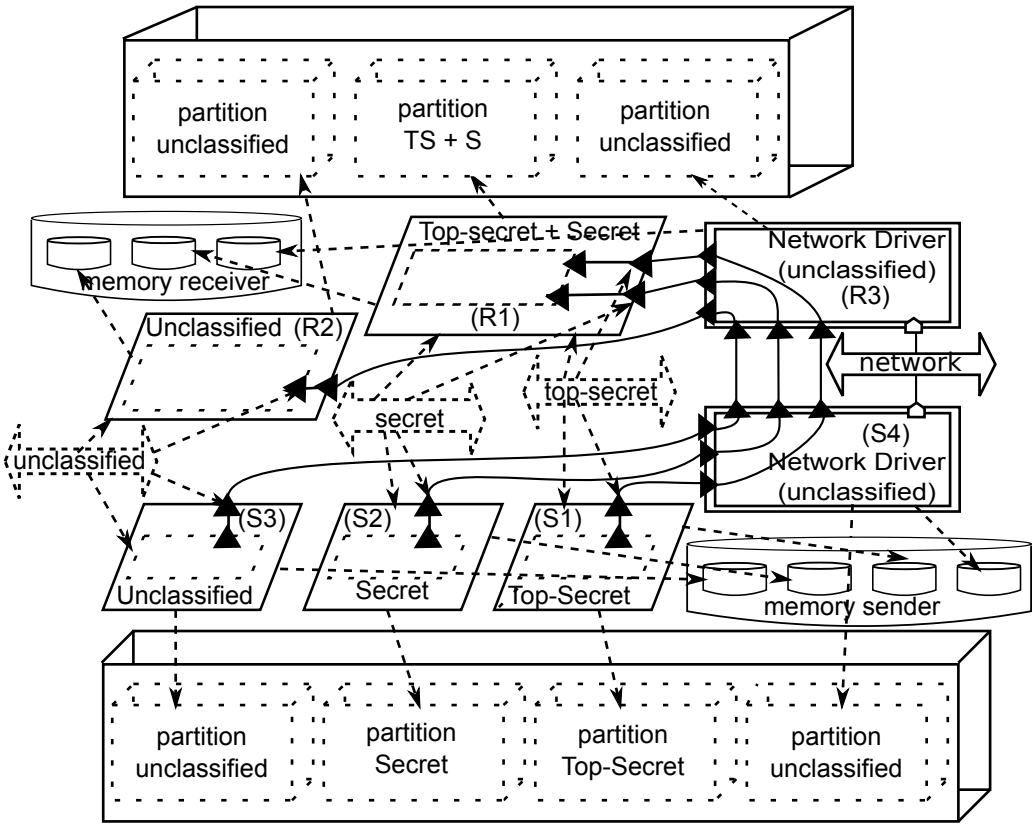

Figure 8. AADL architecture

6.3. System validation from AADL models

We first validate our architecture against our REAL theorems (cf. section 4). The following theorems were experienced:

1. **Structural analysis**: the division of the system into partitions, the use of appropriate components and properties (`virtual processor` and `process` components for partitions modelling, `processor` components for the specification of partitioned kernels, etc.). As illustrated in figure 8, our AADL model meets our modelling pattern so that this theorem is validated.
2. **Space isolation**: memory of each node is divided into `memory` subcomponents that represent memory segments of each partition. Then, theorems verify that each partition is bound to a memory segment for the storage of its data and code, ensuring their isolation from one to another. In our case study, the memory of the sender node is divided into four segments and the one of the receiver is composed of three of them. Then, each partition is associated with a distinct segment.
3. **Time isolation**: each partitioned kernel (`processor` component) must ensure partitions execution. This is achieved by inspecting its properties and verifying that each partition is referenced in the scheduling table. In our example (textual representation, available in POKexamples), each partition is referenced exactly one time in the scheduling table so that this theorem is validated.
4. **Security mechanisms usage**: inter-partitions ports use security mechanisms provided by their containing partitions. In our case-study, each inter-partitions ports specify the security

DOI: 10.1002/spe

mechanisms they use through an association with a `virtual bus` component. In addition, as ports use the same `virtual bus` as their containing partition (a port associated with the `secret virtual bus` is contained in a partition labelled with the `secret virtual bus`) so that this theorem is validated.

5. **Security mechanisms configuration** : each security layer (`virtual bus` component) must be configured with appropriate properties. In our model, the `topsecret` and `secret` security layers specify dedicated properties to define the cipher algorithm they use with its configuration requirements (shared key, initialization data, etc.) so that this theorem is validated.

This first validation step, established at a model-level, ensures that MILS requirements are met in the design. Then, the automatic code generation transforms validated design into executable code. Next sections detail these aspects and explain how we verify the use of security mechanisms.

6.4. Code generation and performance assessment

Ocarina [22] automatically generates code for nodes from this architecture model. This code is then integrated with our MILS-compliant operating system, POK [8], to produce a final implementation. As memory footprint is a major concern for safety-critical applications, we measure the size of each partition to assess the compliance of our approach against this requirements.

Table I reports the size of each component. To get these results, we compile each component separately and remove debugging and useless symbols in the binaries (using the `strip` program). We separate the size of the application (the functional part of the application) from the size of the architecture-related code (functions related to non-functional aspects such as resources creation, tasks management, etc.).

Size of both kernel is similar (19 KB) and can also be considered as *low*. Readers can notice that partitions classified at *secret* or *topsecret* layers have higher memory foorprint than *unclassified* ones. This memory overhead comes from the inclusion of cipher algorithms that add code in the architecture layer. Indeed, application layer size is almost equal in all partitions.

	Component	**Size (total)**	**Architecture code**	**Application code**
Sender	Kernel	19 KBytes	N/A	N/A
	Network driver	53 KBytes	N/A	N/A
	Partition *top-secret*	22 KBytes	21 KBytes	888 Bytes
	Partition *secret*	26 KBytes	25 KBytes	888 Bytes
	Partition *unclassified*	14 KBytes	13 KBytes	888 Bytes
Receiver	Kernel	19 KBytes	N/A	N/A
	Network driver	53 KBytes	N/A	N/A
	Partition *top-secret* & *secret*	35 KBytes	34 KBytes	800 Bytes
	Partition *unclassified*	14 KBytes	13 KBytes	800 Bytes

Table I. Memory footprint of kernels and partitions

6.5. Verification of security mechanisms

We check the correctness of the security mechanisms in the produced system. To do so, a virtual machine (QEMU [27]) executes generated applications and connects them through a virtual ethernet bus. This network can be monitored so that we can see the data that it transports by capturing the traffic between these virtual machines. Figure 9 depicts this deployment strategy, showing two instances of the virtual machine (one for the sender node, another for the receiving node) as well as the virtual network layer.

We capture the network traffic produced by each partition using the *wireshark* [28] tool (also shown in figure 9). It reports data exchanged across the nodes of the distributed system, showing their content and so, the potential use of cipher algorithms. Our experiments show that three kinds

DOI: 10.1002/spe

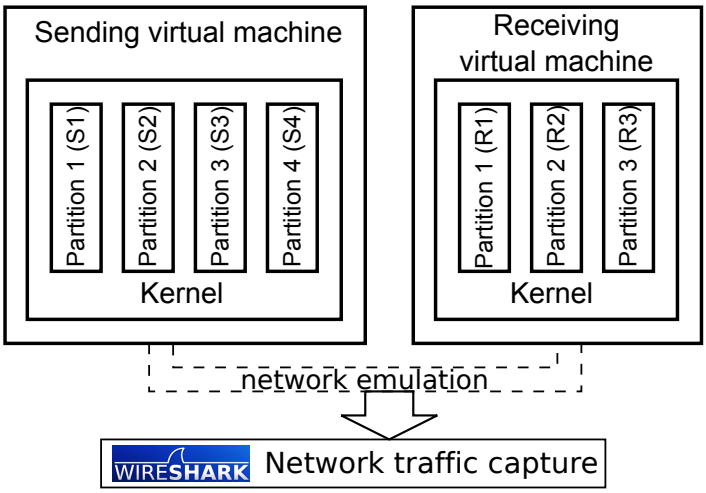

Figure 9. Deployment of our case-study over two virtual machines

of data are being transported: one that is readable in clear, one that needs the configuration of the *topsecret* layer cipher algorithm and another that requires the protection mechanisms of the *secret* layer. Also, we were able to read protected data using the cipher algorithms of each security layer with their appropriate configuration, showing that security requirements are well configured and used.

7. CONCLUSION

This article presented a framework for the design, validation and implementation of MILS architectures. As far as we know, this is the first rapid prototyping approach of MILS systems.

Our modeling patterns dedicated to MILS and their associated validation rules detect security issues at a specification level. This is of particular interest because it ensures security levels isolation in a distributed architecture and check for their implementation correctness. Specification and validation steps report errors before implementation, thus increasing reliability of produced systems.

In addition, automatic code generation (with the Ocarina code generator and the POK MILS O/S) ensures a conformant implementation. This avoids errors related to hand-made code.

7.1. Further work

This work could also be improved in several ways. In particular, if we want to extend our validation framework and validate architecture models using other security policies.

In addition, we plan to evaluate encryption mechanisms with other security mechanisms. During our current experiments, we only use symmetric cipher algorithms but it can be extended with asymmetric ones.

ACKNOWLEDGMENTS

POK is an open-source partitioned kernel developped by many actors from different schools and companies. We would like to thank every contributor of POK.

REFERENCES

1. John Rushby. The Design and Verification of Secure Systems. *Eighth ACM Symposium on Operating System Principles (SOSP)*, Asilomar, 1981; 12–21. (ACM *Operating Systems Review*, Vol. 15, No. 5).
2. John Rushby. Separation and Integration in MILS (The MILS Constitution). *Technical Report*, SRI International 2008.

DOI: 10.1002/spe

3. WW Technology Group. EDICT Tool Suite - http://www.wwtechnology.com/.
4. Uchenick G, Vanfleet M. Multiple Independent Levels of Safety and Security: High Assurance Architecture for MSLS/MLS. *Military Communications Conference, 2005. MILCOM*, IEEE (ed.), 2005.
5. Information Assurance Directorate. U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness 2007.
6. SAE. *Architecture Analysis & Design Language v2.0 (AS5506)* September 2008.
7. National Institute of Standards and Technology (NIST). The Economic Impacts of Inadequate Infrastructure for Software Testing - http://www.nist.gov/director/prog-ofc/report02-3.pdf. *Technical Report* 2002.
8. Delange J. *POK user guide - http://pok.gunnm.org*.
9. Vanfleet WM, Luke JA, Beckwith RW, Taylor C, Calloni B, Uchenick G. MILS:Architecture for High-Assurance Embedded Computing - http://www.stsc.hill.af.mil/crosstalk/2005/08/0508Vanfleet_etal.html. *Crosstalk* 2005; .
10. Zhou J, Alves-Foss J. Architecture-based refinements for secure computer systems design. *PST '06: Proceedings of the 2006 International Conference on Privacy, Security and Trust*, ACM: New York, NY, USA, 2006; 1–11, doi:http://doi.acm.org/10.1145/1501434.1501453.
11. Hansson J, Feiler PH, Morley J. Building Secure Systems Using Model-Based Engineering and Architectural Models - http://www.stsc.hill.af.mil/crosstalk/2008/09/0809HanssonFeilerMorley.html. *Crosstalk* September 2008; .
12. Hansson J, Greenhouse A. Modeling and Validating Security and Confidentiality in System Architectures. *Technical Report*, CMU/SEI 2008.
13. Hansson J, Wrage L, Feiler PH, Morley J, Lewis B, Hugues J. Architectural modeling to verify security and nonfunctional behavior. *IEEE Security and Privacy* 2010; **8**:43–49, doi:http://doi.ieeecomputersociety.org/10.1109/MSP.2009.143.
14. Lodderstedt T, Basin D, Doser J. Secureuml: A uml-based modeling language for model-driven security. *"UML» 2002 - The Unified Modeling Language*, *Lecture Notes in Computer Science*, vol. 2460, Jézéquel JM, Hussmann H, Cook S (eds.). Springer Berlin / Heidelberg, 2002; 426–441.
15. Jürjens J. Umlsec: Extending uml for secure systems development. *«UML» 2002 - The Unified Modeling Language*, *Lecture Notes in Computer Science*, vol. 2460, Jézéquel JM, Hussmann H, Cook S (eds.). Springer Berlin / Heidelberg, 2002; 1–9.
16. Rodríguez A, Fernández-Medina E, Piattini M. Towards a uml 2.0 extension for the modeling of security requirements in business processes. *Trust and Privacy in Digital Business*, *Lecture Notes in Computer Science*, vol. 4083, Fischer-Hübner S, Furnell S, Lambrinoudakis C (eds.). Springer Berlin / Heidelberg, 2006; 51–61.
17. Common Criteria for Security Evaluation – http://www.commoncriteriaportal.org.
18. OMG. *A UML Profile for MARTE, Beta 1*. OMG Document Number: ptc/07-08-04, 2007.
19. Rodriguez A, Fernandez-Medina E, Piattini M. Security requirement with a uml 2.0 profile. *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security*, IEEE Computer Society: Washington, DC, USA, 2006; 670–677, doi:http://dx.doi.org/10.1109/ARES.2006.125.
20. Object Management Group (OMG). Systems Modeling Language (SysML) 2007.
21. Peter H Feiler, David P Gluch, John J Hudak. The Architecture Analysis and Design Language (AADL): An Introduction. *Technical Report* 02 2006.
22. Zalila B, Hugues J, Pautet L. *Ocarina user guide*. TELECOM ParisTech.
23. Olivier Gilles, Jérôme Hugues. Validating Requirements at Model-Level. *Ingénierie Dirigée par les modèles (IDM'08)*, Mulhouse, France, 2008; 35–49.
24. Julien Delange, Laurent Pautet, Alain Plantec, Mickael Kerboeuf, Frank Singhoff, Fabrice Kordon. Validate, simulate, and implement ARINC653 systems using the AADL. *Ada Letter* 2009; **29**(3):31–44, doi:http://doi.acm.org/10.1145/1653616.1647435.
25. Peter H Feiler, Jorgen Hansson. Flow Latency Analysis with the Architecture Analysis and Design Language (AADL) – http://www.sei.cmu.edu/publications/documents/07.reports/07tn010.html. *Technical Report*, SEI 2007.
26. The OpenSSL Project. OpenSSL - http://www.openssl.org. Programa de computador December 1998. URL http://www.openssl.org/.
27. Bellard F. Qemu, a fast and portable dynamic translator. *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, USENIX Association: Berkeley, CA, USA, 2005; 41–41.
28. Angela Orebaugh, Gilbert Ramirez, Josh Burke, Larry Pesce. *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*. Syngress Publishing, 2006.