Design, Verification and Implementation of MILS systems

Julien Delange, Laurent Pautet
TELECOM ParisTech - LTCI UMR 5141
46, rue Barrault
F-75634 Paris CEDEX 13, France
delange@enst.fr, pautet@enst.fr

Fabrice Kordon
LIP6, Univ. P & M. Curie
4 place Jussieu
75252 Paris Cedex 05, France
fabrice.kordon@lip6.fr

***Abstract*—Safety-critical systems are used in many domains (military, avionics, aerospace, etc.) and handle critical data in hostile environements. These systems must protect data 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.

The Multiple Independent Levels of Security (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 to the final implementation.

This paper presents our approach for the design of MILS architectures. We describe security concerns using a modeling language, verify security requirements and automatically implement the system using code generation.

I. Introduction

a) 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 [15] 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.

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.

b) Problem statement: Over the years, the approach was refined [16] and several efforts were made to design MILS systems. However, these improvements focus on specifications analysis [20], [18] and runtime standardization [9] 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 [16]) and map them on a MILS operating system that provides isolation services.

c) 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**.

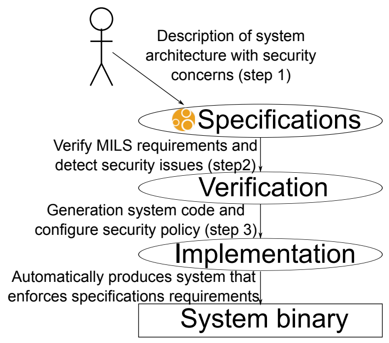
Description of system architecture with security concerns (step 1)
Specifications
Verify MILS requirements and detect security issues (step2)
Verification
Generation system code and configure security policy (step 3)
Implementation
Automatically produces system that enforces specifications requirements
System binary

Fig. 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) [17] because this language fulfills our requirements (our motivations are detailed in III-A).

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 [10].

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 [2].

d) Outline: Our paper is organized through 7 main sections. Section II gives an overview of the MILS approach. Section III presents our modeling guidelines for MILS architectures specification while section IV details their associated validation rules. Section V describes our code generation process and presents our MILS runtime, POK. Finally, section VI illustrates our approach through a case-study that defines a distributed architecture with several nodes of different security levels.

II. MILS OVERVIEW

This section introduces MILS concepts. First, we explain the main principles and then, detail services required by a MILS operating system to isolate security levels.

A. Security isolation concepts

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

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.

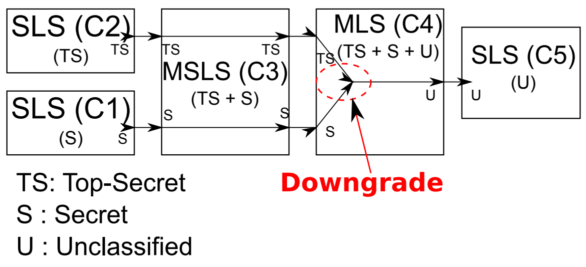

Fig. 2. Example of a MILS architecture

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.

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.

B. MILS operating system

The main purpose of a MILS operating system is to enforce resources partitioning between components that share different security levels [19]. This partitioning policy prevents from potential security leaks and covert channels.

e) MILS layers: A MILS system is divided in two main layers:

1) The **kernel** layer enforces resources partitioning and maintains isolation across components. This is the most critical part of a MILS architecture as it contains isolation services.

2) The **partition** layer contains resources and services (runtime library, third-party functions, etc.) required by components. A partition is isolated by the MILS kernel and contains one or several components.

The figure 3 illustrates this architecture. 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*).

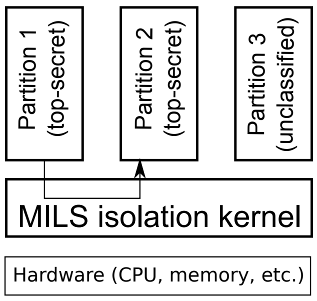

Fig. 3. MILS execution platform layers

f) Kernel isolation services: The MILS kernel isolates partitions in space and time. Space isolation means that each partition has a memory space to store data and that communications between partitions are monitored and explicitly granted. So, partitions have an independent memory space other partitions cannot access and no other channel than the one allowed can be established.

Time isolation means that each partition are 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 the MILS terminology, these requirements are said to be **NEAT** [18]:

- **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 every-time.
- **T**amperproof: security functions as well as data cannot be modified.

The kernel layer contains only few services and does not contain device drivers as in traditional operating systems. This would introduce additional code that may break isolation services. Instead, devices are handled by one or several partitions, through a direct access to the hardware. By doing so, kernel remains small and prone to verification.

C. 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.

g) Architecture analysis: Several approaches were designed about MILS architectures analysis and refinement. Among them, [22] 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 modeling languages such as AADL [7], [8]. AADL models are evaluated against MILS requirements, validating security isolation enforcement. This integrates MILS in existing modeling languages and eases designer work, avoiding use of different formalisms.

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.

h) MILS operating system: A MILS protection profile is currently being written [16] for the Common Criteria [1]. This document describes the services of a MILS-compliant operating system. However, a protection profile for separation kernel [9] exists but this is not specific to MILS.

These solutions define services for security isolation, but do not ease their analysis. 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 MILS systems from architectural models.

III. 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.

A. Motivations for choosing AADL

Several modeling languages exist to represent systems architectures. Popular ones are UML and its associated MARTE profile [12] or its security-specific [14] extensions, SysML [11] or AADL [17].

Our reasons for using AADL are the following:

• 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, there may be inconsistencies between different annexes so that their use in the same model lead to a semantics leakage.

B. 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 [4].

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`). It is then by the receiver thread to execute application-level code (not illustrated on the figure).

Deployment concerns are shown 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 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.

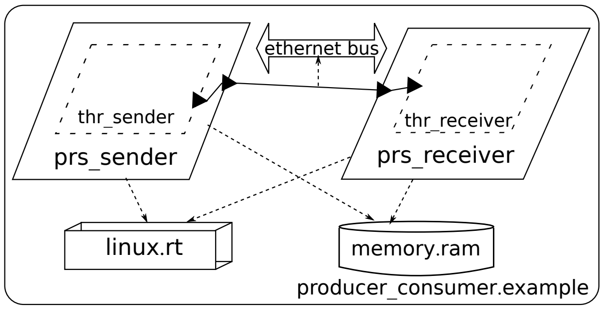

Fig. 4. AADL producer/consumer

C. 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 [21] toolset.

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 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).

```
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

2) Kernel modeling: a MILS kernel is specified using the `processor` component. It models both hardware and the associated software to isolate partitions.

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.

4) 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, specified in listing 2, contains two partitions: the first one is executed during 500ms while the second one is executed for 1s. By default, we assume the runtime executes partitions with a cyclic round-robin scheduling protocol.

```
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 : mils_kernel.i;
   p1     : topsecret_partition.i;
   p2     : 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

5) 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.

6) 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.

IV. 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 [6], an AADL-dedicated constraint language. Requirements are expressed through theorems that are validated thanks to a dedicated tool.

A. 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.

B. 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.

C. 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.

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

E. 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 [3] or flow latency [5]) can be issued on MILS systems, checking various system requirements and increasing the reliability of the development process.

F. 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 IV-E to code generation (described in next sections).

V. Automatic Implementation

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

1) The **code generation** process (supported by Ocarina [21]) which creates partitions and kernel code (illustrated with dashed boxes in figure 5)
2) The **MILS operating system** (POK [2]) which provides isolation services (solid boxes in figure 5)

Next subsections detail each step of this process.

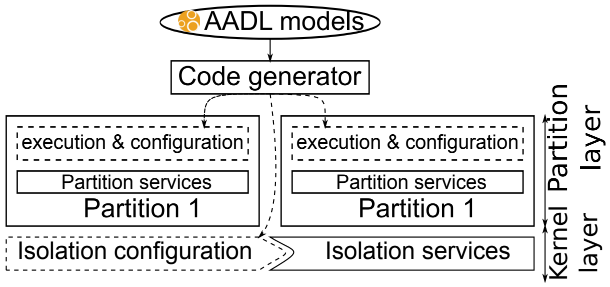

Fig. 5. Detail of our implementation process

A. Code generation

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 modeling partitions: `process`, `threads`, `data` and their `features` (communication ports).

For partition 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 when an interface (AADL `port`) uses a security layer.

Automatic configuration of cipher algorithms ensures data encryption according to the specification. It avoids potential security threat 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).

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.

B. 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.

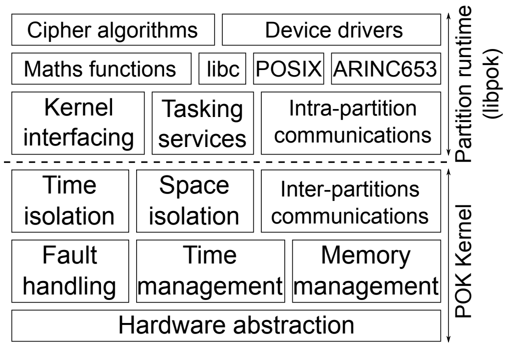

Fig. 6. POK architecture

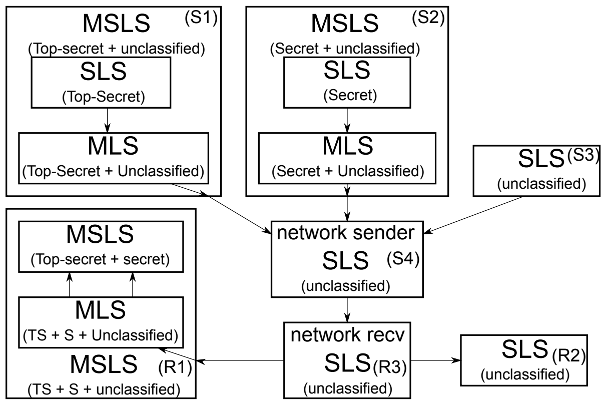

Fig. 7. Overview of our case study

1) Kernel layer: Our design guidelines lead us to keep it as small as possible 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. This functionnality ensures that only allowed partitions can communicate, avoiding covert channels.

2) Partition runtime: Due to its lower criticality, this layer contains more services than the kernel. The core layer (services depicted at the bottom in figure 6) provides tasking, intra-partition communication functions.

On top of this core layer, compatibility layers supports several standards (such as POSIX or ARINC653) to ease application portability. These are independent and rely only on the core layer.

It also includes cipher algorithms to crypt data. They are then used by the generated code to crypt/uncrypt data before sending/receiving them.

Finally, this layer provides device drivers. In POK, drivers are executed in partitions, implementing them in the kernel could potentially break isolation services.

VI. Case Study

A. 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 transfered through an unclassified channel, which is a typical case-study when we share classified data over a network.

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

To send classified data at an unclassified security level, we downgrade/upgrade data security-level. Data sent or received by SLS components S1, S2 or R1 is modified by MLS components. To ensure data protection, components that handle classified data encrypt/decrypt it before/after sending/receiving to/from unclassified entities.

SLS components S4 and R3 represent software that manages network hardware. It is responsible to send/receive packets on the network. They are not verified and so, transport data only at an unclassified security level.

We map this abstract architecture in AADL using our modeling rules (cf. figure 8). It is a distributed system deployed on two nodes sharing classified information on 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 R1 and S1, 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).

Ocarina [21] automatically generates code for nodes from this architecture model. This code is then integrated with POK [2], our MILS O/S to produce a final implementation. Next section reports our results, showing encryption correctness as well as compliance with safety-critical system needs.

B. Validation

We first validate our architecture against our REAL theorems (cf. section IV). No error was reported, showing the specification preserves security isolation.

Then, we check for correctness of the security implementation. For that purpose, we execute generated applications in an emulator connected through a virtual ethernet bus. We capture the network traffic produced by each partition and inspect each ethernet frame using the *wireshark* [13] tool. The tool reports

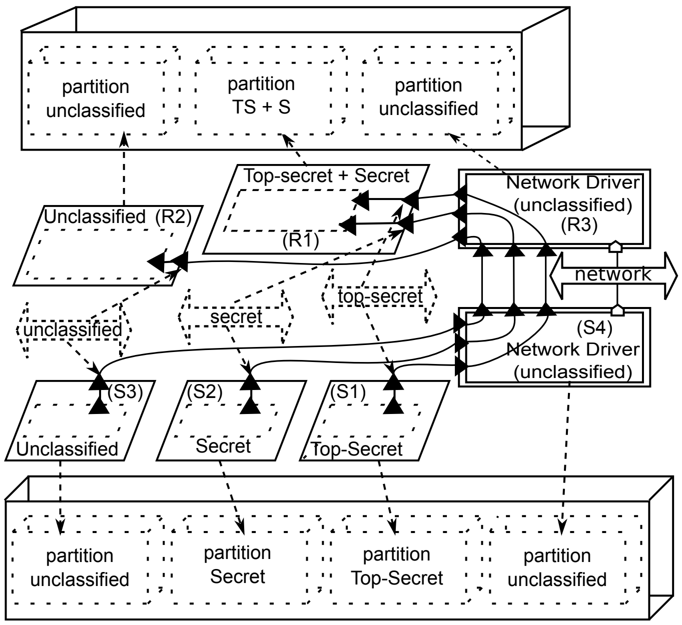

Fig. 8. AADL architecture

that captured data exchanged across partitions was encrypted, so that an attacker could not read it. In addition, partitions that receives data print expected results, showing that uncryption mechanisms are also well implemented.

VII. 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.

A. *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 symetric cipher algorithms but it can be extended with asymetric 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] "Common Criteria for Security Evaluation – http://www.commoncriteriaportal.org."

[2] J. Delange, *POK user guide - http://pok.gunnm.org*.

[3] J. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon, "Validate, simulate, and implement ARINC653 systems using the AADL," *Ada Lett.*, vol. 29, no. 3, pp. 31–44, 2009.

[4] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The Architecture Analysis and Design Language (AADL): An Introduction," Tech. Rep., 02 2006.

[5] P. H. Feiler and J. Hansson, "Flow Latency Analysis with the Architecture Analysis and Design Language (AADL)," SEI, Tech. Rep., 2007.

[6] O. Gilles and J. Hugues, "Validating requirements at model-level," in *Ingénierie Dirigée par les modèles (IDM'08)*, Mulhouse, France, jun 2008, pp. 35–49.

[7] J. Hansson, P. H. Feiler, and J. Morley, "Building secure systems using model-based engineering and architectural models," *Crosstalk*, September 2008.

[8] J. Hansson and A. Greenhouse, "Modeling and Validating Security and Confidentiality in System Architectures," CMU/SEI, Tech. Rep., 2008.

[9] Information Assurance Directorate, "U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness," 2007.

[10] National Institute of Standards and Technology (NIST), "The Economic Impacts of Inadequate Infrastructure for Software Testing," Tech. Rep., 2002.

[11] Object Management Group (OMG), "Systems Modeling Language (SysML)," 2007.

[12] OMG, *A UML Profile for MARTE, Beta 1*. OMG Document Number: ptc/07-08-04, Aug. 2007.

[13] A. Orebaugh, G. Ramirez, J. Burke, and L. Pesce, *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*. Syngress Publishing, 2006.

[14] A. Rodriguez, E. Fernandez-Medina, and M. Piattini, "Security requirement with a UML 2.0 profile," 2006. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1625372

[15] J. Rushby, "The design and verification of secure systems," in *Eighth ACM Symposium on Operating System Principles (SOSP)*, Asilomar, 1981, pp. 12–21, (ACM *Operating Systems Review*, Vol. 15, No. 5).

[16] ——, "Separation and Integration in MILS (The MILS Constitution)," SRI International, Tech. Rep., 2008.

[17] SAE, *Architecture Analysis & Design Language v2.0 (AS5506)*, September 2008.

[18] G. Uchenick and M. Vanfleet, "Multiple Independent Levels of Safety and Security: High Assurance Architecture for MSLS/MLS," in *Military Communications Conference, 2005. MILCOM*, IEEE, Ed., 2005.

[19] W. M. Vanfleet, J. A. Luke, R. W. Beckwith, C. Taylor, B. Calloni, and G. Uchenick, "MILS:Architecture for High-Assurance Embedded Computing," *Crosstalk*, 2005.

[20] WW Technology Group, "EDICT Tool Suite - http://www.wwtechnology.com/."

[21] B. Zalila, J. Hugues, and L. Pautet, *Ocarina user guide*, TELECOM ParisTech.

[22] J. Zhou and J. Alves-Foss, "Architecture-based refinements for secure computer systems design," 2006.